Tomasz Trejderowski

SQL injection

Praktycznie każdy portal korzystający z baz danych wrażliwy jest na ataki SQL injection. W poprzednim numerze wspomnieliśmy tylko o tej tematyce – dziś przyjrzymy jej się z bliska, oczywiście bez zbędnych powtórzeń.

Ataki typu SQL injection przestały być atakami początkujących dowcipnisiów, a stały się atakami największych hakerów świata, złodziei danych i tożsamości. Nazwa pochodzi od angielskiego *injection* – wtrącenie, wstawienie, intruzja. Generalnie chodzi tu o błędy w skryptach PHP przekazujących całość lub fragment danych wprowadzonych przez użytkownika bezpośrednio do bazy. Przesyłanie ich bez wcześniejszej, najprostszej choćby analizy, jest poważnym błędem – takie podejście daje osobie trzeciej możliwość zmodyfikowania wprowadzanych danych tak, by uzyskać dostęp do naszych tajemnic.

Bezradne systemy bezpieczeństwa

Ataki typu SQL injection są przeprowadzane pomiędzy warstwą programistyczną (język aplikacji internetowej) a warstwą danych (system zarządzania bazą danych). Nie przeszkadzają im więc programy antywirusowe czy firewalle, ponieważ te działają na niższym poziomie. Podobnie bezradne wobec nich są systemy wykrywania intruzów – IDS (ang. *Intrusion Detec-*

Co należy wiedzieć?

- Jak z poziomu PHP obsługiwać bazy danych,
- Jak przesyłane są dane pomiędzy zamieszczonym na stronie formularzem, a bazą danych.

Co obiecujemy?

- Umiejętność dostrzeżenia błędów w tworzonych przez siebie aplikacjach,
- Umiejętność tworzenia skryptów zabezpieczonych przed SQL injection.

Na płycie

W PHP Solutions live konfiguracja umożliwia przetestowanie opisywanych w artykule problemów z bezpieczeństwem.

tion System). W przypadku ataków typu SQL injection nie można stworzyć jednolitych sygnatur ataku, gdyż ataki te przeprowadzane są jednorazowo, na potrzeby konkretnego włamania, a ich struktura za każdym razem może wyglądać zupełnie inaczej.

Bagatelizowane niebezpieczeństwo

Ataki typu SQL injection są niebezpieczne nie ze względu na swoją naturę, ale dlatego, iż większość programistów internetowych niewiele wie o ich istnieniu albo nie potrafi spojrzeć na swój kod z punktu widzenia włamywacza. Niezwykle trudno jest im uwierzyć, że do ich serwisu może zalogować się ktoś, kto zna tylko login pierwszego lepszego użytkownika, lub zgoła ktoś, kto w ogóle nie zna żadnych parametrów logowania.

Zanim zaczniemy

Nim przejdziemy do meritum, poruszmy jeszcze dwie kwestie, które są tak oczywiste, że wiele osób o nich po prostu zapomina.

Po pierwsze: wiele osób używa phpMyAdmina do zarządzania swoimi bazami danych. Należy pamiętać, aby nie konfigurować go na tryb *config*. Przy takich ustawieniach każdy, kto zgadnie ścieżkę dostępu do phpMyAdmin (a nie jest to wielki problem biorąc pod uwagę, że najczęściej umieszczamy go w ścieżce */admin*, */phpMyAdmin*, czy */pma*), może przeglądać zawartość naszych baz danych. Takie rozwiązanie wchodzi w grę tylko i wyłącznie, gdy testujemy nasze aplikacje na lokalnym serwerze (*localhost*), jednak na serwerach podłączonych do Internetu absolutnie nie wchodzi w rachubę!

Drugi problem to bezpośredni dostęp do bazy danych. Niektórzy programiści na etapie budowania i testowania swojej kolejnej aplikacji internetowej tworzą furtki pozwalające na natychmiastowe wykonanie zapytania do bazy danych, bez konieczności wykorzystywania do tego phpMyAdmina czy uruchomionej lokalnie konsoli MySQL. Przykład takiej furtki przedstawia Listing 1.

Programista tworzy sobie najprostszy formularz zawierający jedno pole tekstowe:

```
<input size="49" name="form_query"
       value="">
```

o nazwie `form_query`. Odpowiednio skonstruowany nagłówek (znacznik `<form>`) tego formularza powoduje skierowanie jego zawartości do kodu przedstawionego na Listingu 1. Ten zaś wykona dokładnie takie zapytanie, jakie wpisaliśmy do pola tekstowego.

Rozwiązanie takie może być stosowane na czas testowania naszej aplikacji i tylko wtedy! Gdyby komuś z zewnątrz udało się uruchomić skrypt, to biada naszym danym! W tej wersji kodu nie widać wyników działania wysyłanych zapytań. Lecz mimo to sam fakt, że nie jest konieczne znanie loginu i hasła, by wejść do naszej bazy, wygląda bardzo niepokojąco.

Niebezpieczny formularz logowania

Ten przykład to najprostszy (i zarazem najbardziej niebezpieczny) sposób umożliwienia intruzowi ataku typu SQL injection. Na podstawie korespondencji z innymi programistami PHP jesteśmy w stanie zaryzykować twierdzenie, iż przedstawiony sposób (umożliwiający włamanie) jest stosowany częściej niż techniki bezpieczne. Natomiast zaprezentowany sposób logowania użytkownika nie umożliwia ataku intruzyjnego, dlatego podamy go jako receptę. Ale zanim do niego dojdziemy, przedstawmy istotę problemu. Spójrzmy na kod przedstawiony na Listingu 2. Służy on do zalogowania użytkownika, jeśli poda on prawidłowy login i hasło. Zmienne `$login_form` i `$pass_form`

Listing 1. *Przykład furtki do natychmiastowego wykonywania zapytań MySQL*

```
<?php
   connect_to_database();
   mysql_query($form_query);
?>
```

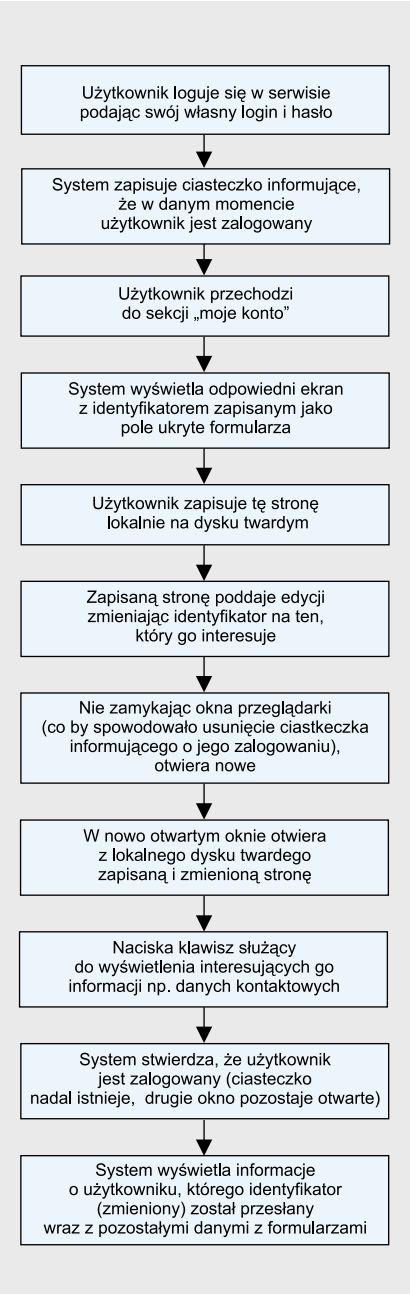
to zawartość odpowiednich pól formularza służącego do logowania: użytkownik podaje w stworzonym dla niego formularzu login i hasło, po czym klika na klawisz powodujący wysłanie tych danych do powyższego kodu. Nasz skrypt natomiast pobiera z bazy danych wszystkie wiersze, dla których spełnione są jednocześnie oba warunki. Ponieważ w bazie danych może istnieć tylko jeden użytkownik o zadanej nazwie, więc tylko podanie prawidłowego loginu i hasła powoduje zwrócenie liczby wierszy różnej od zera – a tym samym uruchomienie dalszych procedur.

Załóżmy teraz teoretyczną sytuację, w której użytkownik wpisuje jako login `login`, zaś jako hasło `'hasło' or 0=0`. Teoretycznie nasz skrypt powinien zareagować negacją (w kodzie na Listingu 2 polecenia po `else`). To przecież oczywiste – użytkownik o loginie *login* nie istnieje w naszej bazie danych. Okazuje się jednak, że to nieprawda. Dlaczego? Przyjrzyjmy się jak wygląda zapytanie MySQL wysyłane do bazy danych, w sytuacji podania powyższych parametrów:

```
SELECT * FROM users WHERE
  login='login' AND
  pass='hasło' or 0=0
```

Pomimo, że użytkownik taki faktycznie nie istnieje i pierwsze dwa warunki zwrócą `FALSE`, trzeci warunek (`0=0`) jest zawsze prawdziwy, więc zawsze zwraca `TRUE`. W sumie powyższe zapytanie przesłane do bazy danych MySQL jest więc równoważne zapytaniu:

```
SELECT * FROM users
```



Rysunek 1. *Schemat obrazujący nieuprawnione wykorzystanie poufnych danych przez zarejestrowanego użytkownika*

Podanie więc takich przykładowych parametrów umożliwi wybranie wszystkich wierszy z tabeli. To z kolei spowoduje, że będzie spełniony warunek sprawdzający, czy liczba wybranych wierszy jest nierówna zero i... użytkownik zostanie zalogowany!

Ktoś może w tym momencie zwrócić uwagę, że generalnie nie jest to dla niego żaden problem, gdyż w przypadku jego serwisu nazwa jest zapisywana do cookie. A więc tak „zalogowany” użytkownik i tak nie będzie mógł

Listing 2. *Procedura logowania użytkownika umożliwiająca ataki typu SQL injection*

```
<?php
   $query=mysql_query("SELECT * FROM users WHERE login='login_form'
                                              AND pass='$pass_form'");
   $lcount=mysql_num_rows($query);
   if($lcount !=0)
   {
      echo 'zalogowano';
      //Inne operacje związane z zalogowaniem użytkownika
   }
   else
   {
      echo 'błędny login lub hasło';
      //Inne operacje związane z podaniem błędnych danych
   }
?>
```

skorzystać z żadnej funkcji dostępnej tylko dla osób zarejestrowanychw, bo za każdym razem jego nazwa będzie wracała z serwera MySQL jako nieistniejąca.

Problem rzeczywiście nie istnieje, jeśli osoba próbująca przeprowadzić atak nie zna loginu dowolnego użytkownika naszego serwisu. Jeśli jednak go zna (a poznać go to w sumie żaden problem: jeśli tylko portal ma forum, wiadomości publikowane na nim sygnowane są nazwami użytkowników), to zastosowanie jako procedury logującej kodu z Listingu 2 urasta nam do rangi katastrofy. Możemy bowiem (wykorzystując SQL injection) zalogować się do serwisu jako dowolny użytkownik, nie znając jego hasła! Jak tego dokonać? Wystarczy jako login podać nazwę dowolnego użytkownika, na końcu dopisać `';--`, zaś pole hasła pozostawić puste.

Załóżmy, że używamy loginu użytkownika *admin*, a więc w pole *login* wpisujemy ciąg `admin';--`. W tym przypadku zapytanie MySQL przesłane do bazy danych wyglądałoby tak:

```
SELECT * FROM users WHERE
  login='admin';--' AND
  pass='hasło' or 0=0
```

Być może niektórzy dostrzegają już istotę problemu. Otóż ciąg `;--` (średnik i dwa minusy) powoduje, że dalszy ciąg zapytania MySQL jest ignorowany. Użyty przez nas system baz danych (w tym przypadku MySQL) zachowa się więc tak, jak gdyby zapytanie to wyglądało tak:

```
SELECT * FROM users WHERE
  login='admin';--
```

Czyli (zgodnie z prawdą) zwróci w odpowiedzi wiersz z tej tabeli zawierający dane użytkownika, którego login podaliśmy. Następująca po nim funkcja `mysql_num_rows($query)` przyporządkuje do zmiennej `$lcount` wartość 1 (tyle wierszy zwróciła baza danych), co ponownie pozwoli oszukać linię sprawdzającą ilość zwróconych wierszy w odpowiedzi i zaloguje danego użytkownika. Problem w tym, że tym razem będzie to istniejący w bazie danych użytkownik. A więc intruz pomyślnie zaloguje się w naszym serwisie znając jedynie login użytkownika, którego (zapewne przypadkiem) wybrał na swoją ofiarę. Jeśli nasz system w jakikolwiek sposób korzysta z realnych finansów (np. mikropłatności), a nie stosujemy dodatkowych zabezpieczeń, intruz może korzystać z „płatnych” sekcji serwisu na konto swojej ofiary. Ta zaś, niczego się nie spodziewając, zaloguje się następnego dnia i z przerażeniem stwierdzi, że jej konto jest... puste!

Dwa i więcej zapytań

Z jeszcze większym problemem mają do czynienia użytkownicy baz danych typu PostgreSQL lub MS-SQL. PHP bowiem nie pozwala na umieszczenie więcej niż jednego zapytania w ciągu przekazanym do funkcji `mysql_query`. Dlatego też, intruz może dowolnie modyfikować jedynie podstawowe zapytanie – w tym przypadku `SELECT`. Dzięki temu ma jedynie możliwość wyświetlania informacji z bazy danych. Zaś w przypadku bazy danych PostgreSQL (MS-SQL) takich ograniczeń już nie ma. W ciągu przekazanym do funkcji `pg_query` (`mssql_query`) można podać więcej niż jedno zapyta-

nie. Załóżmy więc kolejną teoretyczną sytuację, w której intruz podaje jako login wartość `'; delete from users--`. Znajdujący się na początku znak pojedynczego cudzysłowu zakończy część zapytania dotyczącą danych wprowadzonych z zewnątrz. Następujący po nim średnik powoduje zakończenie pierwszego zapytania i rozpoczęcie kolejnego (tu: `delete from users`). Zaś występujący znak podwójnego minusa oznacza w zapytaniu rozpoczęcie komentarza, zatem dalszy ciąg zapytania zostanie zignorowany. Po przetworzeniu więc podanych przez użytkownika danych oraz zinterpretowaniu wymienionych znaków specjalnych, otrzymamy dwa zapytania PostgreSQL (MS-SQL):

```
SELECT * FROM users WHERE login = '';
delete from users
```

W wyniku ich wykonania cała tablica `users` zostanie usunięta!

Podobnie jak użytkowników MySQL, problem ten nie dotyczy również korzystających z baz danych OracleSQL, gdyż składnia SQL Oracle nie dopuszcza do wykonywania kilku komend w jednej linii.

Formularz logowania odporny na intruzje

Powstaje więc pytanie, jak zapobiegać tego typu sytuacjom. Sprawa jest znacznie prostsza, niż by to się mogło wydawać. Przede wszystkim absolutnie nie wolno przesyłać ciągów podanych przez użytkownika jako elementów składowych zapytania do bazy danych. Każdy wprowadzony parametr należy poddać najprostszej choćby analizie i filtrowaniu.

Zamiast przesyłać dane od razu, użyjemy przedstawionej na początku artykułu funkcji `get_from_database()`, aby odczytać odpowiednie wartości z bazy danych i porównać je z wartościami podanymi przez użytkownika. Rozwiązanie takie realizuje Listing 3.

Pierwszą z przedstawionych metod intruzji kod ten wychwytuje przy pierwszym sprawdzeniu. Jeśli użytkownik o podanym loginie (`$login_form`) naprawdę nie istnieje, to funkcja `get_from_database` przyporządkuje do zmiennej `$cnt` wartość `0`. Natomiast jeśli ktoś dopisze do nazwy użytkownika wspomniany ciąg `0=0`, to wartość przyporządkowana do zmiennej `$cnt` będzie równa liczbie użytkowników w bazie danych. Ale i tak spełni to warunek znajdujący się w następnej linii (`$cnt !=1`), czyli spowoduje wyświetlenie komunikatu o błędzie oraz zakończenie wykonywania skryptu.

Listing 3. *Bezpieczne logowanie użytkownika*

```
<?php

   $cnt=get_from_database("users", "where login='$login_form'","count(id)");
   if($cnt !=1)
   {
      echo 'Taki użytkownik nie istnieje!';
      exit;
   }
   $db_pass=get_from_database("users", "where login='$login_form'",
                              "pass");
   if($pass_form == $db_pass)
   {
      echo 'zalogowano';
      //Inne operacje związane z zalogowaniem użytkownika
   }
   else
   {
      echo 'błędne hasło';
      //Inne operacje związane z podaniem błędnych danych
   }
}
?>
```

Kod jest również zupełnie nieczuły na drugą metodę intruzji (poprzez dodanie do nazwy użytkownika ciągu `;--`). Jeśli przyjrzymy się składni funkcji `get_from_database`, to zauważymy, że warunek (argument `$query`, w tym przypadku równy `where login ='$login_form'`) i tak znajduje się na końcu zapytania MySQL, więc pojawienie się ciągu `;--` kompletnie niczego tu nie zmienia.

Integralny element procesu logowania – hasło – jest tu sprawdzany przez porównanie. Więc, jeśli intruz zna tylko login użytkownika, pod którego chce się podszyć, w niczym mu to nie pomoże. Dopóki nie poda prawidłowego hasła, nie zostanie zalogowany.

Atak z wewnątrz

Próba uzyskania tajnych informacji może też być spowodowana działaniem nie intruza z zewnątrz, ale legalnego użytkownika systemu. Problem ten na pewno pojawi się, jeśli przechowujesz zbyt wiele informacji w polach ukrytych formularza. Załóżmy, że w przykładowym serwisie użytkownik może kliknąć odnośnik „Moje konto”. Wyświetla mu się wtedy ekran na którym znajdują się klawisze przenoszące do odpowiednich działów informacji o użytkowniku, np. „Informacje podstawowe”, „Dane teleadresowe”, „Płatności” itd. Kliknięcie każdego z tych odnośników powoduje uruchomienie odpowiedniego fragmentu kodu, w którym musi być skądś pozyskana informacja o identyfikatorze użytkownika, dla którego mają zostać wyświetlone odpowiednie informacje. Identyfikator taki możesz zapisać jako ciastko, ukryte pole formularza lub przekazać w adresie.

Ostatnia z podanych metod jest zdecydowanie najmniej bezpieczna. Jeśli co sprytniejszy użytkownik zauważy, że do wyświetlenia jego własnych informacji np. o płatnościach wykorzystujesz taki adres: *http://adres.pl/user_show_billing.php?user_id=189543,* to bez najmniejszych oporów wprowadzi na końcu takiego adresu identyfikator użytkownika, którego informacje chce uzyskać.

Możesz przekazać informacje o identyfikatorze użytkownika jako pole ukryte formularza, np.:

```
<input type="hidden" name="id"
       value="189543">
```

Niestety taką metodę również można obejść, tylko wymaga to więcej zachodu. Diagram takiego działania został przedstawiony jako algorytm na Rysunku 1.

Należy zdać sobie z tego sprawę, że jest to także atak typu SQL injection. Jakkolwiek nie zmienialiśmy samej składni zapytania, podmienili-

śmy jeden parametr, który jest jej elementem składowym. W ten sposób uzyskaliśmy dostęp do informacji dla nas nie przeznaczonych. A dzisiaj – w dobie coraz bardziej popularnych (niestety) kradzieży tożsamości, dostęp do pełnych danych użytkownika naszego serwisu, przez innego użytkownika, może mu wyrządzić więcej szkody, niż bezpośrednie włamanie do systemu.

Teren działania

Często zdarza się, że programista nie zwraca uwagi na niebezpieczeństwa, jakie czyhają na jego aplikację (uznaną za gotową), przeniesioną z serwera na którym była testowana – *localhost* – na serwer internetowy, w pełni dostępny z Sieci. Zawsze przed publikacją gotowej aplikacji należy sprawdzić dwa podstawowe aspekty.

Po pierwsze – czy poziom przywilejów nie jest zbyt wysoki. Na czas testów i tworzenia aplikacji przyznajemy sobie uprawnienia znacznie wyższe niż tego wymaga realizacja danego zadania. Publikując następnie aplikację, należy wszystkim plikom, użytkownikom, modułom oraz dostępowi do bazy danych nadać najniższe możliwe przywileje, przy których aplikacja działa bezbłędnie.

Drugie zagadnienie dotyczy plików tymczasowych. Ile razy zdarzyło ci się, że wprowadzałeś do danego pliku (np. *users.php*) tak wiele zmian, że na wszelki wypadek postanowiłeś zapisać jego wcześniejszą kopię jako *users.php.bak*, *users.temp* czy pod dowolną inną nazwą? Jeśli teraz nieopatrznie przekopiujesz ten plik razem z innymi na serwer internetowy, to tak, jak gdybyś mówił intruzowi „Zapraszam”!

Wystarczy zła konfiguracja serwera, aby dało się obejrzeć zawartość tego pliku. Pliki tymczasowe czy poprzednie wersje skryptów stanowią dla intruzów cenne źródło informacji o nazwach bądź hasłach!

Inne sposoby obrony

Jest kilka faktów, których uświadomienie sobie i odpowiednie zareagowanie na nie, może w bardzo znacznym stopniu utrudnić przeprowadzenie ataku typu SQL injection w naszej aplikacji internetowej. Warto je poznać!

1) Aby móc konstruować skuteczne (działające) zapytania, intruz musi wiedzieć jak najwięcej o strukturze naszych baz danych. Musi znać nazwy tabel, pól i inne tego typu informacje. Do ich poznania może celowo konstruować błędne zapytania, by po analizie zwracanych komunikatów uzyskać informacje o strukturze bazy danych.
2) Zdecydowanie należy sprawdzać (filtrować, analizować) odpowiednimi procedurami i instrukcjami warunkowymi wszelkie dane pobierane z zewnątrz. Nie wolno dopuścić do sytuacji, w której dane takie są przekazywane bezpośrednio do zapytania, bez jakiejkolwiek kontroli. Przykład takiej analizy został przedstawiony w punkcie „Formularz logowania odporny na intruzje”.
3) Na każdym etapie budowania danej aplikacji należy stosować zasadę minimum przywilejów. Do rozwiązania każdego zagadnienia należy przyznawać wszystkim użytkownikom i modułom minimalne przywileje niezbędne do tego celu. Przed uruchomieniem aplikacji na serwerze internetowym należy dodatkowo sprawdzić wszystkie przywileje i obniżyć je do minimalnego wymaganego poziomu.
4) Bezwzględnie należy tworzyć grupy operacji, które mogą być uruchamiane przez użytkownika w osobne procedury. Jeśli użytkownik ma dostęp do procedury wyświetlającej zawartość bazy danych, należy wszystkie operacje powodujące jej modyfikację przenieść do innej procedury. Każda taka operacja modyfikacji powinna mieć ściśle określone i ciągle monitorowane parametry wejściowe – zgodnie z drugą zasadą.
5) Jeżeli korzystasz z systemu baz danych, który umożliwia wykonywanie w jednym wierszu więcej niż jednego polecenia, koniecznie monitoruj, czy w zapytaniu nie występuje znak średnika, powodujący zakończenie pierwszego i rozpoczęcie drugiej zapytania. Odrzuć wszystkie znaki znajdujące się poza tym średnikiem lub w ogóle zatrzymaj wykonanie zapytania zawierającego średnik czy inne znaki sterujące.
6) Bezwzględnie nie dopuszczaj do wykonywania zapytania, jeśli w jakikolwiek sposób znalazł się w nim znak podwójnego minusa (--) powodujący zignorowanie następującej po nim części zapytania.
7) Przed uruchomieniem aplikacji internetowej na serwerze podłączonym do sieci Internet, należy dokładnie sprawdzić zawartość folderów składowych tej aplikacji, które zamierza się opublikować.
8) Gdzie tylko to możliwe, stosuj haszowanie informacji przechowywanych w bazie danych. Co prawda rodzi to pewne problemy. Np. gdy użytkownik zapomni hasła, a jest ono przechowywane w bazie danych w postaci haszowanej, nie można go odzyskać, jednie można wygenerować dla niego nowe. Ale pomimo tych problemów znacznie utrudnia to intruzowi pozyskanie integralnych informacji o aplikacji internetowej.

Problemy natury moralnej

Ataki typu SQL injection to nie tylko problem dostępu do danego serwisu, czy jego uszkodzenia. Jak już wspomniano – samo pobranie ukrytych informacji dotyczących użytkowników korzystających z serwisu może wyrządzić wiele szkód. Poza kradzieżą tożsamości, nadal jeszcze mało popularną w Polsce, intruz jest w stanie wyciągnąć informacje bardzo praktyczne. Mimo że nie powinno się tak robić, niektóre serwisy w których dokonuje się płatności kartami kredytowymi, przechowują strategiczne informacje o nich (numer, data końca ważności) w niezabezpieczonych bazach danych.

Nie od dziś wiadomo, że serwery rządowe są jednymi z najgorzej zabezpieczonych i najfatalniej oprogramowanych. Programiści otrzymujący lukratywne rządowe zamówienia okazują się być znacznie gorszymi specjalistami niż pracownicy innych firm. Ale tak to już bywa, gdy w państwie dominuje prywata... ■

mysql_pconnect kontra mysql_connect

Łączenie się z serwerem MySQL poprzez wykorzystanie funkcji PHP `mysql_pconnect` jest bardzo złym nawykiem, gdyż obciąża serwer. Ustanowione w ten sposób trwałe połączenie konsumuje znacznie więcej zasobów serwera niż w przypadku połączeń nietrwałych – tworzonych przy pomocy funkcji `mysql_connect`.