# Inspectarea codului sursă pentru prevenirea *SQL Injection*

Adesea, modul cel mai rapid de a descoperi zone vulnerabile în faţa *SQL Injection* într-o aplicaţie este inspectarea codului sursă al acesteia. Aceasta poate constitui singura opţiune dacă nu este permisă utilizarea unor instrumente pentru testarea *SQL Injection* în cadrul procesului de dezvoltare.

Anumite forme de construire a şirurilor de caractere şi de execuţie sunt vizibile chiar la o primă vedere a codului. Nu este, însă, clar dacă datele utilizate în aceste cereri provin din *browser*-ul utilizatorului sau dacă au fost validate corect sau codificate înainte de a fi trimise înapoi utilizatorului.

În continuare, vom prezenta câteva modalităţi pentru descoperirea *SQL Injection* în codul sursă, de la identificarea locului în care pot intra în aplicaţie date controlate de utilizator până la identificarea tipurilor de construcţii de cod sursă care pot conduce la *SQL Injection*.

Există 2 metode principale pentru analizarea codului sursă, în vederea descoperirii vulnerabilităţilor: analiza statică şi cea dinamică a codului. Analiza statică este procesul prin care codul este analizat fără a fi executat. Analiza dinamică este efectuată la *runtime*.

Analiza statică manuală implică inspectarea codului linie cu linie, pentru identificarea potenţialelor vulnerabilităţi. De multe ori însă, acest lucru nu se poate realiza în aplicaţiile mari. O astfel de activitate poate consuma foarte mult timp şi poate fi extrem de laborioasă. În întâmpinarea acestei situaţii, consultanţii de securitate şi dezvoltatorii scriu utilitare şi script-uri sau utilizează diferite instrumente pentru a sprijini activitatea de inspectare a unor cantităţi mari de cod sursă.

Este foarte important să fie adoptată o abordare metodică atunci când este inspectat codul sursă. Scopul este acela de a localiza şi analiza zone din cod care pot avea implicaţii asupra securităţii aplicaţiei. Abordarea prezentată în continuare are drept scop detectarea vulnerabilităţilor de tip infectare (*taint style*). Datele infectate sunt cele care au fost primite de la o sursă care nu este de încredere (variabilele interne pot deveni infectate dacă anumite date infectate sunt copiate în acestea). Aceste date pot fi dezinfectate dacă sunt utilizate rutine de

curăţare sau funcţii de validare a intrărilor. Datele infectate pot cauza probleme de securitate în punctele vulnerabile din program (denumite *sink*).

În contextul inspectării codului în vederea descoperirii vulnerabilităţilor *SQL Injection*, un *sink* este o funcţie sensibilă din punct de vedere al securităţii, care este utilizată pentru a executa instrucţiuni *SQL* în baza de date. Pentru a restrânge aria de inspectare, ar trebui să pornim prin identificarea potenţialelor *sink*-uri. Această activitate nu este simplă, deoarece fiecare limbaj de programare oferă un număr diferit de moduri de construcţie şi execuţie a instrucţiunilor *SQL*. Odată ce un *sink* a fost identificat, poate deveni foarte evident că există o vulnerabilitate *SQL Injection*. Totuşi, de cele mai multe ori va trebui efectuată o analiză suplimentară a codului pentru a determina dacă această vulnerabilitate există. Vulnerabilităţile *SQL Injection* apar de cele mai multe ori atunci când dezvoltatorul aplicaţiei Web nu se asigură că valorile primite de la o sursă *sink* (o metodă din care îşi au originea datele infectate, cum ar fi un formular Web, cookie, parametru de intrare etc.) sunt validate înainte de a fi transferate cererilor *SQL* care vor fi executate pe *server*-ul de baze de date.

Următoarea linie de cod *PHP* ilustrează acest lucru:

```
$result = mysql_query("SELECT * FROM table
                       WHERE column ='$_GET["param"]'");
```

Acest cod este vulnerabil în faţa *SQL Injection* deoarece intrarea utilizatorului este transferată direct unei instrucţiuni *SQL* construite dinamic şi este executată fără ca datele să fie mai întâi validate.

În cele mai multe cazuri, identificarea unei funcţii care este utilizată pentru crearea şi executarea instrucţiunilor *SQL* nu va constitui finalul procesului, deoarece nu va fi posibilă simpla identificare a prezenţei unei vulnerabilităţi din linia de cod. De exemplu, următoarea linie de cod PHP este potenţial vulnerabilă, însă nu putem fi siguri deoarece nu ştim dacă variabila *$param* este infectată sau dacă este validată înainte de a fi transferată funcţiei:

```
$result = mysql_query("SELECT * FROM table
                       WHERE column = '$param'");
```

Pentru decide dacă există o vulnerabilitate, este necesar să urmărim variabila până la originea ei şi să o analizăm în fluxul aplicaţiei. Pentru aceasta, este necesar să identificăm punctele de intrare în aplicaţie (sursa *sink*-urilor) şi să identificăm în codul sursă în ce punct variabila *$param* primeşte o valoare. Încercăm să identificăm o linie similară următoarei:

```
$param = $_GET["param"];
```

Linia precedentă atribuie date controlate de utilizator variabilei *$param*.

Odată ce a fost identificat un punct de intrare, este important să urmărim intrarea pentru a descoperi unde şi cum sunt utilizate datele, iar acest lucru poate fi realizat prin intermediul fluxului de execuţie. Dacă sunt găsite următoarele 2 linii de cod *PHP*, se poate deduce sigur că aplicaţia a fost vulnerabilă la *SQL Injection*, referitor la parametrul *$param* controlat de utilizator:

```
$param = $_GET["param"];
$result = mysql_query("SELECT * FROM table
                       WHERE field = '$param'");
```

Codul precedent este vulnerabil la *SQL Injection* deoarece o variabilă infectată (*$param*) este transferată direct unei instrucţiuni *SQL* construite dinamic (*sink*) şi executate. Dacă sunt găsite următoarele 3 linii de cod *PHP*, se poate deduce sigur că aplicaţia a fost vulnerabilă la *SQL Injection*, însă a fost impusă o limită asupra lungimii şirului de caractere de la intrare. Aceasta înseamnă că este posibil sau nu ca problema să fi putut fi exploatată. În acest caz, trebuie urmărită variabila *$limit* pentru a vedea exact cât spaţiu este disponibil pentru injectare:

```
$param = $_GET["param"];
if (strlen($param) < $limit){
     error_handler("param exceeds max length!")
}
$result = mysql_query("SELECT * FROM table
                       WHERE field = '$param'");
```

Dacă sunt găsite următoarele 2 linii de cod *PHP*, se poate deduce că dezvoltatorul a încercat să prevină *SQL Injection*:

```
$param = mysql_real_escape_string($param);
$result = mysql_query("SELECT * FROM table
                       WHERE field = '$param'");
```

Utilizarea unor filtre (de exemplu, *mysql_real_escape_string*) nu poate preveni complet exploatarea unei vulnerabilităţi *SQL Injection*. Anumite tehnici utilizate în conjuncţie cu condiţiile de mediu vor permite unui atacator să exploateze respectiva vulnerabilitate. Din acest motiv, se poate deduce că aplicaţia poate fi vulnerabilă la *SQL Injection* din cauza parametrului *$param*.

După cum se poate observa din exemplele anterioare simplificate, procesul de inspectare a codului în vederea descoperirii vulnerabilităţilor *SQL Injection* necesită un efort semnificativ. Este important să fie mapate toate dependenţele şi trasate toate fluxurile de date astfel încât să poată fi identificate intrările infectate şi neinfectate, dar să poată fi arătată şi posibilitatea ca o vulnerabilitate să fie exploatabilă. Urmând o abordare metodică, putem asigura că inspectarea identifică şi arată prezenţa (sau absenţa) tuturor vulnerabilităţilor potenţiale de *SQL Injection*.

Inspectarea trebuie pornită prin identificarea funcţiilor care sunt utilizate pentru a construi şi executa instrucţiuni *SQL* (*sink*-uri) cu intrări controlate de către utilizator care sunt potenţial infectate; apoi, trebuie identificate punctele de intrare pentru datele controlate de către utilizator care sunt transferate acestor funcţii (sursele *sink*), iar, în final, trebuie urmărite datele controlate de către utilizator prin fluxul de execuţie al aplicaţiei pentru a stabili dacă acestea sunt infectate atunci când ajung la *sink*. Apoi, se poate decide dacă există o vulnerabilitate şi dacă poate fi exploatată.

Pentru a simplifica activitatea de inspectare manuală a codului, pot fi construite *script*-uri sau programe complexe în orice limbaj pentru a capta şi lega împreună diferitele *pattern*-uri din codul sursă. Următoarele secţiuni vor prezenta exemple de cod care trebuie căutat în *PHP* şi *Java*. Aceste principii şi tehnici pot fi aplicate şi altor limbaje şi vor fi folositoare în identificarea altor fluxuri de cod.

# 1. Modalităţi periculoase de scriere a codului sursă

Pentru a efectua o inspectare eficientă a codului şi a identifica toate vulnerabilităţile potenţiale de *SQL Injection*, trebuie să recunoaştem modalităţile periculoase de scriere a codului, precum cele care încorporează tehnici de construire dinamică a şirurilor de caractere. Câteva dintre aceste tehnici au fost introduse în cursul anterior.

Pentru început, următoarele linii construiesc şiruri de caractere care sunt concatenate cu intrări infectate (date care nu au fost validate):

```
// a dynamically built sql string statement in PHP
$sql = "SELECT * FROM table WHERE field = '$_GET["input"]'";
// a dynamically built sql string statement in C#
String sql = "SELECT * FROM table WHERE field = '" +
request.getParameter("input") + "'";
// a dynamically built sql string statement in Java
String sql = "SELECT * FROM table WHERE field = '" +
request.getParameter("input") + "'";
```

Sursele *PHP* şi *Java* prezentate în continuare arată modul în care unii dezvoltatori construiesc dinamic şi execută instrucţiuni *SQL* care conţin date controlate de către utilizator ce nu au fost validate. Este important să putem identifica această modalitate de scriere a codului atunci când acesta este inspectat cu scopul identificării de vulnerabilităţi.

```
// a dynamically executed sql statement in PHP
mysql_query("SELECT * FROM table
             WHERE field =  '$_GET["input"]'");
// a dynamically executed sql string statement in C#
```

```
SqlCommand command = new SqlCommand("SELECT * FROM table WHERE field =
'" + request.getParameter("input") + "'", connection);
// a dynamically executed sql string statement in Java
ResultSet rs = s.executeQuery("SELECT * FROM table WHERE field = '" +
request.getParameter("input") + "'");
```

Unii dezvoltatori cred că, dacă nu construiesc şi execută instrucţiuni *SQL* dinamice, ci doar transferă date ca parametri actuali ai procedurilor stocate, codul nu va mai fi vulnerabil. Acest lucru nu este adevărat, deoarece şi procedurile stocate pot fi vulnerabile la *SQL Injection*. O procedură stocată este un set de instrucţiuni, având un nume atribuit, care este stocat în baza de date. Un exemplu de procedură stocată *Oracle* vulnerabilă este următorul:

```
-- vulnerable stored procedure in Oracle
CREATE OR REPLACE PROCEDURE SP_ StoredProcedure (input IN VARCHAR2) AS
sql VARCHAR2;
BEGIN
sql := 'SELECT field FROM table WHERE field = ''' || input || '''';
EXECUTE IMMEDIATE sql;
END;
```

În exemplul precedent, variabila *input* este luată direct de la intrarea utilizatorului şi concatenată cu şirul de caractere *sql*. Şirul de caractere *SQL* este transferat instrucţiunii *EXECUTE* ca parametru şi executat. Această procedură este vulnerabilă la *SQL Injection* chiar dacă intrarea utilizatorului este transferată ca parametru.

Dezvoltatorii folosesc metode uşor diferite pentru a interacţiona cu procedurile stocate. Următoarele linii de cod sunt prezentate ca exemple asupra modului în care unii dezvoltatori execută procedurile stocate în cadrul codului:

```
// a dynamically executed sql stored procedure in PHP
$result = mysql_query("select SP_StoredProcedure($_GET['input'])");
```

```
// a dynamically executed sql stored procedure in Java
CallableStatement cs = con.prepareCall("{call SP_ StoredProcedure
request.getParameter("input")}");
string output = cs.executeUpdate();
```

Liniile de cod precedente se execută şi transferă date infectate, controlate de utilizator, ca parametri ai procedurilor stocate. Dacă procedurile stocate sunt construite incorect, într-un mod similar exemplelor prezentate anterior, poate exista o vulnerabilitate *SQL Injection* care poate fi exploatată. Prin urmare, atunci când este inspectat codul sursă, nu este important doar să identificăm vulnerabilităţile din codul sursă al aplicaţiei, ci trebuie inspectat şi codul sursă al procedurilor stocate. Codul sursă dat ca exemplu este suficient pentru a arăta modul în care dezvoltatorii produc cod vulnerabil. Însă, fiecare limbaj oferă un număr de modalităţi pentru a construi şi executa instrucţiuni SQL, cu care trebuie să fim familiari.

Pentru a afirma că o vulnerabilitate există în codul de bază, este necesară identificarea punctelor de intrare pentru a ne asigura că intrarea controlată de utilizator poate fi introdusă în instrucţiunile *SQL*. Pentru aceasta, trebuie să fim familiari cu modul în care intrarea controlată de către utilizator intră în aplicaţie. Din nou, fiecare limbaj de programare oferă un număr de moduri diferite pentru a obţine intrarea utilizatorului. Metoda cea mai comună de a cere intrări din partea utilizatorului este prin folosirea formularelor *HTML*. Următorul cod *HTML* ilustrează modul în care este creat un formular *Web*:

```
<form name="simple_form" method="get" action="process_input.php">
<input type="text" name="foo">
<input type="text" name="bar">
<input type="submit" value="submit">
</form>
```

## 2. Funcţii periculoase

În continuare vom prezenta o listă detaliată a metodelor de construire şi execuţie a comenzilor *SQL*, pentru diferite limbaje.

Limbajul *PHP* oferă suport pentru diferite tipuri de baze de date, printre care *Oracle*, *MySQL* şi *SQL Server*. Funcţiile relevante, referitoare la baza de date *Oracle*, sunt următoarele:

- *oci_parse*() – parsează o instrucţiune înainte de a fi executată (înainte de *oci_execute*());
- *ora_parse*() – parsează o instrucţiune înainte de a fi executată (înainte de *ora_exec*());
- *odbc_prepare*() – pregăteşte o instrucţiune pentru execuţie (înainte de *odbc_execute*());
- *odbc_execute*() – execută o instrucţiune *SQL*;
- *odbc_exec*() – pregăteşte şi execută o instrucţiune *SQL*.

Următoarele linii de cod demonstrează modul în care aceste funcţii pot fi utilizate într-o aplicaţie *PHP*:

```
// oci_parse() - parses a statement before it is executed
$stmt = oci_parse($connection, $sql);
oci_execute($stmt);
// ora_parse() - parses a statement before it is executed
if (!ora_parse($cursor, $sql)){exit;}
else {ora_exec($cursor);}
// odbc_prepare() - prepares a statement for execution
$stmt = odbc_prepare($db, $sql);
$result = odbc_execute($stmt);
// odbc_exec() - prepare and execute a SQL statement
```

```
$result = odbc_exec($db, $sql);
```

În *Java*, lucrurile sunt puţin diferite. *Java* oferă pachetul *java.sql* şi *API*-ul *JDBC* (*Java Database Connectivity*) pentru conexiunea la baza de date. Funcţiile relevante pentru bazele de date cele mai cunoscute sunt următoarele:

- *createStatement*() – creează un obiect instrucţiune pentru trimiterea instrucţiunilor *SQL* la baza de date;
- *prepareStatement*() – creează o instrucţiune *SQL* precompilată şi o stochează într-un obiect;
- *executeQuery*() – execută cererea *SQL* dată;
- *executeUpdate*() – execută instrucţiunea *SQL* dată;
- *execute*() – execută instrucţiunea dată;
- *addBatch*() – adaugă instrucţiunea *SQL* dată la lista curentă de comenzi;
- *executeBatch*() – trimite un *batch* de comenzi în baza de date, pentru execuţie.

Următoarele linii de cod arată modul în care pot fi utilizate aceste funcţii într-o aplicaţie *Java*:

```
// createStatement() – is used to create a statement object that
// is used for sending sql statements to the specified database
statement = connection.createStatement();
// PreparedStatement – creates a precompiled SQL statement and
// stores it in an object.
PreparedStatement sql = con.prepareStatement(sql);
// executeQuery() – sql query to retrieve values from the
// specified table.
result = statement.executeQuery(sql);
// executeUpdate () – Executes an SQL statement, which may be an
// INSERT, UPDATE, or DELETE statement or a statement that
// returns nothing
result = statement.executeUpdate(sql);
// execute() – sql query to retrieve values from the specified
// table.
result = statement.execute(sql);
// addBatch() – adds the given SQL command to the current list of
// commands
statement.addBatch(sql);
statement.addBatch(more_sql);
```

# 3. Urmărirea datelor

Până acum, am înţeles modul în care aplicaţiile Web obţin datele de intrare de la utilizator, am analizat metodele pe care dezvoltatorii le folosesc pentru a procesa datele în limbajul ales şi am observat că practicile inadecvate de programare pot conduce la prezenţa

vulnerabilităţii *SQL Injection*. În continuare, dorim să identificăm o vulnerabilitate *SQL Injection* şi să urmărim datele controlate de către utilizator prin aplicaţie. Abordarea noastră porneşte cu identificarea utilizării funcţiilor periculoase (*sinks*).

Putem realiza o inspectare manuală a codului sursă, linie cu linie, utilizând un editor de text sau un *IDE*. Pentru a economisi timp şi a identifica rapid codul care ar trebui ulterior inspectat în detaliu, metoda cea mai simplă şi directă constă în a folosi utilitarul *grep* din *UNIX* (disponibil şi pentru *Windows*). Va fi necesar să alcătuim o listă de şiruri de căutare încercate şi testate pentru a identifica liniile de cod care ar putea fi vulnerabile la *SQL Injection*, după cum fiecare limbaj de programare oferă un număr de modalităţi diferite de a primi şi procesa intrările, dar şi o mulţime de metode pentru a construi şi executa comenzi SQL.

Utilitarul *grep* realizează căutarea de text în linie de comandă. Un alt utilitar care este foarte folositor este *awk*, general proiectat pentru procesarea datelor bazate pe text, fie în fişiere fie în *stream*-uri de date.

În *PHP*, înainte de a inspecta codul este important să verificăm starea variabilelor *register_globals* şi *magic_quotes*. Aceste setări se configurează în fişierul *php.ini*. Setarea *register_globals* înregistrează variabilele EGPCS (*Environment, Get, Post, Cookie, Server*) ca variabile globale. Acest lucru conduce adeseori la diferite vulnerabilităţi. Opţiunea *magic_quotes* este în prezent depreciată şi urmează să fie eliminată din limbaj. Aceasta este o caracteristică de securitate implementată de către *PHP* pentru a ignora caracterele potenţial dăunătoare transmise aplicaţiei (apostrof, ghilimele, \ etc.).

După ce am stabilit starea acestor opţiuni poate fi inspectat codul.

Se poate utiliza următoarea comandă pentru a căuta recursiv prin directorul cu fişierele sursă utilizările lui *oci_parse*() şi *ora_parse*(), cu intrari ale utilizatorului direct în comanda *SQL*. Aceste funcţii se folosesc înainte de *oci_exec*(), *ora_exec*() şi *oci_execute*() pentru a compila o instrucţiune *SQL*.

```
$ grep -r -n "\(oci\|ora\)_parse\(.*\$_\(GET\|\POST\).*\)" src/ | awk
-F : '{print "filename: "$1"\nline: "$2"\nmatch: "$3"\n\n"}'
```

```
filename: src/oci_parse.vuln.php
line: 4
match: $stid = oci_parse($conn, "SELECT * FROM TABLE WHERE COLUMN =
'$_GET['var']'");
filename: src/ora_parse.vuln.php
line: 13
match:   ora_parse($curs,"SELECT   *   FROM   TABLE   WHERE   COLUMN   =
'$_GET['var']'");
```

Se poate utiliza următoarea comandă pentru a căuta recursiv utilizarea funcţiilor *odbc_prepare*() şi *odbc_exec*() cu intrări directe din partea utilizatorului într-o instrucţiune *SQL*. Funcţia *odbc_prepare*() este utilizată înainte de *odbc_execute*() pentru a compila o instrucţiune *SQL*.

```
$ grep -r -n "\(odbc_prepare\|odbc_exec\)\(.*\$_\(GET\|\POST\).*\)"
src/ |
awk -F : '{print "filename: "$1"\nline: "$2"\nmatch: "$3"\n\n"}'
```

```
filename: src/odbc_exec.vuln.php
line: 3
match: $result = odbc_exec ($con, "SELECT * FROM TABLE WHERE COLUMN =
'$_GET['var']'");
filename: src/odbc_prepare.vuln.php
line: 3
match: $result = odbc_prepare ($con, "SELECT * FROM TABLE WHERE COLUMN
= '$_GET['var']'");
```

Pentru a găsi fişierele sursă *Java* care utilizează comenzile *prepareStatement*(), *executeQuery*(), *executeUpdate*(), *execute*(), *addBatch*() şi *executeBatch*(), este utilă comanda următoare:

```
$ grep -r -n
"preparedStatement(\|executeQuery(\|executeUpdate(\|execute(\|addBatch
(\|executeBatch(" src/ | awk -F : '{print "filename: "$1"\nline:
"$2"\nmatch:
"$3"\n\n"}'
```

# 4. Inspectarea codului *PL/SQL*

Codul *Oracle PL/SQL* este foarte diferit şi, în majoritatea cazurilor, mai nesigur decât codul convenţional de programare din *PHP*, *.NET*, *Java* etc. De exemplu, sistemul *Oracle* a suferit de-a lungul timpului multiple vulnerabilităţi de injectare *PL/SQL* în cadrul pachetelor predefinite ale bazei de date. Codul *PL/SQL* se execută implicit cu privilegiile celui care l-a definit, astfel că a devenit o ţintă pentru atacatorii care caută un mod de a-şi creşte privilegiile.

O procedură stocată poate fi executată fie cu drepturile apelantului (*authid current_user*) sau cu drepturile proprietarului (*authid definer*). Acest comportament poate fi specificat cu ajutorul clauzei *authid* atunci când este creată procedura.

De obicei, codul *PL/SQL* nu este disponibil în fişiere text. Pentru a analiza sursa unei proceduri *PL/SQL* avem 2 opţiuni:

1) exportul codului sursă din baza de date (utilizând pachetul *dbms_metadata*). Se poate folosi următorul script *SQL*Plus* pentru a exporta instrucţiunile *LDD* din baza de date *Oracle*:

```
set echo off feed off pages 0 trims on term on trim on linesize 255
long 500000
head off
--
execute DBMS_METADATA.SET_TRANSFORM_PARAM(DBMS_METADATA.SESSION_
TRANSFORM,'STORAGE',false);
spool getallunwrapped.sql
--
select 'spool ddl_source_unwrapped.txt' from dual;
--
-- create a SQL scripts containing all unwrapped objects
select 'select
dbms_metadata.get_ddl('''||object_type||''','''||object_name||''','''|
|owner||''') from dual;'
from (select * from all_objects where object_id not in (select o.obj#
from source$ s, obj$ o,user$ u where ((lower(s.source) like
'%function%wrapped%') or (lower (s.source) like '%procedure%wrapped%')
or (lower(s.source) like '%package%wrapped%'))
and o.obj#=s.obj# and u.user#=o.owner#))
where object_type in ('FUNCTION', 'PROCEDURE', 'PACKAGE', 'TRIGGER')
and owner in
('SYS')
order by owner,object_type,object_name;
--
-- spool a spool off into the spool file.
select 'spool off' from dual;
spool off
--
-- generate the DDL_source
--
@getallunwrapped.sql
Quit
```

2) construirea propriilor instrucţiuni *SQL* pentru a căuta în baza de date cod *PL/SQL* de interes. Sistemul *Oracle* stochează codul sursă *PL/SQL* în vizualizările *ALL_SOURCE* şi *DBA_SOURCE* (coloana *TEXT*). De interes imediat ar putea fi orice cod care utilizează comanda *execute immediate* sau unităţile pachetului *dbms_sql*. *PL/SQL* este case-insensitive, deci codul căutat poate fi găsit doar dacă este utilizată o funcţie pentru caractere (de exemplu, *lower*). Dacă aceste funcţii primesc intrări nevalidate, poate fi posibil să injectăm instrucţiuni *SQL* arbitrare.

Următoarea instrucţiune *SQL* ne permite să obţinem codul sursă *PL/SQL*:

```
SELECT owner AS Owner, name AS Name, type AS Type, text AS Source FROM
dba_source
WHERE ((LOWER(Source) LIKE '%immediate%')
OR (LOWER(Source) LIKE '%dbms_sql')) AND owner='PLSQL';
```

Să presupunem că în coloana *source* obţinem:

```
execute immediate(param);
```

```
execute immediate('select count(*) from '||param) into i;
execute immediate('select count(*) from all_users
                   where user_id='||param) into i;
```

Astfel, avem 3 candidaţi pentru o inspecţie mai atentă. Cele 3 instrucţiuni sunt vulnerabile deoarece datele controlate de utilizator sunt transferate funcţiilor periculoase fără a fi validate. Similar dezvoltatorilor de aplicaţii, administratorii de baze de date (*DBA*) pot mai întâi copia parametrii în variabile locale. Pentru a căuta blocurile *PL/SQL* care copiază valorile parametrilor în şiruri de caractere *SQL* create dinamic, se poate folosi următoarea instrucţiune *SQL*:

```
SELECT owner AS Owner, name AS Name, type AS Type, text AS Source FROM
dba_source where lower(Source) like '%:=%||%''%';
```

Presupunem că răspunsul este:

```
Owner         Name                 Type          Source
------------------------------------------------------------------------------------------
-------------------------------------------------------------
SYSMAN SP_ StoredProcedure  Procedure     sql := 'SELECT field FROM table WHERE
                                                  field = ''' || input || '''';
```

Instrucţiunea anterioară a găsit un pachet care creează dinamic o instrucţiune *SQL* din intrări controlate de către utilizator. Pentru a inspecta lucrurile mai atent, este utilă următoarea instrucţiune *SQL* care obţine sursa pachetului:

```
SELECT text AS Source FROM dba_source WHERE name='SP_STORED_PROCEDURE'
AND
owner='SYSMAN' order by line;
```

```
Source
------------------------------------------------------------------------
---
1 CREATE OR REPLACE PROCEDURE SP_ StoredProcedure (input IN VARCHAR2)
AS
2 sql VARCHAR2;
3 BEGIN
4 sql := 'SELECT field FROM table WHERE field = ''' || input || '''';
5 EXECUTE IMMEDIATE sql;
6 END;
```

Procedura stocată anterioară este vulnerabilă la *SQL Injection* chiar dacă intrarea utilizatorului este transferată printr-un parametru.

Următorul script *PL/SQL* poate fi utilizat pentru a căuta codul *PL/SQL* din baza de date care este potenţial vulnerabil la *SQL Injection*. Rezultatul va trebui privit mai atent, însă va permite restrângerea căutărilor ulterioare.

```
select distinct a.owner,a.name,b.authid,a.text SQLTEXT
```

```
from all_source a,all_procedures b
where (
lower(text) like '%execute%immediate%(%||%)%'
or lower(text) like '%dbms_sql%'
or lower(text) like '%grant%to%'
or lower(text) like '%alter%user%identified%by%'
or lower(text) like '%execute%immediate%''%||%'
or lower(text) like '%dbms_utility.exec_ddl_statement%'
or lower(text) like '%dbms_ddl.create_wrapped%'
or lower(text) like '%dbms_hs_passthrough.execute_immediate%'
or lower(text) like '%dbms_hs_passthrough.parse%'
or lower(text) like '%owa_util.bind_variables%'
or lower(text) like '%owa_util.listprint%'
or lower(text) like '%owa_util.tableprint%'
or lower(text) like '%dbms_sys_sql.%'
or lower(text) like '%ltadm.execsql%'
or lower(text) like '%dbms_prvtaqim.execute_stmt%'
or lower(text) like '%dbms_streams_rpc.execute_stmt%'
or lower(text) like '%dbms_aqadm_sys.execute_stmt%'
or lower(text) like '%dbms_streams_adm_utl.execute_sql_string%'
or lower(text) like '%initjvmaux.exec%'
or lower(text) like '%dbms_repcat_sql_utl.do_sql%'
or lower(text) like '%dbms_aqadm_syscalls.kwqa3_gl_executestmt%'
)
and lower(a.text) not like '% wrapped%'
and a.owner=b.owner
and a.name=b.object_name
and a.owner not in
('OLAPSYS','ORACLE_OCM','CTXSYS','OUTLN','SYSTEM','EXFSYS',
'MDSYS','SYS','SYSMAN','WKSYS','XDB')
order by 1,2,3
```

# 5. Inspectarea automată a codului sursă

Inspectarea manuală este un proces lung şi laborios, care necesită să fim familiari cu aplicaţia. Acest proces poate fi, însă, automatizat. Utilitarele care realizează acest lucru produc uneori *false positives* sau *false negatives*.

Unele utilitare folosesc doar expresii regulate pentru identificarea funcţiilor *security-sensitive*. Există utilitare care pot identifica aceste funcţii care transferă direct date infectate ca parametri ai altor funcţii. De asemenea, există utilitare care combină aceste funcţionalităţi cu capacitatea de a identifica sursele *sink*-urilor (punctele din aplicaţie în care îşi au originea datele infectate).