# Prevenirea *SQL Injection* în *Oracle*

Vom prezenta câteva reguli ce asigură prevenirea *SQL Injection* în *Oracle*. Ca orice set de astfel de reguli, acestea nu descriu singurul mod de realizare a acestui obiectiv. Regulile prezentate sunt recomandate deoarece sunt simplu de formulat, de urmărit şi de auditat atunci când se face o analiză manuală a codului şi, cel mai important, deoarece garantează obiectivul ca sursele *PL/SQL* din baza de date să fie rezistente în faţa *SQL Injection*.

## 1. Reducerea „suprafeţei de atac”

### 1.1 Expunerea bazei de date doar prin intermediul unei *API PL/SQL*

Această regulă se referă la stabilirea unui utilizator al bazei de date ca fiind singurul prin intermediul căruia se poate conecta un client. Acest utilizator poate deţine doar sinonime private, care pot desemna doar unităţi de cod *PL/SQL* (subprograme). Aplicând această regulă, unităţile *PL/SQL* sunt deţinute de către utilizatori prin care clientul nu se poate conecta. Privilegiul *Execute* doar asupra acestor unităţi *PL/SQL* este acordat utilizatorului la care se poate conecta clientul.

Această abordare defineşte formal *API*-ul. Unităţile de cod *PL/SQL* din *API* pot avea drepturile celui care le-a definit sau ale celui care le invocă, conform scopului lor.

Această paradigmă situează responsabilitatea de a preveni *SQL Injection* acolo unde îi este locul: în subsistemul reprezentat de stiva aplicaţiei care execută codul *SQL* şi oferă singura abordare a cărei siguranţă poate fi demonstrată. Paradigma este o extensie naturală a ideii conform căreia tot codul sursă care este responsabil cu aplicarea integrităţii datelor este plasat în baza de date.

*Baza de date trebuie expusă clienţilor doar prin intermediul unei API PL/SQL. Privilegiile trebuie controlate cu atenţie astfel încât clientul să nu aibă acces la obiectele aplicaţiei de alte tipuri (în special tabele şi vizualizări).*

## 1.2 Utilizarea drepturilor apelantului

Subprogramele stocate şi metodele *SQL* se execută cu o mulţime de privilegii. Implicit, aceste privilegii sunt cele ale proprietarului schemei (creatorul obiectelor). Drepturile creatorului nu implică doar privilegiile, ci sunt utilizate şi pentru rezolvarea referinţelor obiect.

Dacă o unitate de program nu necesită să fie executată cu privilegiile creatorului, trebuie specificat că acea unitate se execută cu privilegiile apelantului. Drepturile apelantului pot diminua riscul de *SQL Injection*. În orice caz, utilizarea acestor drepturi nu garantează eliminarea completă a riscului de *SQL Injection*.

Subprogramele stocate care vor fi executate cu drepturile apelantului trebuie să includă clauza *AUTHID CURRENT_USER*.

***Exemplu:***

```
Connect / as sysdba;
-- Procedura care utilizeaza drepturile creatorului
CREATE OR REPLACE
PROCEDURE change_password(p_username VARCHAR2 DEFAULT NULL,
                          p_new_password VARCHAR2 DEFAULT NULL)
IS
  v_sql_stmt VARCHAR2(500);
BEGIN
    v_sql_stmt := 'ALTER USER '||p_username ||' IDENTIFIED BY '
                  || p_new_password;

    EXECUTE IMMEDIATE v_sql_stmt;
END change_password;
/

Connect hr; --pentru testarea procedurii anterioare
Exec sys.change_password('sys', 'oracle');

-- Procedura apartine lui SYS si, implicit,
-- se executa cu drepturile acestuia.

Connect / as sysdba;
-- Procedura care utilizeaza drepturile apelantului
CREATE OR REPLACE
PROCEDURE change_password(p_username VARCHAR2 DEFAULT NULL,
                          p_new_password VARCHAR2 DEFAULT NULL)
AUTHID CURRENT_USER
IS
  v_sql_stmt VARCHAR2(500);
BEGIN
    v_sql_stmt := 'ALTER USER '||p_username ||' IDENTIFIED BY '
                  || p_new_password;

    EXECUTE IMMEDIATE v_sql_stmt;
END change_password;
/

Connect hr;
```

```
Exec sys.change_password('sys', 'oracle'); -- nu mai functioneaza
                                        -- insufficient privileges
Exec sys.change_password('hr', 'oracle'); -- OK

-- SQL Injection: incercare de a atribui schemei HR un spaţiu mai
-- mare în tablespace
Exec sys.change_password('hr', 'oracle quota unlimited on
users');
-- insufficient privileges
```

Drepturile creatorului se folosesc atunci când dorim să acordăm utilizatorilor acces nerestricţionat la unul sau mai multe tabele sau prin intermediul unui subprogram.

Drepturile apelantului se utilizează atunci când scopul subprogramului este acela de a efectua o operaţie parametrizată, care foloseşte privilegiile utilizatorului care o invocă. Folosirea acestor drepturi ajută la limitarea lor şi, prin urmare, minimizează expunerea.

## 1.3 Reducerea intrărilor arbitrare

Deoarece un atac *SQL Injection* este posibil doar dacă sunt acceptate intrări din partea utilizatorului, rezultă că atacurile pot fi prevenite prin limitarea acestor intrări.

Mai întâi, trebuie reduse interfeţele cu utilizatorul final doar la cele cu adevărat necesare. De exemplu, într-o *API PL/SQL*, se expun doar rutinele destinate clientului. Se vor şterge toate interfeţele legate de *debug*, testare, cele depreciate sau nefolositoare. Acestea nu adaugă nimic la funcţionalitate, însă oferă unui atacator mai multe mijloace de a ajunge în aplicaţie.

Acolo unde este necesară o intrare din partea utilizatorului trebuie folosite caracteristicile limbajului pentru a ne asigura că sunt specificate date doar de tipul necesar. De exemplu, nu trebuie specificat un parametru *VARCHAR2* atunci când parametrul este utilizat ca număr; nu trebuie utilizat tipul *number* atunci când sunt necesare doar numere întregi pozitive, ci se va folosi tipul *natural*.

## 1.4 Îmbunătăţirea securităţii bazei de date

Baza de date *Oracle* conţine caracteristici de securitate care ajută la protejarea ei în faţa mai multor tipuri de atacuri, inclusiv *SQL Injection*.

Câteva practici care trebuie urmărite pentru securizarea bazei de date *Oracle* sunt următoarele:

- Criptarea datelor sensibile astfel încât acestea să nu poată fi vizualizate;
- Evaluarea tuturor privilegiilor *PUBLIC* şi revocarea lor acolo unde este cazul;
- Evitarea acordării privilegiului *EXECUTE ANY PROCEDURE*;
- Evitarea acordării privilegiilor *WITH ADMIN option*;

- Asigurarea că utilizatorii aplicaţiei primesc implicit numărul minim de privilegii;
- Nu se acordă acces la pachetele *Oracle* standard care pot opera asupra sistemului de operare (*UTL_HTTP*, *UTL_SMTP*, *UTL_TCP*, *DBMS_PIPE*, *UTL_MAIL* şi *UTL_FTP*);
- Blocarea conturilor implicite ale bazei de date şi expirarea parolelor implicite;
- Ştergerea *script*-urilor exemplu şi a programelor din directorul *Oracle*;
- Execuţia *listener*-ului *Oracle* ca utilizator neprivilegiat;
- Activarea managementului parolelor. Se vor aplica reguli referitoare la lungimea parolelor, istoric, complexitate, obligaţia ca utilizatorii să îşi schimbe parolele în mod regulat.

# 2. Utilizarea instrucţiunilor SQL fixate la momentul compilării

Dacă textul complet al instrucţiunii *SQL*, şi nu doar un *template* al acesteia, este fixat la momentul compilării, nu va mai fi necesar niciun efort pentru a arăta că site-ul care lansează acea comandă este sigur faţă de *SQL Injection*.

Termenul de instrucţiune *SQL* fixată a momentul compilării implică o instrucţiune *SQL* care nu se poate modifica la *runtime* şi care poate fi determinată prin citirea codului sursă (expresie *PL/SQL varchar2* statică – nu poate fi modificată la *runtime* şi poate fi precalculată la momentul compilării).

Codul *SQL* încapsulat (*embedded SQL*) garantează instrucţiuni *SQL* fixate la momentul compilării, însă nu reprezintă singurul mijloc pentru a realiza acest lucru. Următoarea secvenţă de cod prezintă o alternativă :

```
declare
      Stmt constant varchar2(80) :=
            'alter session
            set NLS_Date_Format = ''AD yyyy-mm-dd hh24:mi:ss''';
begin
execute immediate Stmt;
...
end;
```

În exemplul anterior, am presupus că specificaţiile funcţionale ale aplicaţiei prevăd un raport care listează valori *datetime*, într-un mod particular.

Utilizarea cuvântului cheie *constant* cu o instrucţiune de atribuire care foloseşte doar expresii statice *varchar2 PL/SQL* fixează textul instrucţiunii *SQL* la momentul compilării. Auditorul nu trebuie să studieze vreo extindere a codului care are loc între declaraţia lui *Stmt* şi utilizarea lui ca argument al lui *execute immediate*. Compilatorul *PL/SQL* va refuza să compileze unitatea respectivă dacă aceasta include cod care modifică *Stmt*.

Prin urmare, este important să separăm discuţia despre metoda de execuţie a codului *SQL* la un site particular de cea despre *template*-ul sintaxei *SQL* care va fi executată la acel site. *SQL* încapsulat suportă doar următoarele tipuri de instrucţiuni: *select, insert, update, delete, merge, lock table, commit, rollback, savepoint* şi *set transaction*.

Pentru toate celelalte tipuri de instrucţiuni, trebuie utilizată una dintre metodele *PL/SQL* pentru *SQL* dinamic. Mai mult, pentru acestea, instrucţiunea *execute immediate* este suficientă în majoritatea cazurilor.

Există cazuri care necesită instrucţiuni *SQL* de tipuri pe care *SQL* încapsulat nu le suportă, însă acestea apar foarte rar în codul aplicaţiilor obişnuite.

*Atunci când specificaţiile de proiectare pentru codul unei aplicaţii obişnuite propun utilizarea altor lucruri decât SQL încapsulat, trebuie realizată o analiză raţională a acestuia. Proiectarea adecvată poate aduce siguranţă aplicaţiei.*

Reciproca este adevărată : atunci când o instrucţiune de unul dintre tipurile suportate de *SQL* încapsulat urmează să fie executată, şi când textul instrucţiunii poate fi fixat la momentul compilării, nu mai este necesar SQL dinamic. Prin urmare, SQL încapsulat ar trebui folosit întotdeauna în astfel de cazuri.

Următorul exemplu nu respectă această regulă:

```
...
q constant varchar2(1) := '''';
SQL_VC2_Literal constant varchar2(32767) :=
q||Raw_User_Input||q;
begin
Stmt :=
'select c2 from t where c1 = '||SQL_VC2_Literal;
execute immediate Stmt bulk collect into v;
...
```

*Dacă este posibil, se recomandă să se utilizeze instrucţiuni SQL care sunt fixate la momentul compilării. Să se utilizeze SQL încapsulat atunci când tipul instrucţiunii este suportat. Altfel, să se utilizeze instrucţiunea execute immediate cu un argument PL/ SQL constant, obţinut doar din expresii statice varchar2.*

## 3. Utilizarea instrucţiunilor *SQL* statice

Dacă nu este necesar *SQL* dinamic, utilizaţi *SQL* static; acesta are următoarele avantaje:

- *SQL* static reduce vulnerabilitatea la *SQL Injection*
- Compilarea cu succes creează dependenţe între obiectele schemei

- Poate îmbunătăţi performanţele, comparativ cu *DBMS_SQL*.

Există două situaţii comune referitoare la *SQL* dinamic, în care dezvoltatorii utilizează adesea *SQL* static atunci când serveşte scopului propus, fiind mai sigur.

- tratarea numărului variabil de valori din listele *IN*
- tratarea operatorului *LIKE*.

***Exemplu:***

```
CONN hr

SET SERVEROUTPUT ON

-- Cerere care contine o clauza IN
SELECT d.department_name, l.city, c.country_name
   FROM departments d, locations l, countries c
   WHERE d.location_id = l.location_id
   AND c.country_id = l.country_id
   AND c.country_name in ('Germany','Canada')
/

SELECT d.department_name, l.city, c.country_name
   FROM departments d, locations l, countries c
   WHERE d.location_id = l.location_id
   AND c.country_id = l.country_id
   AND c.country_name in ('Germany','Canada','United Kingdom')
/
```

Dorim să permitem utilizatorului să poată introduce oricâte valori în clauza *IN*. Deoarece numărul de valori din clauza *IN* poate varia la *runtime*, rezultă că putem rezolva problema cu ajutorul unei instrucţiuni *SQL* dinamice.

```
CREATE OR REPLACE PROCEDURE show_dept_loc_concat
(p_countries IN VARCHAR2)
AS
  TYPE cv_infotyp IS REF CURSOR;
  cv   cv_infotyp;
  v_sql_stmt VARCHAR2(4000);
  v_dept departments.department_name%TYPE;
  v_city locations.city%TYPE;
  v_country countries.country_name%TYPE;
BEGIN

  v_sql_stmt := 'SELECT d.department_name, l.city, c.country_name
   FROM departments d, locations l, countries c
   WHERE d.location_id = l.location_id
   AND c.country_id = l.country_id
   AND c.country_name in (' || p_countries ||')';

  OPEN cv FOR v_sql_stmt;
  LOOP
      FETCH cv INTO v_dept, v_city, v_country;
      EXIT WHEN cv%NOTFOUND;
```

```
      DBMS_OUTPUT.PUT_LINE('Dept Info: '||v_dept ||
                          ', ' || v_city || ', ' || v_country);
  END LOOP;
  CLOSE cv;
EXCEPTION WHEN OTHERS THEN
   dbms_output.PUT_LINE(sqlerrm);
   dbms_output.PUT_LINE('SQL statement: ' || v_sql_stmt);
END;
/

EXEC show_dept_loc_concat('''Canada'',''Germany''')
```

La execuţia procedurii, cele două valori au fost transmise în cadrul unui singur şir de caractere, însă lista de valori din clauza *IN* va fi construită cu două valori separate în cerere.

Apelul anterior funcţionează şi returnează rezultatul aşteptat. Însă există riscul apariţiei *SQL Injection,* deoarece folosim *SQL* dinamic cu valori de intrare concatenate. Pentru a elimina acest risc, rescriem procedura astfel încât aceasta să utilizeze un argument de legătură.

```
CREATE OR REPLACE PROCEDURE show_dept_loc_dynamicsql
(p_countries IN VARCHAR2)
AS
  TYPE cv_infotyp IS REF CURSOR;
  cv   cv_infotyp;
  v_sql_stmt VARCHAR2(500);
  v_dept departments.department_name%TYPE;
  v_city locations.city%TYPE;
  v_country countries.country_name%TYPE;
BEGIN

  v_sql_stmt := 'SELECT d.department_name, l.city, c.country_name
   FROM departments d, locations l, countries c
   WHERE d.location_id = l.location_id
   AND c.country_id = l.country_id
   AND c.country_name in (:p_countries)';

  OPEN cv FOR v_sql_stmt USING p_countries;
  LOOP
      FETCH cv INTO v_dept, v_city, v_country;
      EXIT WHEN cv%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Dept Info: '||v_dept ||
                          ', ' || v_city || ', ' || v_country);
  END LOOP;
  CLOSE cv;
EXCEPTION WHEN OTHERS THEN
   dbms_output.PUT_LINE(sqlerrm);
   dbms_output.PUT_LINE('SQL statement: ' || v_sql_stmt);
END;
/

EXEC show_dept_loc_dynamicsql('''Canada'',''Germany''')
```

Cele două valori sunt transmise ca un singur şir de caractere în cadrul procedurii. Lista de valori din clauza *IN* va fi construită conţinând o singură valoare în cerere.

Clauza *IN* va deveni:

```
IN(''Canada', 'Germany'');
```

Apelul anterior nu returnează nicio linie.

Am putea rescrie codul utilizând mai multe argumente de legătură (p_country1, p_country2 etc.) astfel încât fiecare valoare să fie interpretată separat. Această soluţie nu este acceptabilă dacă există prea multe valori posibile.

Putem utiliza o colecţie pentru a stoca valorile din lista *IN*, iar apoi să interogăm din nou aceste valori într-o instrucţiune *SQL*.

```
CREATE OR REPLACE TYPE str2tblType
AS TABLE OF VARCHAR2(4000)
/
```

Definim o funcţie care acceptă ca argument de intrare o listă delimitată de valori şi le parsează într-o colecţie. Apoi, creem o procedură care utilizează funcţia în lista de valori a clauzei *IN* a unei instrucţiuni *SQL* statice.

```
CREATE OR REPLACE FUNCTION str2list
    ( p_str IN VARCHAR2,
      p_delim IN VARCHAR2 DEFAULT ',' )
RETURN str2tblType
AS
l_str LONG DEFAULT p_str || p_delim;
l_n   NUMBER;
l_data str2tblType := str2tblType();
BEGIN
  LOOP
    l_n := INSTR( l_str, p_delim );
    EXIT WHEN (NVL(l_n,0) = 0);
    l_data.EXTEND;
    l_data(l_data.COUNT) := LTRIM(RTRIM(SUBSTR(l_str,1,l_n-1)));
    l_str := SUBSTR( l_str, l_n+1 );
  END LOOP;
  RETURN l_data;
END;
/

CREATE OR REPLACE PROCEDURE show_dept_loc_staticsql
(p_countries IN VARCHAR2)
AS
BEGIN
FOR i IN
  (SELECT d.department_name, l.city, c.country_name
   FROM departments d, locations l, countries c
   WHERE d.location_id = l.location_id
   AND c.country_id = l.country_id
   AND c.country_name in
     (SELECT column_value FROM TABLE(str2list(p_countries))))
  LOOP
```

```
      DBMS_OUTPUT.PUT_LINE('Dept Info: '||i.department_name ||
                          ', ' || i.city || ', ' || i.country_name);
  END LOOP;
END;
/

EXEC show_dept_loc_staticsql('Canada, Germany')
```

***Exemplu:*** (tratarea comparaţiei utilizând operatorul *LIKE*)

```
CONN oe

SET SERVEROUTPUT ON

CREATE OR REPLACE
PROCEDURE list_products_dynamic (p_product_name VARCHAR2 DEFAULT NULL)
AS

TYPE cv_prodtyp IS REF CURSOR;
cv   cv_prodtyp;
v_prodname products.product_name%TYPE;
v_minprice products.min_price%TYPE;
v_listprice products.list_price%TYPE;
v_stmt  VARCHAR2(400);

BEGIN

v_stmt := 'SELECT product_name, min_price, list_price FROM products
           WHERE product_name like ''%'||p_product_name||'%''';

  OPEN cv FOR v_stmt;
dbms_output.put_line(v_stmt);
  LOOP
      FETCH cv INTO v_prodname, v_minprice, v_listprice;
      EXIT WHEN cv%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Product Info: '||v_prodname ||', '||
                           v_minprice ||', '|| v_listprice);
  END LOOP;
  CLOSE cv;
END;
/

EXEC list_products_dynamic('Laptop')
```

Utilizarea concatenării pentru construirea unei instrucţiuni *SQL* dinamice expune aplicaţia la atacuri *SQL Injection*. Următoarea cerere obţine o listă a utilizatorilor bazei de date, cu ajutorul operatorului pe mulţimi *UNION*, concatenat în cererea *SQL* dinamică.

```
EXEC list_products_dynamic(''' and 1=0 union select cast(username as
nvarchar2(100)), null, null from all_users --')
```

Pentru a utiliza *SQL* static, acceptăm intrarea utilizatorului şi concatenăm şirul de caractere necesar ('%') la o variabilă locală, care este transmisă apoi unei instrucţiuni *SQL* statice.

```
CREATE OR REPLACE
PROCEDURE list_products_static (p_product_name VARCHAR2 DEFAULT NULL)
AS

v_bind  VARCHAR2(400);

BEGIN
  v_bind := '%'||p_product_name||'%';

  FOR i in
  (SELECT product_name, min_price, list_price FROM products
            WHERE product_name like v_bind)

  LOOP
      DBMS_OUTPUT.PUT_LINE('Product Info: '||i.product_name ||', '||
                           i.min_price ||', '|| i.list_price);
  END LOOP;
END;
/
```

Procedura funcţionează pentru o valoare de intrare corectă, iar încercarea unui atac *SQL Injection* eşuează.

```
EXEC list_products_static('Laptop')

EXEC list_products_static(''' and 1=0 union select cast(username as
nvarchar2(100)), null, null from all_users --')
```

**Ce se întâmplă dacă trebuie utilizat *SQL* dinamic?**

*SQL* dinamic este inevitabil în următoarele situaţii:

- nu cunoaştem textul complet al instrucţiunilor *SQL* care trebuie executate într-o procedură *PL/SQL*. De exemplu, o instrucţiune *SELECT* care include un identificator (un nume de tabel care este necunoscut la momentul compilării, o clauză *WHERE* al cărei număr de condiţii este necunoscut la momentul compilării etc.).
- Dorim să executăm instrucţiuni *LDD* şi alte instrucţiuni *SQL* care nu sunt suportate în programele *SQL* statice.
- Dorim să scriem un program care poate trata modificările din definirea datelor fără a fi necesară recompilarea.

Dacă trebuie utilizat *SQL* dinamic, se recomandă ca instrucţiunile să nu fie construite utilizând concatenarea valorilor de intrare, ci utilizând argumente de legătură (*bind*). Dacă nu poate fi evitată utilizarea valorilor de intrare, trebuie validate aceste intrări; de asemenea, trebuie luată în considerare constrângerea intrărilor utilizatorului la o listă predefinită de valori, preferabil numerice.

# 4. Utilizarea argumentelor de legătură

## 4.1 Utilizarea argumentelor de legătură cu *SQL* dinamic

Pot fi utilizate argumente de legătură în clauzele *WHERE, VALUES* şi *SET* ale instrucţiunilor *SQL,* atât timp cât aceste argumente nu sunt utilizate ca identificatori *Oracle* (de exemplu, nume de coloane şi nume de tabele) sau cuvinte cheie.

De exemplu, următoarea instrucţiune *SQL* dinamică cu valori concatenate pentru şirul de caractere:

```
v_stmt :=
 'SELECT '||filter(p_column_list)||' FROM employees '||
 'WHERE department_name = '''|| p_department_name ||'''';

EXECUTE IMMEDIATE v_stmt;
```

poate fi rescrisă sub forma următoarei instrucţiuni *SQL* dinamice cu un *placeholder* (:1) utilizând un argument de legătură (*p_department_name*);

```
v_stmt :=
 'SELECT '||filter(p_column_list)||' FROM employees '||
 'WHERE department_name = :1';

EXECUTE IMMEDIATE v_stmt USING p_department_name;
```

Anterior, am analizat exemple referitoare la valorile din lista *IN* şi la operatorul *LIKE*; în aceste situaţii, nu este necesar să utilizăm *SQL* dinamic.

## 4.2 Utilizarea argumentelor de legătură cu *PL/SQL* dinamic

Similar lui *SQL* dinamic, trebuie evitată scrierea de cod *PL/SQL* dinamic cu ajutorul concatenării şirurilor de caractere. Impactul lui *SQL Injection* în *PL/SQL* dinamic este chiar mai serios decât în *SQL* dinamic deoarece, în acest caz, pot fi scrise şi injectate mai multe instrucţiuni.

Dacă trebuie folosit *PL/SQL* dinamic, se recomandă utilizarea argumentelor de legătură. De exemplu, următorul cod *PL/SQL*:

```
v_stmt :=
  'BEGIN
     get_phone (''' || p_fname ||
                ''','''|| p_lname ||'''); END;';

EXECUTE IMMEDIATE v_stmt;
```

poate fi rescris ca *PL/SQL* dinamic cu varabile *placeholder* (:1, :2) utilizând argumente de legătură (*p_fname*, *p_lname*):

```
v_stmt :=
  'BEGIN
     get_phone(:1, :2); END;';

EXECUTE IMMEDIATE v_stmt USING p_fname, p_lname;
```

***Exemplu:*** (*PL/SQL Injection*)

```
Conn hr

SET SERVEROUTPUT ON

--
-- Cod vulnerabil la SQL injection
--
CREATE OR REPLACE FUNCTION get_avg_salary (p_job VARCHAR2)
RETURN NUMBER
AS
avgsal employees.salary%TYPE;
v_blk VARCHAR2(4000);

BEGIN
-- bloc PL/SQL utilizat pentru calculul salariului mediu
v_blk := 'BEGIN SELECT AVG(salary) INTO :avgsal
           FROM hr.employees
           WHERE job_id = '''||P_JOB||'''; END;';

EXECUTE IMMEDIATE v_blk
USING OUT avgsal;

dbms_output.put_line('Code: ' || v_blk);

RETURN avgsal;
END;
/
```

Deşi este utilizat un argument de legătură *OUT*, argumentul de intrare *p_job* este concatenat în şirul de caractere dinamic. Acest lucru conduce la o vulnerabilitate *SQL Injection*.

```
exec dbms_output.put_line('Average salary is '
||get_avg_salary('SH_CLERK'))


SELECT salary FROM employees WHERE email = 'PFAY'
/
```

Încercăm un atac *SQL Injection*: modificarea salariului unui angajat.

```
exec dbms_output.put_line('Average salary is '
||get_avg_salary('SH_CLERK''; UPDATE hr.employees SET salary=4500
WHERE email=''PFAY''; END;--'))


SELECT salary FROM employees WHERE email = 'PFAY'
/
```

Salariul angajatului respectiv a fost modificat.

```
ROLLBACK
/
```

Vom utiliza un argument de legătură *IN* (*p_job*) cu codul *PL/SQL* dinamic.

```
--
-- Codul sigur la SQL injection
--

CREATE OR REPLACE FUNCTION get_avg_salary (p_job VARCHAR2)
RETURN NUMBER
AS
avgsal employees.salary%TYPE;
v_blk VARCHAR2(4000);

BEGIN
v_blk := 'BEGIN SELECT AVG(salary) INTO :avgsal
           FROM hr.employees
           WHERE job_id = :p_job; END;';

EXECUTE IMMEDIATE v_blk
USING OUT avgsal, IN p_job;

dbms_output.put_line('Code: ' || v_blk);

RETURN avgsal;
END;
/
```

Codul funcţionează corect pentru o intrare validă.

```
exec dbms_output.put_line('Average salary is '
||get_avg_salary('SH_CLERK'))


SELECT salary FROM employees WHERE email = 'PFAY'
/

exec dbms_output.put_line('Average salary is '
||get_avg_salary('SH_CLERK''; UPDATE hr.employees SET salary=4500
WHERE email=''PFAY''; END;--'))
```

Apelul anterior al funcţiei va returna *NULL*, deoarece nu există nicio valoare a coloanei *JOB_ID* care să corespundă şirului de caractere injectat. Totodată, valoarea salariului nu a fost modificată.

```
SELECT salary FROM employees WHERE email = 'PFAY'
/
```

**Ce se întâmplă dacă nu putem utiliza argumente de legătură?**

Există cazuri când astfel de argumente nu pot fi utilizate:

- instrucţiunile LDD
- identificatorii *Oracle*.

În acest caz, toate intrările concatenate instrucţiunii dinamice trebuie filtrate şi „curăţate” (utilizând pachetul *DBMS_ASSERT*).

# 5. Filtrarea intrărilor cu *DBMS_ASSERT*

Pentru a proteja de atacurile *SQL Injection* aplicaţiile care nu utilizează argumente de legătură împreună cu *SQL* dinamic, şirurile de caractere concatenate trebuie filtrate şi „curăţate”.

Un caz de utilizare primar pentru *SQL* dinamic cu şiruri de caractere concatenate corespunde situaţiei în care un identificator *Oracle* (de exemplu, numele unui tabel) este necunoscut la momentul compilării codului.

Pachetul *DBMS_ASSERT* conţine un număr de funcţii care pot fi utilizate pentru a filtra şirurile de caractere de intrare, în particular a celor care vor fi folosite ca identificatori *Oracle*.

Dintre funcţiile acestui pachet, amintim şi exemplificăm următoarele:

- *ENQUOTE_LITERAL* – acceptă un parametru de intrare *VARCHAR2*; dacă intrarea este un literal *SQL* bine format, funcţia încadrează şirul de caractere între apostrofuri şi îl returnează. Această funcţie verifică dacă toate celelalte apostrofuri sunt prezente în perechi. Dacă sunt găsite apostrofuri individuale, apare eroarea *ORA-06502: PL/SQL: numeric or value error exception is raised*.
- *SIMPLE_SQL_NAME* – verifică dacă şirul de caractere este un nume *SQL* simplu, adică:

- primul caracter este alfabetic
- conţine doar caractere alfanumerice sau caracterele _, $ şi #
- numele între apostrofuri trebuie încadrate între două apostrofuri şi pot conţine orice caracter, inclusiv apostroful (reprezentat prin două apostrofuri)
- funcţia ignoră *blank*-urile de la începutul şi de la sfârşitul şirului de caractere şi nu validează lungimea şirului de caractere de intrare.

Dacă şirul de caractere nu este conform acestor reguli, va apărea eroarea *ORA-44003: invalid SQL name*.

***Exemplu***:

```
CONNECT sys/oracle AS SYSDBA
set echo on
DROP USER testuser CASCADE
/
```

```
DROP USER eviluser CASCADE
/

--Crearea utilizatorului care va fi victima SQL Injection
GRANT
  Unlimited Tablespace,
  Create Session,
  Create Table,
  Create Procedure
TO testuser identified by testuser
/

--Crearea utilizatorului răuvoitor
GRANT
  Unlimited Tablespace,
  Create Session,
  Create Table,
  Create Procedure
TO eviluser identified by eviluser
/

CONNECT testuser/testuser
SET SERVEROUTPUT ON

-- Crearea tabelului care va fi atacat de eviluser.
CREATE TABLE t(a varchar2(10))
/

BEGIN
      INSERT INTO t (a) values ('a');
      INSERT INTO t (a) values ('b');
      INSERT INTO t (a) values ('b');
      INSERT INTO t (a) values ('c''d');
      commit;
END;
/

CREATE OR REPLACE PROCEDURE Count_Rows(w in varchar2)
authid definer as

      -- Useful constant

      Quote constant varchar2(1) := '''';

      -- The statement we will execute

      Stmt  constant varchar2(32767) :=
            'SELECT count(*) FROM t WHERE a='||
                  Quote||w||Quote;

      -- The count of rows returned by the statement

      Row_Count  number;
BEGIN
      EXECUTE IMMEDIATE Stmt INTO Row_Count;
      DBMS_OUTPUT.PUT_LINE(Row_Count||' rows');
END;
```

```
/

BEGIN Count_Rows('a'); END;
/

BEGIN Count_Rows('b'); END;
/

BEGIN Count_Rows('c''''d'); END;
/

BEGIN Count_Rows('x'); END;
/

GRANT EXECUTE ON Count_Rows TO PUBLIC
/

connect eviluser/eviluser

SET SERVEROUTPUT ON

-- Crearea unei functii care şterge toate liniile tabelului t.
CREATE OR REPLACE FUNCTION f RETURN varchar2 authid current_user as
      pragma autonomous_transaction;
BEGIN
      -- Execute immediate because t is not granted to public
      EXECUTE IMMEDIATE 'DELETE FROM t';
      COMMIT;
      RETURN 'a';
END;
/

GRANT EXECUTE ON f TO PUBLIC
/

BEGIN testuser.Count_Rows('a'' and eviluser.f=''a'); END;
/
```

Şirul de caractere dinamic din funcţia *Count_Rows* a devenit:

```
SELECT count(*)
FROM t
WHERE a = '''a'' and eviluser.f=''a''';

connect testuser/testuser
SET SERVEROUTPUT ON
SELECT * FROM t
/
-- Nu mai exista linii în tabel.

BEGIN
      INSERT INTO t (a) values ('a');
      INSERT INTO t (a) values ('b');
      INSERT INTO t (a) values ('b');
      INSERT INTO t (a) values ('c''d');
      commit;
END;
```

```
/


CREATE OR REPLACE PROCEDURE Count_Rows(w in varchar2)
authid definer as

      -- Useful constants

      Quote      constant varchar2(1) := '''';
      Quote_Quote      constant varchar2(2) := Quote||Quote;

      Safe_Literal    varchar2(32767) :=
            Quote||replace(w,Quote,Quote_Quote)||Quote;

      -- The statement we will execute

      Stmt  constant varchar2(32767) :=
            'SELECT count(*) FROM t WHERE a='||
                  DBMS_ASSERT.ENQUOTE_LITERAL(Safe_Literal);

      -- The count of rows returned by the statement

      Row_Count  number;
BEGIN
      EXECUTE IMMEDIATE Stmt INTO Row_Count;
      DBMS_OUTPUT.PUT_LINE(Row_Count||' rows');
END;
/
```

*DBMS_ASSERT.ENQUOTE_LITERAL* consideră că fiecare prim apostrof este caracter *escape* şi îl elimină, deci fiecare apostrof din şirul de caractere trebuie înlocuit prin două apostrofuri.

Pentru parametri corecţi, procedura funcţionează corect, iar atacul SQL Injection nu va afecta tabelul.

```
BEGIN Count_Rows('a'); END;
/

BEGIN Count_Rows('b'); END;
/

BEGIN Count_Rows('c''''d'); END;
/

BEGIN Count_Rows('c''d'); END;
/

BEGIN Count_Rows('x'); END;
/

-- Repetam atacul.

connect eviluser/eviluser
SET SERVEROUTPUT ON
```

```
BEGIN testuser.Count_Rows('a'' and eviluser.f=''a'); END;
/

BEGIN testuser.Count_Rows('b'); END;
/

connect testuser/testuser

set serveroutput on

SELECT * FROM t
/
```

***Exemplu:***

```
CONN hr/hr

SET SERVEROUTPUT ON
```

Scopul procedurii următoare este de a extrage o anumită coloană, dintr-un tabel particular. Numele tabelului şi al coloanei sunt furnizate ca parametri la *runtime*. Instrucţiunea *SQL* dinamică, formată cu valorile de intrare concatenate, constituie o vulnerabilitate.

```
CREATE OR REPLACE
PROCEDURE show_col (p_colname varchar2, p_tablename   varchar2)
AS
type t is varray(200) of varchar2(25);
Results t;

  Stmt CONSTANT VARCHAR2(4000) :=
    'SELECT '|| p_colname || ' FROM '|| p_tablename ;

BEGIN
  DBMS_Output.Put_Line ('SQL Stmt: ' || Stmt);
  EXECUTE IMMEDIATE Stmt bulk collect into Results;
for j in 1..Results.Count() loop
DBMS_Output.Put_Line(Results(j));
end loop;
--EXCEPTION WHEN OTHERS THEN
  --Raise_Application_Error(-20000, 'Wrong table name');
END show_col;
/

execute show_col('Email','EMPLOYEES');
execute show_col('Email','EMP');
```

Următorul atac *SQL Injection* se execută cu succes:

```
execute show_col('Email','EMPLOYEES where 1=2 union select Username c1
from All_Users --');
```

Codul aşteaptă un nume *SQL*, dar apelantul a utilizat un şir de caractere care conduce la următoarea instrucţiune *SQL*:

```
SELECT username c1 FROM all_users ;
```

Rescriem procedura utilizând *DBMS_ASSERT.SIMPLE_SQL_NAME*:

```
CREATE OR REPLACE
PROCEDURE show_col2 (p_colname varchar2, p_tablename   varchar2)
AS
type t is varray(200) of varchar2(25);
Results t;
Stmt CONSTANT VARCHAR2(4000) :=
    'SELECT '||dbms_assert.simple_sql_name( p_colname ) || ' FROM '||
dbms_assert.simple_sql_name( p_tablename ) ;

BEGIN
  DBMS_Output.Put_Line ('SQL Stmt: ' || Stmt);
 EXECUTE IMMEDIATE Stmt bulk collect into Results;
for j in 1..Results.Count() loop
DBMS_Output.Put_Line(Results(j));
end loop;
--EXCEPTION WHEN OTHERS THEN
  --Raise_Application_Error(-20000, 'Wrong table name');
END show_col2;
/


execute show_col2('Email','EMPLOYEES');
execute show_col2('Email','EMP');
execute  show_col2('Email','EMPLOYEES where 1=2 union select Username
c1 from All_Users --');
```

Încercarea anterioară va eşua, deoarece funcţia *SIMPLE_SQL_NAME* filtrează intrarea utilizatorului înainte de execuţie. Dacă este găsit un nume *SQL* invalid, instrucţiunea eşuează şi previne astfel atacul *SQL Injection*.