SQL Injection

SQL Injection este una dintre cele mai periculoase vulnerabilităţi, cu efecte devastatoare asupra diverselor activităţi (comerciale, industriale, financiare etc.), deoarece conduce la expunerea informaţiilor sensibile stocate în baza de date a unei aplicații. Astfel de informații includ nume de utilizatori, parole, nume, adrese, numere de telefon şi detalii despre carduri bancare.

Această vulnerabilitate apare atunci când este oferită unui atacator posibilitatea de a influenţa cererile *SQL* pe care o aplicaţie le transferă unei baze de date *back-end*.

SQL Injection nu afectează doar aplicaţiile Web; orice cod sursă care acceptă intrări de la o sursă care nu este de încredere şi foloseşte acele intrări pentru a alcătui instrucţiuni *SQL* dinamice poate fi vulnerabil. Un astfel de caz poate apărea și în cazul clienţilor *fat* dintr-o arhitectură *client-server*.

Probabil că *SQL Injection* există de când bazele de date au fost conectate prima dată la aplicaţiile Web. Rain Forest Puppy este cunoscut pentru descoperirea lui, sau cel puţin pentru aducerea unui fapt în atenţia publicului. În ziua de Crăciun a anului 1998, Rain Forest Puppy a scris un articol intitulat „*NT Web Technology Vulnerabilities*” pentru revista on-line Phrack, scrisă de şi pentru hackeri. Rain Forest Puppy a redactat un material asupra *SQL Injection* („How I hacked PacketStorm”) la începutul anului 2000, care a detaliat modul în care era utilizat *SQL Injection* pentru a compromite un site Web popular. De atunci, au fost dezvoltate şi rafinate tehnici pentru exploatarea acestui mecanism.

Vom studia care sunt cauzele apariţiei lui *SQL Injection*. Pentru aceasta, trebuie analizat modul în care sunt structurate aplicaţiile Web. Apoi, vom analiza ce anume cauzează *SQL Injection* la nivel de cod, şi ce practici şi comportamente de dezvoltare ne conduc la acest lucru.

1. Cum funcţionează aplicaţiile Web?

Majoritatea utilizatorilor folosesc aplicaţiile Web zilnic, fie în cadrul profesiei, fie doar pentru a accesa mesageria, a rezerva o călătorie, a cumpăra un produs de la un magazin on-line, a vizualiza ştiri etc.

Indiferent de limbajul în care sunt scrise, aplicaţiile Web au în comun faptul că sunt interactive şi, de cele mai multe ori, lucrează cu baze de date. Acest tip de aplicaţii (*Database-driven Web applications*) sunt foarte frecvente în societatea informaţională actuală. De obicei, acestea constau dintr-o bază de date *back-end* cu pagini Web care conţin script-uri pe partea de server scrise într-un limbaj de programare, script-uri care pot extrage informaţii specifice din baza de date în funcţie de diferitele interacţiuni dinamice cu utilizatorul.

Unele dintre cele mai comune aplicaţii *database-driven* sunt aplicaţiile de *e-commerce,* în care informaţiile sunt stocate în baza de date. O aplicaţie web *database-driven* are, de obicei, 3 niveluri:

- Nivelul de prezentare (un *browser* Web sau un utilitar)
- Nivelul logic (un limbaj de programare)
- Nivelul de stocare (o bază de date)

Browser-ul web (nivelul de prezentare) trimite cereri nivelului intermediar (logic), care le soluţionează cu ajutorul interogărilor şi actualizărilor asupra bazei de date (nivelul de stocare).

De exemplu, un magazin online prezintă un formular de căutare, care permite alegerea şi sortarea produselor de interes, şi furnizează o opţiune pentru rafinarea căutării astfel încât să verifice anumite constrângeri legate de buget. Pentru a vizualiza produsele din magazin care costă mai puţin de 100$, se poate utiliza următorul *URL*:

```
http://www.victim.com/products.php?val=100
```

Următorul script *PHP* ilustrează modul în care intrarea furnizată de utilizator (*val*) este transferată unei instrucţiuni *SQL* dinamice. Următoarea secţiune de cod *PHP* este executată atunci când este utilizat *URL*-ul de mai sus.

```
// conexiune la baza de date
$conn = mysql_connect("localhost","username","password");

// construirea dinamica a comenzii SQL
$query = "SELECT * FROM Products WHERE Price < '$_GET["val"]' " .
              "ORDER BY ProductDescription";
```

```
// executia cererii in baza de date
$result = mysql_query($query);

// iteram prin multimea de inregistrari
while($row = mysql_fetch_array($result, MYSQL_ASSOC))
{
      // afisarea rezultatului in browser
      echo "Description: {$row['ProductDescription']} <br>" .
      "Product ID : {$row['ProductID']} <br>" .
     "Price : {$row['Price']} <br><br>";
}
```

Cererea *SQL* pe care script-ul *PHP* o construieşte şi o execută este următoarea:

```
SELECT *
FROM products
WHERE price < 100
ORDER BY ProductDescription;
```

1.1 O arhitectură simplă

După cum a fost menţionat anterior, o aplicaţie Web *database-driven* are 3 niveluri. În figura următoare este prezentat exemplul simplu, descris anterior, pe 3 niveluri.

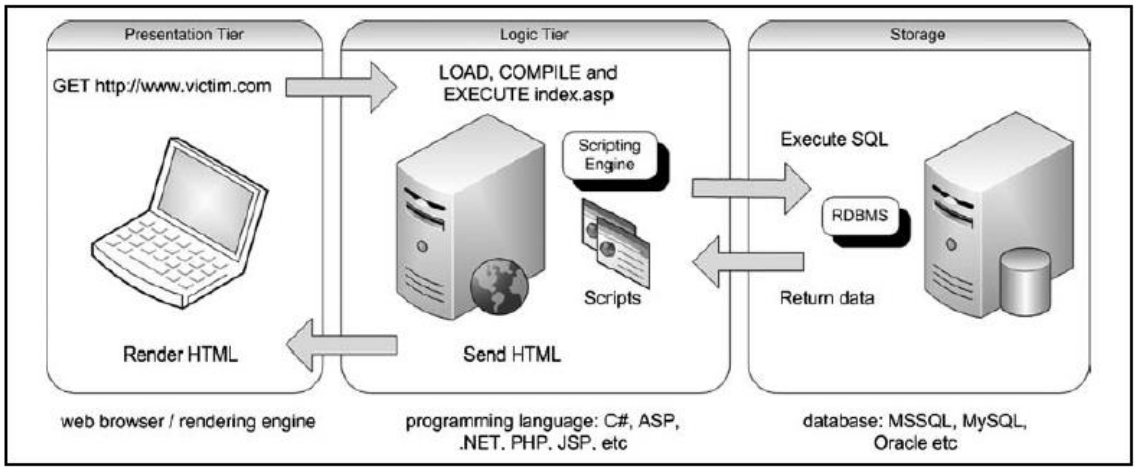

Figura 1. Arhitectura pe 3 niveluri

Nivelul de prezentare afișează informaţiile referitoare la servicii (căutare, cumpărare etc.) şi comunică cu celelalte niveluri furnizând rezultate nivelului client.

Nivelul logic controlează funcţionalitatea aplicaţiei efectuând procesarea corespunzătoare.

Nivelul de date constă din serverele de baze de date, unde informaţia este stocată şi poate fi regăsită. Acest nivel asigură independenţa datelor de serverele de aplicaţii sau de logica aplicaţiei. Deoarece datele au propriul lor nivel, sunt îmbunătăţite scalabilitatea şi performanţa.

În figura 1, *browser*-ul Web trimite cereri nivelului intermediar, care trimite operaţiile corespunzătoare asupra bazei de date. O regulă fundamentală într-o arhitectură 3-tier este că nivelul de prezentare nu comunică niciodată direct cu nivelul de date; într-un astfel de model, comunicarea trebuie să treacă prin nivelul intermediar (*middleware*).

Din punct de vedere conceptual, arhitectura 3-tier este liniară.

1.2 O arhitectură mai complexă

Soluţiile 3-tier nu sunt scalabile, motiv pentru care acest model a fost reevaluat şi a fost creat un concept nou, care să asigure scalabilitate şi mentenanţă: paradigma dezvoltării *n-tier* a aplicaţiilor.

În cadrul acesteia, o soluţie 4-tier implică utilizarea unui *middleware*, numit server de aplicaţii, între serverul Web şi baza de date. Un server de aplicaţii într-o arhitectură *n-tier* este un server care găzduieşte o *API* care expune logica şi procesele aplicaţiei pentru a fi utilizate de către aplicaţie.

Pot fi adăugate servere web suplimentare, pe măsură ce este necesar. Pe lângă acestea, serverul de aplicaţii poate realiza comunicarea cu mai multe surse de date.

Figura 2 prezintă o arhitectură 4-tier simplă.

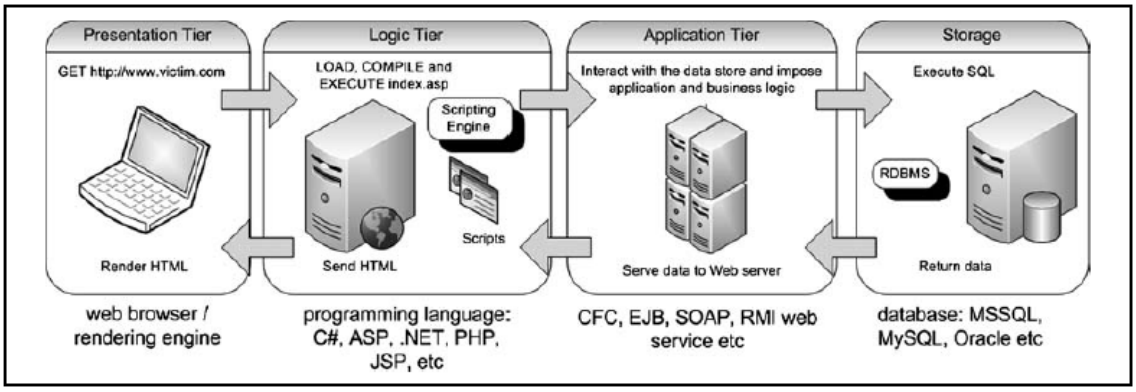

Figura 2. Arhitectura 4-tier

În figura 2, *browser*-ul Web trimite cereri nivelului logic (*middle-tier*),care apelează *API*-urile expuse de către serverul de aplicaţii care rezidă în cadrul nivelului de aplicaţie. Acesta rezolvă solicitările lansând interogări şi actualizări asupra bazei de date (nivelul de stocare).

Conceptul de bază al unei arhitecturi pe niveluri implică „ruperea” unei aplicaţii în părţi logice (niveluri), fiecare dintre acestea având roluri generale sau specifice. Nivelurile pot fi localizate pe maşini diferite sau pe aceeaşi maşină, unde sunt separate virtual sau conceptual unul de celălalt. Cu cât sunt utilizate mai multe niveluri, cu atât este mai specific rolul fiecăruia.

Separarea responsabilităţilor unei aplicaţii în niveluri multiple:

- determină scalabilitatea acesteia
- permite o mai bună separare a *task*-urilor de dezvoltare
- este mai uşor de citit
- componentele sale sunt reutilizabile.

Această abordare poate face ca aplicaţiile să fie mai robuste, prin eliminarea unui punct unic de cădere a lor. De exemplu, decizia de a schimba baza de date nu necesită decât câteva modificări asupra anumitor părţi din nivelul de aplicaţie; nivelurile de prezentare şi logic rămân neschimbate.

Arhitecturile 3-tier şi 4-tier sunt cele mai utilizate pe Internet. Modelul *n-tier* este extrem de flexibil şi permite ca o mulţime de niveluri să fie separate logic şi desfăşurate într-o diversitate de moduri.

2. *SQL Injection*

Aplicaţiile Web devin mai sofisticate, iar complexitatea tehnică a acestora creşte. Disponibilitatea acestor sisteme şi sensibilitatea datelor pe care le stochează şi procesează devin critice pentru majoritatea activităţilor. Aplicaţiile Web, infrastructura şi mediile corespunzătoare utilizează tehnologii diverse şi pot conţine o cantitate semnificativă de cod modificat şi personalizat.

SQL Injection este un atac în care codul *SQL* este inserat sau adăugat în parametrii aplicaţiei sau ai utilizatorului, care sunt ulterior transmişi server-ului *SQL* pentru parsare şi execuţie.

Orice procedură care construieşte instrucţiuni *SQL* poate fi vulnerabilă.

Forma primară de *SQL Injection* constă din inserarea directă de cod în parametrii care sunt concatenaţi cu comenzile *SQL* şi executarea acestor comenzi.

Un atac mai puţin direct injectează cod în şiruri de caractere care sunt destinate stocării într-un tabel sau ca metadate. Când şirurile de caractere stocate sunt concatenate într-o comandă *SQL* dinamică, acest cod este executat. Atunci când o aplicaţie Web eşuează să „cureţe” în mod adecvat parametrii care sunt transmişi instrucţiunilor *SQL* create dinamic, este posibil ca un atacator să modifice construirea instrucţiunilor *SQL* din *back-end*.

Dacă un atacator este capabil să modifice o instrucţiune *SQL*, este posibil ca aceasta să fie executată cu aceleaşi drepturi ca ale utilizatorului aplicaţiei. Atunci când este utilizat serverul *SQL* pentru a executa comenzile care interacţionează cu sistemul de operare, procesul se va executa cu aceleaşi drepturi ca ale componentei care a executat comanda (de exemplu, baza de date, serverul de aplicaţii sau serverul Web) şi care, de obicei, are drepturi puternice.

Pentru a ilustra acest lucru, ne întoarcem la exemplul anterior, al unui magazin online.

Pentru a vizualiza produsele care costă mai puţin de 100$, utilizăm următorul *URL*:

```
http://www.victim.com/products.php?val=100
```

De această dată, însă, dorim să injectăm propriile comenzi *SQL* adăugându-le parametrului de intrare *val*. Acest lucru se poate realiza adăugând şirul de caractere *'OR '1'='1* în *URL*:

```
http://www.victim.com/products.php?val=100’ OR ‘1’=’1
```

Astfel, instrucţiunea *SQL* pe care script-ul *PHP* o construieşte şi o execută va returna toate produsele din baza de date indiferent de preţ. Acest lucru are loc deoarece am modificat logica interogării. Clauza adăugată conduce la o instrucţiune care întoarce întotdeauna *true* (deoarece 1 = 1). Cererea construită şi executată este următoarea:

```
SELECT *
FROM ProductsTbl
WHERE Price < '100.00' OR '1'='1'
ORDER BY ProductDescription;
```

Exemplul precedent arată modul în care un atacator poate prelucra o instrucţiune *SQL* creată dinamic, care este formată cu ajutorul intrărilor din partea utilizatorului ce nu au fost validate sau codificate. Astfel, pot fi realizate acţiuni pe care dezvoltatorul aplicaţiei nu le-a prevăzut sau nu şi le-a dorit.

Ce se întâmplă dacă aplicaţia este administrată de la distanţă utilizând un *CMS* (*Content Management System*)? Un *CMS* este o aplicaţie web care permite crearea, editarea, gestionarea şi publicarea conţinutului pe un site web, fără a fi necesară o bună cunoaştere a vreunui limbaj. Pentru accesarea aplicaţiei *CMS*, poate fi utilizat următorul *URL*:

```
http://www.victim.com/cms/login.php?username=foo&password=bar
```

Aplicaţia *CMS* cere un nume de utilizator şi o parolă pentru a permite accesul la funcţionalităţile sale. Accesarea *URL*-ului precedent va conduce la eroarea „*Incorrect username or password, please try again*”. Codul pentru script-ul login.php este următorul:

```
// conexiune la baza de date
$conn = mysql_connect("localhost","username","password");
// construirea dinamica a instructiunii SQL
$query = "SELECT userid FROM CMSUsers WHERE user = '$_GET["user"]' " .
               "AND password = '$_GET["password"]'";
// executia cererii in baza de date
$result = mysql_query($query);
// aflarea numarului de linii returnate din baza de date
$rowcount = mysql_num_rows($result);
// daca este intoarsa o linie, atunci informatia de conectare este
valida,
// deci utilizatorul este directionat catre paginile de administrare
if ($rowcount != 0)
{
     header("Location: admin.php");
}
// daca nu este returnata nicio linie, atunci informatia de conectare
este invalida
else
{
     die('Incorrect username or password, please try again.')
}
```

Script-ul anterior creează dinamic o instrucţiune *SQL* care va returna o mulţime de înregistrări dacă au fost furnizate un nume de utilizator şi o parolă valide. Instrucţiunea SQL este următoarea:

```
SELECT userid
FROM CMSUsers
WHERE user = 'foo' AND password = 'bar';
```

Problema acestui cod este că dezvoltatorul de aplicaţie consideră că numărul de înregistrări returnate este întotdeauna 0 sau 1. În exemplul anterior am modificat înţelesul cererii astfel încât aceasta să întoarcă mereu *true*. Dacă utilizăm aceeaşi tehnică pentru aplicaţiile *CMS*, putem determina ca logica aplicaţiei să eşueze. Prin adăugarea şirului de caractere ‘OR ‘1=’1 în *URL* se obține:

```
http://www.victim.com/cms/login.php?username=foo&password=bar’ OR
‘1’=’1
```

Instrucţiunea *SQL* construită şi executată de script-ul *PHP* va returna toate valorile coloanei *userid* din tabelul *CMSUsers*. Din nou, a fost modificată logica cererii, deoarece operatorul *OR* va returna întotdeauna *true*.

```
SELECT userid
FROM CMSUsers
WHERE user = 'foo' AND password = 'password' OR '1'='1';
```

Logica aplicaţiei poate considera că dacă baza de date returnează mai mult de 0 înregistrări, atunci au fost introduse informaţiile corecte pentru autentificare, deci poate fi acordat acces la script-ul protejat admin.php.

Observație: Nu testaţi exemplele pe sisteme sau aplicaţii Web, decât dacă aveţi permisiunea din partea proprietarului acestora. Există legi prin care astfel de fapte sunt infracţiuni şi pot fi condamnate ca atare!

Resurse

Există câteva resurse disponibile care arată cât de amplă este problema *SQL Injection*. De exemplu, site-ul web *Common Vulnerabilities and Exposures* (*CVE*) furnizează o listă al cărei scop este de a furniza denumiri comune pentru problemele cunoscute. Scopul *CVE* este de a facilita partajarea datelor despre vulnerabilităţi. Acest site adună informaţii despre vulnerabilităţile cunoscute şi furnizează analize statistice asupra tendinţelor în securitate. În raportul din 2007 (http://cwe.mitre.org/documents/vuln-trends/index.html), *CVE* listează un total de 1754 vulnerabilităţi *SQL Injection* în baza sa de date. *SQL Injection* a cuprins 13.6% dintre toate vulnerabilităţile raportate în 2006.

Open Web Application Security Project (*OWASP*) listează defectele cauzate de injectare (care includ *SQL Injection*) drept a doua cea mai frecventă vulnerabilitate de securitate care a afectat aplicaţiile web în 2007.

3. Cauze ale SQL Injection

SQL este limbajul standard pentru accesarea serverelor de baze de date. Majoritatea aplicaţiilor web interacţionează cu o bază de date, iar majoritatea limbajelor de programare pentru aplicaţiile web (*ASP*, *C#*, *.NET*, *Java*, *PHP*) furnizează modalităţi programatice de conectare la o bază de date şi interacţiune cu aceasta.

Vulnerabilităţile *SQL Injection* apar de cele mai multe ori atunci când dezvoltatorul nu se asigură că valorile primite de la un formular web, cookie, parametru de intrare etc. sunt validate înainte de a fi transferate cererilor *SQL* care vor fi executate pe serverul de baze de date. Dacă un atacator poate să controleze intrarea care este trimisă unei cereri *SQL* şi să o prelucreze astfel încât să fie interpretată drept cod în loc de date, atunci atacatorul poate executa cod pe baza de date *back-end*.

Fiecare limbaj de programare oferă diferite moduri de a construi şi executa instrucţiunile *SQL*, iar dezvoltatorii folosesc o combinaţie a acestor metode pentru a-şi realiza obiectivele. De

cele mai multe ori, tutorialele şi exemplele de cod (care pot fi găsite pe diverse site-uri care ajută dezvoltatorii să rezolve probleme comune de programare) prezintă soluţii nesigure. Fără o înţelegere solidă a bazei de date cu care interacţionează şi o cunoaştere a problemelor potenţiale de securitate a codului aplicaţiei, dezvoltatorii pot produce adesea aplicaţii inerent nesigure care sunt vulnerabile faţă de *SQL Injection*.

3.1 Construirea dinamică a şirurilor de caractere

Construirea dinamică a şirurilor de caractere este o tehnică de programare care permite dezvoltatorilor să construiască instrucţiuni de tip *SQL* dinamic, la *runtime*. Dezvoltatorii pot astfel crea aplicaţii flexibile.

O instrucţiune *SQL* dinamică este construită la momentul execuţiei şi este utilă atunci când trebuie decis la *runtime* ce câmpuri trebuie obţinute, diferitele criterii pentru interogări şi, eventual, tabelele care vor fi interogate.

Pe de altă parte, dezvoltatorii pot obţine acelaşi rezutat într-un mod mult mai sigur, prin utilizarea cererilor parametrizate. Acestea sunt cereri care au unul sau mai mulţi parametri încapsulaţi în instrucţiunea *SQL*. Parametrii pot fi transferaţi acestor cereri la *runtime*; parametrii care conţin intrări utilizator încapsulate nu vor fi interpretate drept cod pentru execuţie, deci nu mai există posibilitatea de injectare a codului. Această metodă de încapsulare a parametrilor în *SQL* este mai eficientă şi mult mai sigură decât construirea dinamică şi execuţia instrucţiunilor *SQL* utilizând tehnici de construire a şirurilor de caractere.

Următorul cod *PHP* arată modul în care anumiţi dezvoltatori construiesc instrucţiuni *SQL* dinamice, ca şiruri de caractere, folosind intrarea primită din partea utilizatorului. Înregistrarea returnată depinde dacă valoarea introdusă de utilizator este prezentă în cel puţin una dintre înregistrările din baza de date.

```
// sir de caractere construit dinamic in PHP
$query = "SELECT * FROM table WHERE field = '$_GET["input"]'";
// sir de caractere construit dinamic in .NET
query = "SELECT * FROM table WHERE field = '" +
            request.getParameter("input") + "'";
```

Atunci când sunt construite dinamic astfel de cereri, în cazul în care codul nu este validat sau codificat înainte de transferarea lui către instrucţiunea creată dinamic, un atacator poate introduce clauze ale instrucţiunii *SQL* care ulterior va fi transferată bazei de date şi executată.

3.2 Caractere *escape* tratate incorect

SQL interpretează caracterul apostrof (‘) drept “graniţă” între cod şi date. Se presupune că orice urmează după apostrofuri este cod care trebuie executat, iar ceea ce este încapsulat între

apostrofuri reprezintă date. Prin urmare, se poate determina rapid dacă un site web este vulnerabil faţă de *SQL Injection* tastând un apostrof în *URL* sau într-un câmp din pagina sau aplicaţia web.

În continuare, este descris codul sursă pentru o aplicaţie simplă care transferă direct intrarea primită din partea utilizatorului către o instrucţiune *SQL* creată dinamic.

```
// construirea instructiunii SQL dinamice
$SQL = "SELECT * FROM table WHERE field = '$_GET["input"]'";
// executia instructiunii SQL
$result = mysql_query($SQL);
// determinarea numarului de linii returnate din baza de date
$rowcount = mysql_num_rows($result);
// iterarea prin multimea de linii returnate
$row = 1;
while ($db_field = mysql_fetch_assoc($result))
 {
      if ($row <= $rowcount)
      {
            print $db_field[$row] . "<BR>";
            $row++;
      }
}
```

Dacă furnizăm un caracter apostrof ca intrare în aplicaţie, putem primi una dintre mai multe erori (în funcţie de numărul de factori de mediu, ca limbaj de programare şi bază de date folosită, tehnologii de protecţie şi apărare implementate):

```
Warning: mysql_fetch_assoc(): supplied argument is not a valid MySQL result resource
```

Următoarea eroare furnizează şi informaţie asupra modului în care a fost formulată instrucţiunea *SQL*:

```
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 "VALUE"'
```

Motivul pentru care a apărut această eroare este că apostroful a fost interpretat ca delimitator de şiruri de caractere. Din punct de vedere sintactic, cererea *SQL* executată la *runtime* este incorectă (are prea mulţi delimitatori de şiruri de caractere) şi, prin urmare, baza de date returnează o eroare.

Apostroful este văzut în *SQL* drept caracter special. Acest caracter este folosit în atacurile *SQL Injection* pentru a fi construite propriile cereri.

Apostroful nu este singurul caracter care serveşte drept cracter *escape*. De exemplu, în *Oracle*, caracterele *blank*, “||”, “,”, “.”, “*/” şi “””” au semnificaţii speciale.

-- Caracterul | poate fi folosit pentru a adauga o functie la o valoare
-- Functia va fi executata, iar rezultatul recuperat si concatenat.
http://www.victim.com/id=1||utl_inaddr.get_host_address(local)--
-- Un asterisc urmat de slash poate semnifica terminarea unui comentariu si/sau
-- directiva a optimizorului in Oracle
http://www.victim.com/hint=*/ from dual—

3.3 Tipuri de date tratate incorect

Mulţi dezvoltatori de aplicaţii cred că, pentru a evita *SQL Injection*, este suficient să trateze sau să valideze intrările, astfel încât să fie eliminat caracterul apostrof. Însă, atunci când sunt tratate date numerice, nu mai este necesar ca acestea să fie încapsulate între apostrofuri.

În continuare, prezentăm codul unei aplicaţii simple care transferă intrarea utilizatorului direct către o instrucţiune *SQL* creată dinamic. Scriptul acceptă un parametru numeric (*$userid*) şi afişează informaţie despre acel utilizator. Cererea presupune că acest parametru este numeric, deci nu apare între apostrofuri.

```
// construirea instructiunii SQL dinamice
$SQL = "SELECT * FROM table WHERE field = $_GET["userid"]"
// executia instructiunii SQL
$result = mysql_query($SQL);
// aflarea numarului de linii returnate din baza de date
$rowcount = mysql_num_rows($result);
// iterarea prin multimea de linii returnate
$row = 1;
while ($db_field = mysql_fetch_assoc($result))
{
      if ($row <= $rowcount)
      {
            print $db_field[$row] . "<BR>";
            $row++;
      }
}
```

În *MySQL* există o funcţie denumită *LOAD_FILE* care citeşte un fişier şi returnează conţinutul acestuia ca string. Fişierul trebuie localizat pe serverul de baze de date şi trebuie furnizată calea absolută a lui, ca parametru al funcţiei. Următoarea linie furnizată ca intrare poate permite unui atacator să citească conţinutul fişierului /etc/passwd, care conţine atributele şi numele utilizatorilor sistemului:

1 UNION ALL SELECT LOAD_FILE('/etc/passwd')—

Această intrare este interpretată direct ca sintaxă *SQL*, deci nu mai este nevoie ca atacatorul să devieze cererea folosind apostrofuri. Instrucţiunea *SQL* obţinută este următoarea:

```
SELECT * FROM TABLE
WHERE
USERID = 1 UNION ALL SELECT LOAD_FILE('/etc/passwd')--
```

3.4 Mulţime de cereri tratată incorect

Unele aplicaţii complexe trebuie să conţină instrucţiuni *SQL* dinamice, deoarece tabelul sau coloanele care trebuie interogate pot să nu fie cunoscute în etapa de dezvoltare a aplicaţiei sau pot să nu existe. Un exemplu este cel al unei aplicaţii care interacţionează cu o bază de date de dimensiuni mari care stochează date în tabele care sunt create periodic. Putem considera o aplicaţie care întoarce date referitoare la condica unui angajat. Datele fiecărui angajat sunt introduse într-un tabel nou, într-un format care conţine data primei zile din lună (*employee_employee-id_01012009*).

Dezvoltatorul web trebuie să permită ca instrucţiunea să fie creată dinamic pe baza datei la care este executată cererea.

În continuare, prezentăm codul sursă al unei aplicaţii simple care transferă intrarea utilizatorului direct către o instrucţiune *SQL* creată dinamic. Scriptul foloseşte valori generate de aplicaţie ca intrare; această intrare constă dintr-un nume de tabel şi 3 nume de coloane. Apoi, afişează informaţii despre un angajat. Aplicaţia permite utilizatorului să selecteze datele care doreşte să fie returnate.

Deoarece aplicaţia a generat deja intrarea, dezvoltatorul are încredere în aceste date; pe de altă parte, acestea sunt controlate de către utilizator, deoarece au fost furnizate printr-o cerere *GET*. Un atacator poate furniza datele referitoare la tabel şi coloane pentru valorile generate de aplicaţie.

```
// construirea instructiunii SQL dinamice
$SQL = "SELECT $_GET["column1"], $_GET["column2"], $_GET["column3"]
             FROM $_GET["table"]";
// executia instructiunii SQL
$result = mysql_query($SQL);
// determinarea numarului de linii returnate din baza de date
$rowcount = mysql_num_rows($result);
// iterarea prin multimea de cereri returnate
$row = 1;
while ($db_field = mysql_fetch_assoc($result))
 {
     if ($row <= $rowcount)
     {
          print $db_field[$row] . "<BR>";
          $row++;
     }}
```

Dacă un atacator ar prelucra cererea *http* şi ar înlocui valoarea *users* pentru numele tabelului şi câmpurile *user*, *password* şi *Super_priv* pentru numele de coloane generate de aplicaţie, ar putea afişa numele şi parolele pentru utilizatorii din sistem. *URL*-ul construit la utilizarea aplicaţiei este următorul:

http://www.victim.com/user_details.php?table=users&column1=user&column2=
password&column3=Super_priv

Dacă injectarea a avut loc cu succes, sunt returnate următoarele date în locul celor care ţin de orele lucrate.

```
+----------------+-------------------------------------------+--------------+
|   user         |   password                                |   Super_priv |
+----------------+-------------------------------------------+--------------+
|   root         |   *2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19 |   Y          |
|   sqlinjection |   *2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19 |   N          |
|   0wned        |   *2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19 |   N          |
+----------------+-------------------------------------------+--------------+
```

3.5 Erori tratate incorect

Tratarea improprie a erorilor poate introduce o mulţime de probleme de securitate pentru un site web. Cea mai comună problemă apare atunci când mesajele de eroare interne, detaliate, sunt afişate utilizatorului sau atacatorului. Aceste mesaje relevă detalii de implementare care pot furniza atacatorului indicii importante asupra defectelor potenţiale din site. Mesajele de eroare detaliate ale bazei de date pot fi folosite pentru a extrage informaţii din baza de date asupra modului în care pot fi îmbunătăţite sau construite injectările care deviază cererea utilizatorului sau o prelucrează cu scopul de a returna date suplimentare sau, în anumite cazuri, de a obţine toate datele din baza de date (*SQL Server*).

Exemplul care urmează a fost scris în *C#* pentru *ASP.NET* şi utilizează un server de baze de date *SQL Server* ca *back end* (această bază de date furnizează cele mai detaliate mesaje de eroare). Scriptul generează dinamic şi execută o instrucţiune *SQL* atunci când utilizatorul aplicaţiei selectează un identificator de utilizator dintr-o listă *drop-down*.

```
private void SelectedIndexChanged(object sender, System.EventArgs e)
{
// Crearea instructiunii SELECT care cauta o inregistrare cu id-ul
specificat
// in proprietatea Value.
string SQL;
SQL = "SELECT * FROM table ";
SQL += "WHERE ID=" + UserList.SelectedItem.Value + "";
// Definim obiectele ADO.NET.
```

```
OleDbConnection con = new OleDbConnection(connectionString);
OleDbCommand cmd = new OleDbCommand(SQL, con);
OleDbDataReader reader;
// Deschiderea bazei de date si citirea informatiei.
try
{
      con.Open();
      reader = cmd.ExecuteReader();
      reader.Read();
      lblResults.Text = "<b>" + reader["LastName"];
      lblResults.Text += ", " + reader["FirstName"] + "</b><br>";
      lblResults.Text += "ID: " + reader["ID"] + "<br>";
      reader.Close();
}
catch (Exception err)
{
      lblResults.Text = "Error getting data. ";
      lblResults.Text += err.Message;
}
finally
{
      con.Close();
}
}
```

Dacă un atacator încearcă să prelucreze cererea *http* înlocuind valoarea *ID* aşteptată pentru a obţine propria lui instrucţiune *SQL*, acesta ar putea utiliza mesajele de eroare cu scopul de a afla informații referitoare la baza de date. De exemplu, dacă atacatorul introduce cererea următoare, execuţia instrucţiunii *SQL* ar conduce la afişarea unui mesaj de eroare informativ care va conţine versiunea de *SGBD* pe care o utilizează aplicaţia web:

```
' and 1 in (SELECT @@version) –
```

Eroarea returnată este următoarea;

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the
nvarchar value 'Microsoft SQL Server 2000 - 8.00.534 (Intel X86) Nov 19 2001
13:23:50 Copyright (c) 1988-2000 Microsoft Corporation Enterprise Edition on
Windows NT 5.0 (Build 2195: Service Pack 3) ' to a column of data type int
```

3.6 Aplicări multiple tratate incorect

Listele albe (*white listing*) constituie o tehnică prin care toate caracterele sunt nepermise, mai puţin cele din lista respectivă. Această abordare pentru validarea intrării constă în crearea unei liste cu toate caracterele posibile care sunt permise pentru intrare şi respingerea oricărui alt caracter.

Se recomandă utilizarea listelor albe, şi nu a celor negre. Tehnica *black listing* presupune că toate caracterele sunt permise, mai puţin cele din lista neagră. Abordarea pentru validarea

intrărilor este de a crea o listă cu toate caracterele posibile şi codificările lor asociate care ar putea fi folosite cu reavoinţă; intrarea lor va fi respinsă. Riscul potenţial asociat cu utilizarea unei liste de caractere inacceptabile este acela că se poate trece cu vederea un caracter inacceptabil la definirea listei sau se pot omite una sau mai multe alternative de reprezentare ale caracterului inacceptabil.

În proiectele web mari poate apărea o problemă: unii dezvoltatori vor urma recomandările şi vor valida intrările, spre deosebire de alţii.

De asemenea, dezvoltatorii de aplicaţii au tendinţa de a proiecta aplicaţia centrat pe utilizator şi încearcă să ghideze utilizatorul prin fluxul aşteptat, considerând că acesta va urma paşii logici care au fost prevăzuţi. De exemplu, se aşteaptă ca, dacă utilizatorul a ajuns la un al treilea formular dintr-o serie de formulare, acesta trebuie să fi completat primul şi al doilea formular. În realitate, este foarte simplu ca fluxul să nu fie respectat, solicitând resurse direct prin *URL*.

Fie următorul fragment de cod al unei aplicații:

```
// procesarea formularului 1
if ($_GET["form"] = "form1")
{
      // parametrul este un string?
      if (is_string($_GET["param"]))
      {
            // obtine lungimea sirului si verifica daca este in
      intervalul stabilit
            if (strlen($_GET["param"]) < $max)
            {
               // transmite sirul unui validator extern
                  $bool = validate(input_string, $_GET["param"]);
                  if ($bool = true) {
                        // continua procesarea
                  }
            }
      }
}

// procesarea formularului 2
if ($_GET["form"] = "form2")
{
      // nu este nevoie sa validam param deoarece l-a validat
formularul 1
      $SQL = "SELECT * FROM TABLE WHERE ID = $_GET["param"]";
      // executa instructiunea sql
      $result = mysql_query($SQL);
      // determina cate linii au fost returnate din baza de date
      $rowcount = mysql_num_rows($result);
      $row = 1;
      // itereaza prin multimea de inregisrari returnate
      while ($db_field = mysql_fetch_assoc($result)) {
            if ($row <= $rowcount){
```

```
            print $db_field[$row] . "<BR>";
            $row++;
            }
      }
}
```

Dezvoltatorul de aplicaţii consideră că nu este nevoie să valideze intrarea în cel de-al doilea formular, având în vedere că primul formular a efectuat această validare. Un atacator poate apela direct al doilea formular, fără a-l utiliza pe primul, sau ar putea să trimită date valide ca intrare în primul formular şi să prelucreze apoi datele aşa cum sunt ele trimise în al doilea formular. Primul *URL* prezentat în continuare va eşua deoarece intrarea este validată; al doilea va conduce la un atac *SQL Injection*, deoarece intrarea nu este validată.

[1] http://www.victim.com/form.php?form=form1&param=' SQL Failed --
[2] http://www.victim.com/form.php?form=form2&param=' SQL Success –

3.7 Configuraţia nesigură a bazei de date

Se poate diminua posibilitatea de acces, cantitatea de date care poate fi accesată neautorizat sau prelucrată, nivelul de acces la sisteme interconectate şi pagubele care pot fi cauzate de către atacurile *SQL Injection* într-o varietate de moduri. Securizarea codului aplicaţiei este punctul de început, însă nu trebuie trecută cu vederea baza de date însăşi.

Bazele de date vin cu un număr de utilizatori impliciţi. *Microsoft SQL Server* utilizează contul de administrator „sa”, *MySQL* foloseşte conturile de utilizator „root” şi „anonymous”, iar în *Oracle* sunt create conturile SYS, SYSTEM, DBSNMP şi OUTLN. Acestea nu sunt singurele conturi create, există mai multe. În multe cazuri, aceste conturi sunt preconfigurate cu parole implicite şi bine cunoscute.

Unii administratori de sistem şi baze de date instalează serverele de baze de date pentru a fi executate ca root, SYSTEM sau Administrator (cu privilegii de administrator). Serviciile de pe server, în special serverele de baze de date, ar trebui executate dintr-un cont mai puţin privilegiat pentru a reduce pagubele potenţiale asupra sistemului de operare şi altor procese în eventualitatea unui atac cu succes asupra bazei de date. Acest lucru nu a fost, însă, posibil pentru versiunile mai vechi de Oracle sub Windows, deoarece acesta trebuia executat cu privilegii SYSTEM.