PHP. Bezpieczne programowanie

Autor: Chris Shiflett
Tłumaczenie: Sławomir Dzieniszewski
ISBN: 83-246-0425-1
Tytuł oryginału: Essential PHP Security
Format: B5, stron: 128

Bezpieczeństwo danych w sieci to temat, który jest ostatnio poruszany niezwykle często. Serwery internetowe zajmujące się przetwarzaniem transakcji elektronicznych, wyświetlaniem stron WWW i przesyłaniem danych stały się ulubionym celem ataków komputerowych przestępców. Kluczowym zagadnieniem jest więc bezpieczeństwo aplikacji działających na tych serwerach. Aplikacje napisane w najpopularniejszym języku, w PHP, stanowią dla hakerów łakomy kąsek. Nie jest to jednak wina języka, a raczej twórców aplikacji, którzy w projektach nie uwzględniają mechanizmów obronnych.

Książka „PHP. Bezpieczne programowanie" zawiera przegląd metod pozwalających na ochronę aplikacji internetowych przed różnymi rodzajami ataków. Czytając ją, nauczysz się projektować bezpieczne formularze, zapobiegać przechwytywaniu informacji z baz danych oraz zabezpieczać mechanizmy sesji. Dowiesz się, w jaki sposób uchronić się przed kradzieżą danych oraz uniemożliwić atak polegający na wstrzykiwaniu poleceń i kodu SQL. Poznasz także ogólne zasady ochrony kodu źródłowego.

- Ataki na formularze
- Zabezpieczanie przed wykonywaniem skryptów
- Ochrona baz danych
- Zabezpieczanie mechanizmów sesji i danych logowania
- Uniemożliwianie uruchamiania obcych aplikacji
- Ochrona systemu plików na serwerze
- Utrzymywanie aplikacji na współdzielonym serwerze i eliminowanie związanych z tym zagrożeń

Poznaj różne rodzaje ataków i stwórz mechanizmy obronne

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

ROZDZIAŁ 3.

Bazy danych i SQL

Bardzo często kod PHP wykorzystywany jest jako połączenie między różnymi źródłami danych a użytkownikiem. Prawdę powiedziawszy, niektórzy twierdzą, że język PHP jest raczej platformą programową niż zwykłym językiem programowania. Jest on na przykład bardzo często wykorzystywany do komunikacji z bazą danych.

Język PHP jest dobrze przystosowany do tej roli głównie z uwagi na to, że potrafi się kontaktować z długą listą różnych baz danych. Oto tylko kilka najbardziej popularnych systemów baz danych, które język PHP obsługuje:

DB2	ODBC	SQLite
InterBase	Oracle	Sybase
MySQL	PostgreSQL	DBM

Tak jak jest w przypadku każdego zewnętrznego miejsca składowania danych, tak i bazy danych niosą ze sobą pewne ryzyko. Mimo iż w niniejszej książce nie zajmujemy się bezpieczeństwem baz danych, powinniśmy pamiętać, że pisząc aplikacje WWW, należy mieć na względzie problemy związane z bezpieczeństwem baz danych, w szczególności w sytuacjach, kiedy należy traktować dane otrzymywane z bazy danych jako dane zewnętrzne (dane wejściowe).

Tak jak wspomniałem w rozdziale 1., wszelkie dane wejściowe powinny być filtrowane, a wszelkie dane zwracane (dane wyjściowe) powinny być opatrywane odpowiednimi znakami ucieczki. W odniesieniu do komunikacji z bazami danych będzie to oznaczać, że wszystkie dane nadchodzące z bazy danych muszą być filtrowane, a wszelkie dane wysyłane do bazy muszą być opatrzone znakami ucieczki.

Jeden z typowych błędów polega na zapominaniu, że zapytanie `SELECT` również wysyła dane do bazy danych. Mimo iż jego zadaniem jest pobieranie danych, samo jednak zawiera dane wyjściowe wysyłane przez aplikacje.

Wielu programistów PHP zapomina o konieczności filtrowania danych nadchodzących z bazy danych, ponieważ w bazie przechowywane są zazwyczaj wyłącznie dane już przefiltrowane. Mimo iż ryzyko związane z tym zaniedbaniem nie jest duże, ja jednak zalecałbym, aby nie ułatwiać sobie w ten sposób życia. Podejście to bowiem pokłada zbyt wielkie zaufanie w bezpieczeństwie bazy danych, a ponadto łamie zasadę tworzenia w miarę możliwości dodatkowych zabezpieczeń. Należy pamiętać, że wprowadzanie takich dodatkowych zabezpieczeń

wzmacnia ogólne bezpieczeństwo aplikacji i komunikacja z bazą danych jest znakomitym przykładem korzyści płynących z zaimplementowania dodatkowych mechanizmów ochronnych. Jeśli komuś uda się w jakiś sposób wprowadzić do bazy danych niebezpieczne dane, to odpowiedni mechanizm filtrowania danych w aplikacji je wyłapie, pod warunkiem oczywiście, że będzie istniał.

W niniejszym rozdziale przyjrzymy się paru problemom związanym z bezpieczeństwem komunikacji z bazami danych, między innymi ujawnieniu uwierzytelnień dostępu czy wstrzykiwaniu kodu SQL. Szczególną uwagę należy zwrócić właśnie na wstrzykiwanie kodu SQL z uwagi na to, że często wykrywa się w popularnych aplikacjach PHP podatność na takie ataki.

Ujawnienie uwierzytelnień dostępu

Jednym z ważnych problemów związanych z korzystaniem z baz danych jest niebezpieczeństwo ujawnienia uwierzytelnień umożliwiających dostęp do bazy danych — czyli nazwy użytkownika i hasła. Często są one dla wygody zachowywane w pliku takim jak *db.inc*:

```
<?php

$db_user = 'myuser';
$db_pass = 'mypass';
$db_host = '127.0.0.1';

$db = mysql_connect($db_host, $db_user, $db_pass);

?>
```

Zarówno nazwa użytkownika `myuser`, jak i hasło `mypass` są szczególnie ważne dla bezpieczeństwa bazy danych i aplikacji, dlatego też należy zwrócić na nie szczególną uwagę. Umieszczanie ich w kodzie źródłowym wiąże się z pewnym ryzykiem, którego jednak raczej nie da się uniknąć. Bez nich nasza baza danych nie mogłaby być chroniona za pomocą nazwy użytkownika i hasła.

Jeśli przyjrzymy się plikowi *httpd.conf* (standardowemu plikowi konfiguracyjnemu serwera WWW Apache), przekonamy się, że domyślny typ zawartości zdefiniowany został tam jako `text/plain`. Ustawienie takie jest szczególnie niebezpieczne, gdy plik z hasłami, taki jak *db.inc*, jest przechowywany w katalogu dokumentów WWW. Każdemu dokumentowi przechowywanemu w katalogu dokumentów WWW przypisany zostaje bowiem odpowiedni adres URL i ponieważ serwer Apache nie określa zazwyczaj specjalnego typu zawartości dla plików *.inc*, więc każde żądanie tego rodzaju zasobu zwróci plik z hasłami w postaci zwykłego pliku tekstowego (domyślny typ zawartości). W ten sposób żądający bez trudu uzyska dostęp do uwierzytelnień dostępu bazy danych.

Aby lepiej wyjaśnić zagrożenia związane z taką sytuacją, załóżmy, że mamy serwer WWW, w którym dokumenty przechowywane są w katalogu */www*. Jeśli plik *db.inc* będzie przechowywany w podkatalogu *www/inc*, to otrzyma własny adres URL — *http://example.org/inc/dbinc* (zakładając, że adres internetowy hosta serwera WWW to *example.org*). Zaglądając pod ten adres URL, użytkownik będzie mógł obejrzeć kod źródłowy pliku *db.inc* w postaci zwykłego tekstu (jak to określa domyślny typ zawartości). Dlatego też, jeśli plik *db.inc* przechowywany jest w którymkolwiek z podkatalogów katalogu dokumentów */www*, zawsze istnieje ryzyko ujawnienia uwierzytelnień dostępu osobom niepowołanym.

Najlepszym rozwiązaniem tego problemu jest przechowywanie plików załączanych (z rozszerzeniem *.inc* od ang. includes) poza katalogiem dokumentów WWW. Aby korzystać z nich za pomocą dyrektyw `include` czy `require`, nie trzeba ich umieszczać w żadnym ściśle określonym katalogu w systemie plików — można umieścić je w dowolnym miejscu, trzeba tylko zadbać, by serwer WWW miał uprawnienia do odczytywania plików przechowywanych w tym katalogu. Jak więc widać, umieszczanie ich w katalogu dokumentów WWW wiąże się jedynie z niepotrzebnym ryzykiem, a każda metoda, która będzie próbować zmniejszyć to ryzyko, nie przenosząc ich jednak poza katalog dokumentów WWW, będzie tylko półśrodkiem. Generalnie w katalogu dokumentów WWW należy umieszczać wyłącznie te pliki i zasoby, które koniecznie muszą być dostępne po podaniu odpowiedniego adresu URL. W końcu jest to katalog dostępny publicznie dla wszystkich użytkowników internetu.

Rada ta odnosi się również do baz SQLite. Bardzo wygodnym rozwiązaniem jest umieszczenie bazy danych w bieżącym katalogu, dzięki czemu można się do niej odwoływać, podając tylko nazwę, a nie całą ścieżkę do katalogu bazy danych. Niemniej w ten sposób umieszczamy bazę danych w katalogu dokumentów WWW i narażamy na niepotrzebne ryzyko. Baza taka może zostać skompromitowana za pomocą prostego żądania HTTP, o ile nie przedsięweźmiemy odpowiednich kroków, aby ją dodatkowo zabezpieczyć przed możliwością bezpośredniego dostępu z zewnątrz. Najbardziej polecanym rozwiązaniem pozostaje jednak umieszczenie bazy danych poza drzewem dokumentów WWW.

Jeśli z powodu jakichś czynników zewnętrznych nie będziemy mogli skorzystać z optymalnego rozwiązania polegającego na umieszczeniu wszystkich załączanych plików poza katalogiem dokumentów WWW, można zawsze skonfigurować serwer Apache, tak aby odrzucał żądania HTTP o pliki *.inc*:

```
<Files ~ "\.inc$">
    Order allow,deny
    Deny from all
</Files>
```

W rozdziale 8. opisuję metodę chronienia uwierzytelnień dostępu bazy danych szczególnie efektywną w środowiskach, gdzie na tym samym hoście działają różne programy (w takim przypadku pliki znajdujące się poza katalogiem WWW nadal narażone są ryzyko ujawnienia).

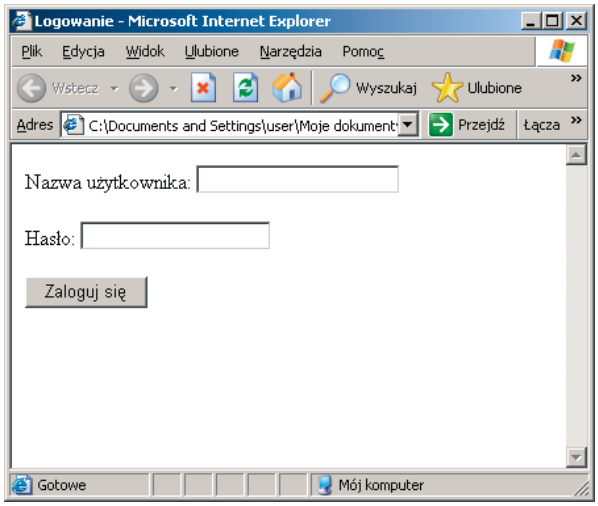
Wstrzykiwanie kodu SQL

Podatność na technikę ataku zwaną wstrzykiwaniem kodu SQL (ang. SQL injection) jest jedną z najczęstszych słabości aplikacji PHP. Jest to szczególnie zadziwiające, jako że ataki tego typu możliwe są tylko wtedy, gdy programista popełni od razu dwa poważne błędy — zapomni o filtrowaniu danych nadchodzących do aplikacji z zewnątrz (danych wejściowych) i jednocześnie zapomni o konieczności opatrywania znakami ucieczki danych wysyłanych do aplikacji (danych wyjściowych). Nie powinno się zapominać o żadnym z tych dwu podstawowych zabiegów, a tylko stosowanie obu technik jednocześnie daje gwarancję poważnego ograniczenia błędów.

Wykorzystanie techniki wstrzykiwania kodu SQL zazwyczaj wymaga ze strony atakującego odrobiny zgadywania i pewnego eksperymentowania — musi odgadnąć, jak wygląda schemat naszej bazy danych (zakładając oczywiście, że nie udało mu się zdobyć dostępu do naszego kodu źródłowego lub schematu bazy danych). Rozważmy następujący prosty formularz logowania:

```
<form action="/login.php" method="POST">
<p>Nazwa użytkownika: <input type="text" name="username" /></p>
<p>Hasło: <input type="password" name="password" /></p>
<p><input type="submit" value="Zaloguj się" /></p>
</form>
```

Rysunek 3.1 pokazuje, jak formularz ten będzie wyglądać w oknie przeglądarki.



Rysunek 3.1. Prosty formularz logowania wyświetlony w oknie przeglądarki

Haker, który zobaczy taki formularz, będzie mógł spróbować zgadnąć, za pomocą jakiego zapytania sprawdzamy wpisywaną nazwę użytkownika i hasło. Przeglądając kod źródłowy HTML, atakujący może spróbować zgadywać, jakie stosujemy konwencje nazewnicze. Typowe założenie hakera będzie takie, że nazwy używane w formularzu powinny odpowiadać nazwom kolumn w tabelach bazy danych. Oczywiście samo tylko stosowanie różnych nazw w obu przypadkach nie jest wystarczającym zabezpieczeniem.

Na podstawie tego formularza można z dużym prawdopodobieństwem zgadywać, że zapytanie wysyłane przez aplikację do bazy danych będzie miało następującą postać (i tak jest w istocie w moim przykładzie):

```
<?php

$password_hash = md5($_POST['password']);

$sql = "SELECT count(*)
        FROM   users
        WHERE  username = '{$_POST['username']}'
        AND    password = '$password_hash'";

?>
```

Typowego zabezpieczenia polegającego na szyfrowaniu hasła użytkownika za pomocą algorytmu MD5 nie można już dłużej traktować jako wystarczającej ochrony. Ostatnimi czasy wykryto kilka słabych punktów algorytmu szyfrującego MD5. Ponadto wiele baz danych korzystających z tego algorytmu szyfrującego osłabia jego skuteczność, starając się uprościć mechanizm odszyfrowywania danych zaszyfrowanych algorytmem MD5. Odpowiednie przykłady można znaleźć pod adresem: *http://md5.rednoize.com/*.

Najlepszym rozwiązaniem jest specjalne znakowanie haseł użytkowników poprzez dodawanie do nich łańcucha, który będzie unikatowy dla naszej aplikacji, na przykład w ten sposób:

```
<?php

$salt = 'SHIFLETT';
$password_hash = md5($salt . md5($_POST['password'] . $salt));

?>
```

Oczywiście atakujący nie musi poprawnie odgadnąć schematu bazy danych już za pierwszym razem. Prawie zawsze musi najpierw przeprowadzić parę eksperymentów. Dobrym przykładem takiego eksperymentu jest przesłanie aplikacji pojedynczego cudzysłowu jako nazwy użytkownika, ponieważ w ten sposób haker może zdobyć kilka istotnych informacji. Wielu programistów, obsługując błędy pojawiające się w trakcie wykonywania zapytań skierowanych do baz danych, korzysta z odpowiednich funkcji takich jak `mysql_error()`. Tutaj pokazuję przykład takiego podejścia:

```
<?php

mysql_query($sql) or exit(mysql_error());

?>
```

Jednak rozwiązanie takie, choć bardzo pomocne podczas programowania, jeśli pozostawimy je w gotowej aplikacji, może zdradzić atakującemu bardzo istotne informacje. Jeśli haker prześle jako nazwę użytkownika pojedynczy cudzysłów, a jako hasło `mypass`, zapytanie wysłane przez aplikację przyjmie następującą postać:

```
<?php

$sql = "SELECT *
        FROM   users
        WHERE  username = '''
        AND    password = 'aO29dOdf84eb5549c641eO4a9ef389e5'";

?>
```

Jeśli takie zapytanie zostanie wysłane do bazy MySQL, tak jak zademonstrowałem to w poprzednim przykładzie, to użytkownikowi wyświetlony zostanie następujący błąd:

```
You have an error in your SQL syntax. Check the manual that corresponds to your
MySQL server version for the right syntax to use near 'WHERE username = ''' AND
password = 'aO29dOdf84eb55
```

Jak widać, bez wielkiego wysiłku atakujący poznał nazwy dwóch kolumn w naszej bazie danych (`username` i `password`) oraz porządek, w jakim pojawiają się w zapytaniu. Dodatkowo haker już wie, że dane nie są odpowiednio filtrowane (nie został wyświetlony błąd aplikacji informujący o nieprawidłowej nazwie użytkownika) ani opatrywane znakami ucieczki (pojawił

się błąd bazy danych), a ponadto poznał całą składnię klauzuli `WHERE`. Znając format klauzuli `WHERE`, atakujący może teraz wpływać na to, które rekordy będą dopasowywane przez zapytanie.

W tym momencie intruz ma wiele możliwości. Może na przykład spróbować przygotować zapytanie, które zostanie zakwalifikowane jako prawidłowe niezależnie od tego, czy uwierzytelnienia dostępu będą właściwe, podając następującą nazwę użytkownika:

```
myuser' or 'foo' = 'foo' --
```

Zakładając, że jako hasło poda `mypass`, wygenerowane zapytanie przyjmie postać:

```
<?php

$sql = "SELECT *
        FROM   users
        WHERE  username = 'myuser' or 'foo' = 'foo' --
        AND    password = 'a029d0df84eb5549c641e04a9ef389e5'";

?>
```

Ponieważ `--` oznaczają początek komentarza SQL, baza danych uzna, że w tym momencie zapytanie się kończy. Dzięki temu użytkownik będzie się mógł zalogować bez konieczności podawania prawidłowej nazwy użytkownika i hasła.

Jeśli z kolei atakujący będzie znał jedną z prawidłowych nazw użytkownika, na przykład `chris`, to może podać nazwę użytkownika w sposób następujący:

```
chris' --
```

Jeśli nazwa użytkownika `chris` okaże się prawidłowa, to atakujący będzie mógł przejąć kontrolę nad kontem tego użytkownika, ponieważ zapytanie przyjmie następującą postać:

```
<?php
$sql = "SELECT *
        FROM   users
        WHERE  username = 'chris' --
        AND    password = 'a029d0df84eb5549c641e04a9ef389e5'";
?>
```

Na szczęście przed atakami opartymi na technice wstrzykiwania kodu SQL można się dość łatwo zabezpieczyć. Jak wspomniałem w rozdziale 1., należy zawsze filtrować dane wejściowe i opatrywać znakami ucieczki dane wyjściowe.

Mimo iż oczywiście żadnej z tych operacji nie należy zaniedbywać, to nawet wykonanie tylko jednej z nich pozwala na wyeliminowanie ryzyka związanego z atakami polegającymi na wstrzykiwaniu kodu SQL. Jeśli na przykład przefiltrujemy dane wejściowe, ale zapomnimy opatrzyć dane wyjściowe znakami ucieczki, to aplikacja nadal może wyświetlać błędy bazy danych (nawet prawidłowe dane mogą spowodować wygenerowanie niepoprawnego zapytania SQL), mało jest jednak prawdopodobne, że właściwe dane zmienią zachowanie zapytania w sposób nieprzewidywalny. Z kolei, jeśli opatrzymy znakami ucieczki dane wyjściowe, ale zapomnimy o filtrowaniu danych wejściowych, to dzięki opatrzeniu danych wysyłanych do bazy znakami ucieczki będziemy mieli gwarancję, że dane te nie będą mogły zmienić formatu zapytania SQL, co ochroni nas przed wielu typowymi atakami opartymi na technice wstrzykiwania kodu SQL.

Oczywiście należy zawsze stosować oba środki zapobiegawcze. Sposób filtrowania danych wejściowych zależeć będzie całkowicie od rodzaju filtrowanych danych (parę przykładów można znaleźć w rozdziale 1.), natomiast opatrywanie znakami ucieczki danych wysyłanych do bazy wymaga zazwyczaj przygotowania tylko jednej funkcji. W przypadku użytkowników bazy MySQL będzie to funkcja `mysql_real_escape_string()`:

```
<?php

$clean = array();
$mysql = array();

$clean['last_name'] = "O'Reilly";
$mysql['last_name'] = mysql_real_escape_string($clean['last_name']);

$sql = "INSERT
        INTO   user (last_name)
        VALUES ('{$mysql['last_name']}')";

?>
```

Zawsze należy się starać używać funkcji właściwej dla naszej bazy danych, jeśli tylko język PHP ją oferuje. Jeśli nie, ostatnią deską ratunku jest zawsze funkcja `addslashes()`.

Jeśli wszystkie dane służące do przygotowania zapytania zostaną odpowiednio przefiltrowane i opatrzone znakami ucieczki, to ryzyko ataków typu wstrzykiwania kodu SQL spada do zera.

Jeśli korzystamy z bazy danych, która pozwala na korzystanie z parametrów wiążących lub zmiennych zastępczych (takiej jak PEAR::DB czy PDO), to możemy dodać jeszcze jedną warstwę zabezpieczeń. Na przykład załóżmy, że przesyłamy do bazy PEAR::DB następujące zapytanie (dodające do tabeli użytkowników nazwiska użytkowników):

```
<?php
$sql = 'INSERT
        INTO   user (last_name)
        VALUES (?)';
$dbh->query($sql, array($clean['last_name']));
?>
```

Ponieważ przesyłane dane nie mogą teraz wpłynąć na format zapytania, ryzyko udanego ataku techniką wstrzykiwania kodu SQL jeszcze się zmniejsza. System baz danych PEAR::DB automatycznie będzie opatrywać dane znakami ucieczki i ujmować je w cudzysłowy, dzięki czemu trafią do bazy danych w bezpiecznej postaci i nasza odpowiedzialność ograniczona zostanie do konieczności odpowiedniego filtrowania danych wejściowych.

Jeśli użyjemy parametrów wiążących, to nasze dane nigdy nie znajdą się w kontekście, w którym mogłyby zostać potraktowane jako cokolwiek innego niż dane przesyłane do bazy. Dzięki temu nie będziemy musieli opatrywać danych wysyłanych znakami ucieczki, ponieważ zabieg ten nie będzie teraz niczego zmieniał (jeśli mimo wszystko zdecydujemy się na dodanie znaków ucieczki), bowiem nie będziemy wysyłać żadnych znaków, które będą musiały zostać potraktowane w ten sposób. Parametry wiążące są jednym z najlepszych zabezpieczeń przeciw wstrzykiwaniu kodu SQL.

Ujawnienie danych przechowywanych w bazie

Kolejnym ryzykiem związanym z bazami danych, które należy uwzględnić, jest niebezpieczeństwo ujawnienia osobom postronnym ważnych danych przechowywanych w bazie. Jeśli przechowujemy tam informacje takie jak numery kart kredytowych, numery PESEL, numery ubezpieczenia społecznego (w USA), lub też inne podobnie ważne dane, należy bezwzględnie się upewnić, że informacje przechowywane w bazie danych będą bezpieczne.

Mimo iż problemy związane z bezpieczeństwem bazy danych nie wchodzą w zakres tej książki (i przeważnie nie należą do zakresu kompetencji programisty tworzącego aplikację PHP), warto wiedzieć, że aby je dodatkowo zabezpieczyć, można szyfrować najbardziej istotne dane. Dzięki temu nawet zdobycie przez osobę postronną zawartości bazy danych nie będzie wielką tragedią, pod warunkiem, że osoba ta nie wejdzie w posiadanie klucza szyfrującego. Więcej informacji na temat szyfrowania i kryptografii można znaleźć w dodatku C.