KAKO PISATI ORACLE PL/SQL PROGRAME OTPORNE NA NAPADE SQL INJEKCIJOM

SAŽETAK

SQL injekcija je tehnika injekcije programskog koda koja iskorištava sigurnosnu ranjivost u bazi podataka.
Dinamički generiranoj SQL naredbi napadač dodaje kod koji mijenja semantiku te SQL naredbe.
U radu je prikazano kako se, pridržavanjem Oracle preporuka, može napisati PL/SQL programski kod koji je otporan na napade SQL injekcijom.

ABSTRACT

SQL injection is a code injection technique that exploits a security vulnerability occurring in the database.
Dynamically generated SQL statements are semantically changed by the attacker who adds code.
The paper describes how, by following Oracle recommendations, we can write PL/SQL code that is resistant to SQL injection attacks.

1. UVOD

SQL je deklarativan programski jezik, koji radi sa skupovima podataka, tj. može vraćati (čitati) ili ažurirati (unositi, mijenjati, brisati) ne samo jedan redak podataka, već čitav skup redaka (skup se može sastojati od nula, jednog ili samo par redaka, ali i od jako velikog broja redaka). SQL se u programiranju ne koristi samostalno, već se koristi zajedno s nekim drugim programskim jezikom, uglavnom proceduralnim (pod pojam *proceduralni jezik* uključujemo i uobičajene objektno-orijentirane jezike). Česti način povezivanja SQL-a s drugim jezikom je taj da se SQL ugrađuje u drugi jezik, koji tada obično nazivamo jezik-domaćin (engl. host language). Jezik-domaćin može biti npr. jezik opće namjene kao što je Java, C#, C++ i dr., ili neki jezik koji je specifično pisan za rad sa SQL-om kao što je Oracle PL/SQL.

U ovisnosti od toga kako se SQL poziva u jeziku-domaćinu, moguća je veća ili manja osjetljivost SQL koda na napad SQL injekcijom. SQL injekcija je tehnika injekcije programskog koda koja iskorištava sigurnosnu ranjivost u bazi podataka. Dinamički generiranoj SQL naredbi napadač dodaje kod koji mijenja semantiku te SQL naredbe. Treba napomenuti da statički SQL nije osjetljiv na napade injekcijom. No, ne znači da je sav dinamički SQL osjetljiv na napade injekcijom – samo određeni dinamički SQL kod je osjetljiv na te napade.

Provjeri (ne)otpornosti SQL koda na napad injekcijom može se pristupiti na (barem) dva različita načina. Jedan je način danas uobičajeniji, a to je testiranje otpornosti na napad injekcijom. Slično stanje je i kod pisanja softvera – uglavnom se ispravnost softvera testira, a rjeđe se dokazuje. No, kao što bi kod pisanja softvera trebalo naći pravu mjeru između dokazivanja korektnosti programa (na više ili manje formalan način) i naknadnog testiranja korektnosti programa, tako bi i kod povećanja otpornosti programskog koda na sigurnosne ranjivosti trebalo primijeniti sličan pristup. Specifično to vrijedi i za povećanje otpornosti programskog koda na napad SQL injekcijom. Drugim riječima, nije dovoljno raditi npr. samo penetracijska testiranja, već treba već kod pisanja programskog koda obratiti pažnju na to da on bude što je moguće sigurnosno otporniji.

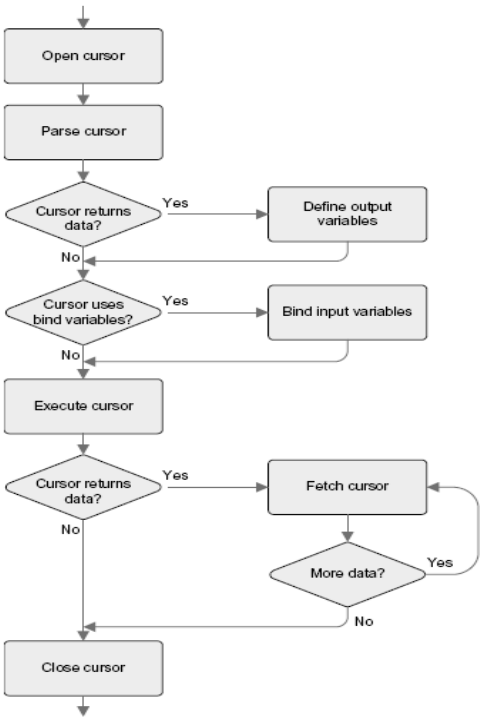
Firma Oracle je u zadnjih par godina posvetila značajnu pažnju povećanju otpornosti programskog koda na napad SQL injekcijom. Njezini su stručnjaci napravili preporuke za pisanje programskog koda koje, ako ih se držimo, garantiraju otpornost koda na SQL injekciju (iako se to, vjerojatno, ne može matematički dokazati), i prikazali ih u materijalu [7]. U ovom se radu daje skraćeni i pojednostavljeni prikaz tih preporuka, pa je za detaljan uvid svakako preporučljivo proučiti taj materijal. No, treba naglasiti da je riječ o preporukama koje se tiču isključivo pisanja PL/SQL programskog koda, tj. onoga gdje je jezik-domaćin PL/SQL, a ne npr. Java, C#, C++ i dr. Također, preporuke se odnose isključivo na PL/SQL kod koji se izvodi unutar Oracle baze, a ne na PL/SQL kod koji se izvodi unutar npr. Forms ili Reports programa (tj. na strani klijenta ili aplikacijskog servera). Zapravo, ne samo za ranjivost na napade SQL injekcijom, već i za sve ostale ranjivosti baze podataka vrijedi da je uvijek sigurnije PL/SQL kod pisati unutar baze podataka. A, najčešće se time postižu i bolje performanse kod izvođenja, te se piše programski kod koji je modularniji, manje je redundantan i lakše se mijenja.

2. PROCESIRANJE SQL NAREDBI; STATIČKI I DINAMIČKI SQL

U Oracle relacijskom sustavu sve se SQL naredbe, bez obzira na koji način bile građene, na kraju izvode na isti način, kao tzv. kursori baze podataka. U Oracle sustavu kursor baze podataka je privatno SQL područje u kojem se čuvaju informacije za procesiranje određene SQL naredbe. Kod procesiranja SQL naredbe dešavaju se sljedeće radnje:

- Otvaranje (OPEN) kursora. Alociraju se memorijske strukture za kursor u privatnoj memoriji serverskog procesa pridruženog sesiji – UGA memoriji (User Global Area). SQL naredba još nije pridružena kursoru;
- Parsiranje (PARSE) tj. gramatička analiza kursora. SQL naredba pridružuje se kursoru. Njena parsirana reprezentacija, koja uključuje plan izvršenja (engl. execution plan) učitava se u dio SGA memorije (System Global Area), tzv. library cache. Strukture u UGA memoriji se ažuriraju tako da pokazuju na djeljivi kursor u library cache-u;
- Definiranje izlaznih (DEFINE OUTPUT) varijabli. Izlazne varijable nisu nužne samo za SELECT naredbu, već i za INSERT, UPDATE, DELETE naredbe koje imaju RETURNING klauzulu;
- Povezivanje ulaznih (BIND INPUT) varijabli;
- Izvršavanje (EXECUTE) kursora. Kod nekih naredbi, npr. kod najvećeg broja upita, realno procesiranje zapravo čeka sljedeću fazu;
- Dohvaćanje (FETCH) kursora. Ako SQL naredba vraća retke, to se dešava upravo u ovom koraku. Kod upita je upravo ovo korak u kojem se dešava najviše realnog procesiranja;
- Zatvaranje (CLOSE) kursora. Resursi koji su pridruženi kursoru u UGA memoriji se oslobađaju. Međutim, djeljivi kursor u library cache memoriji se ne briše, već ostaje za eventualno ponovno korištenje u budućnosti.

Sljedeća slika (iz [1]), grafički prikazuje te korake, onako kako se najčešće (ali ne uvijek) dešavaju:



Kursori mogu u programskom kodu biti "skriveni" ili mogu biti "vidljivi" - govorimo o implicitnim i eksplicitnim kursorima.

Kod PL/SQL implicitnog kursora koriste se samo DML naredbe (INSERT, UPDATE, DELETE) ili SELECT naredba, kao u ovom primjeru:

```
SELECT stupac1, stupac2
  INTO varijabla1, varijabla2
  FROM tablica;
```

Postoje različite varijante eksplicitnog kursora, a ova je jedna od jednostavnijih:

```
DECLARE
  CURSOR art_c IS
    SELECT sifra, naziv
      FROM m_artikli;
BEGIN
  FOR redak IN art_c
  LOOP
    DBMS_OUTPUT.PUT_LINE
     (redak.sifra || ' ' ||
      redak.naziv);
  END LOOP;
END;
```

Prethodna dva primjera prikazuju tzv. **statički SQL**. Budući da je pojam *statički SQL* semantički dosta preopterećen, autori [7] predlažu da se, barem za potrebe govora o SQL injekciji, radije upotrebljava pojam **ugrađeni SQL** (engl. embeded SQL). Takav kod nije osjetljiv na napad SQL injekcijom!

Uz statički, ili kako ga je bolje nazvati ugrađeni SQL, postoji i **dinamički SQL**. Moglo bi se očekivati da se dinamički SQL kod uvijek dinamički stvara tokom izvođenja programa, no to nije uvijek tako. Naziv *dinamički* zapravo zbunjuje, jer je moguće da i takav kod ne bude promijenjen za vrijeme izvođenje programa i, kako će se poslije vidjeti, u tom slučaju (kad se ne mijenja) isto nije podložan napadu SQL injekcijom! No, činjenica je da se dinamički kod, bez obzira da li se stvarno mijenja tokom izvođenja ili ne, sintaktički i semantički provjerava od strane PL/SQL kompajlera/interpretera tek kod izvođenja programa. Može se reći da je prednost dinamičkog koda da omogućava fleksibilno (pa i generičko) programiranje, a mana dinamičkog koda je upravo ta da se provjerava tek u fazi izvođenja, što znači da se u toku izvođenja mogu pojaviti neke greške koje bi se u statičkom kodu otkrile unaprijed, u vrijeme prevođenja.

Dinamički kod (SQL i PL/SQL) u Oracle bazi pojavio se prvi puta u verziji Oracle 7, i to kao **paket DBMS_SQL**. Treba napomenuti da, iako se paket zove DBMS_SQL, on omogućava ne samo dinamički SQL, nego i dinamički PL/SQL. Primjer korištenja paketa DBMS_SQL za dinamički SQL:

```
DECLARE
  stmt_str VARCHAR2(200);
  cur_hdl INT;
  rows_processed INT;
  name VARCHAR2 (10);
  salary INT;
BEGIN
  -- otvaranje kursora
  cur_hdl:=
    dbms_sql.open_cursor;
  -- definiranje SELECT naredbe
  stmt_str :=
    'SELECT ename, sal' ||
    '  FROM emp' ||
    ' WHERE job = :jobname';
  dbms_sql.parse (
    cur_hdl, stmt_str,
    dbms_sql.native);
  dbms_sql.bind_variable (
    cur_hdl, 'jobname',
    'SALESMAN');
  dbms_sql.define_column
    (cur_hdl, 1, name, 200);
  dbms_sql.define_column
    (cur_hdl, 2, salary);
  rows_processed :=
    dbms_sql.execute (cur_hdl);

  LOOP
    IF dbms_sql.fetch_rows
      (cur_hdl) > 0
    THEN
      dbms_sql.column_value
        (cur_hdl, 1, name);
      dbms_sql.column_value
       (cur_hdl, 2, salary);
      -- <procesiranje podataka>
    ELSE
      EXIT;
    END IF;
  END LOOP;
   dbms_sql.close_cursor (cur_hdl);
END;
```

Zbog složenosti rada sa paketom DBMS_SQL, Oracle je u verziji 8i napravio tzv. **prirodni dinamički SQL (engl. Native Dynamic SQL – NDS)**. Kao i paket DBMS_SQL, tako i prirodni dinamički SQL omogućava ne samo dinamički SQL, već i dinamički PL/SQL. Jedan primjer - korištenje naredbe EXECUTE IMMEDIATE:

```
CREATE FUNCTION broj_redaka
  (tablica_p VARCHAR2)
  RETURN NUMBER
IS
  naredba VARCHAR2 (1000);
  broj_redaka NUMBER;
BEGIN
  naredba :=
    'SELECT COUNT (*)' ||
    '  FROM ' || tablica_p;
  EXECUTE IMMEDIATE naredba
    INTO broj_redaka;
  RETURN broj_redaka;
END;
```

Uz naredbu EXECUTE IMMEDIATE, NDS čini i tzv. dinamička kursor varijabla. U nastavku slijedi primjer rada sa dinamičkom kursor varijablom koji je ekvivalentan prije navedenom primjeru rada sa DBMS_SQL paketom:

```
DECLARE
  cur SYS_REFCURSOR;
  stmt_str VARCHAR2(200);
  name VARCHAR2(20);
  salary NUMBER;
BEGIN
  stmt_str :=
    'SELECT ename, sal' ||
    '  FROM emp' ||
    ' WHERE job = :1';
  OPEN cur FOR stmt_str
    USING 'SALESMAN';
  LOOP
    FETCH cur INTO name, salary;
    EXIT WHEN cur%NOTFOUND;
    --<procesiranje podataka>
  END LOOP;
  CLOSE cur;
END;
```

Vidimo koliko je ovaj primjer jednostavniji od primjera rada s paketom DBMS_SQL. No, paket DBMS_SQL ima i nekih svojih prednosti. Naime, paket DBMS_SQL omogućava tzv. metodu 4 kod rada sa dinamičkim SQL-om. Četiri metode rada sa dinamičkim SQL-om su sljedeće:

- Metoda 1 dozvoljava samo DML naredbe, i to bez "bind" varijabli;
- Metoda 2 dozvoljava i DML naredbe sa "bind" varijablama;
- Metoda 3 dozvoljava i upite, tj. naredbu SELECT, ali ona mora imati fiksan broj stupaca i fiksan broj "bind" varijabli;
- Metoda 4 dozvoljava da broj stupaca i broj "bind" varijabli u SELECT naredbi bude nepoznat sve do trenutka izvođenja naredbe.

3. PRIMJERI KODA OSJETLJIVOG NA NAPAD SQL INJEKCIJOM

Kako može doći do napada SQL injekcijom, najbolje se prikazuje na primjerima. Slijede primjeri iz materijala [7].

Pretpostavimo da imamo ovakav dio dinamičkog koda (zanemarimo to što ovakav programski kod nikako ne bi trebalo pisati, jer ne koristi bind varijable; zapravo, u konkretnom slučaju uopće ne treba koristi dinamički SQL):

```
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;
...
```

Kako se vidi, definirane su dvije znakovne (varchar2) konstante. Konstanta *q* postavlja se na vrijednost jednostrukog navodnika (jednostruki navodnik se kod pridruživanja mora napisati kao dva puta jednostruki navodnik). Konstanta *SQL_VC2_Literal* dobije se tako da se prvom dijelu (SELECT) dodaje konstanta *q*, pa onda ono što je korisnik unio (što se nalazi u varijabli Raw_User_Input), te opet konstanta *q.* Ako korisnik unese npr. *Smith*, onda će se dobiti sljedeća ("normalna") naredba:

```
select c2 from t where c1 = 'Smith'
```

Međutim, pretpostavimo da je maliciozni (i vješti) korisnik unio ovo:

```
'
union
select Username c1 from All_Users --
```

Vrlo je važno da je na kraju unio niz -- , koji u jeziku PL/SQL označava početak komentara. Tako je izveden sljedeći kod:

```
select c1 from t where c2 = ''
union
select Username c1 from All_Users --'
```

koji je istovjetan sljedećem kodu (zato što prvi SELECT ne vraća ništa, a komentar se može zanemariti):

```
select Username c1 from All_Users
```

Dakle, očito je da postoji veliki nesrazmjer između onoga što je programer htio postići dok je pisao kod i onoga što je na kraju izvedeno. Napadač je iskoristio ranjivost napisanog koda i izvršio napad pomoću SQL injekcije - dodao je kod koji očito mijenja semantiku naredbe koju je zamislio programer!

Drugi primjer, u kojem je programer želio omogućiti dinamički WHERE uvjet:

```
Stmt := 'select c1 from t where '
  || Raw_User_Input;
execute immediate Stmt
  bulk collect into v;
```

Pretpostavimo da je maliciozni korisnik unio ovo:

```
0 = 1
union
select Username c1 from All_Users
```

pa je izveden sljedeći kod:

```
select c1 from t where 0 = 1
union
select Username c1 from All_Users
```

koji je istovjetan sljedećem kodu (zato što prvi SELECT ne vraća ništa):

```
select Username c1 from All_Users
```

I ovdje postoji veliki nesrazmjer između onoga što je programer htio postići i onoga što je na kraju izvedeno.

Iz ovih primjera može se intuitivno razumjeti da do SQL injekcije dolazi onda kada PL/SQL program izvršava SQL naredbu čiji se tekst kreira za vrijeme izvođenja na temelju korisnikovog unosa, a taj se unos ne provjerava. Naravno, bilo bi najbolje kada tekst SQL naredbe ne bi ovisio o korisnikovom unosu, već bi se u cijelosti mogao izvesti iz izvornog koda programa koji izvodi tu PL/SQL naredbu. Međutim, ponekad postoje takvi zahtjevi koji traže da se tekst SQL naredbe kreira na temelju korisnikovog unosa. Tada treba osigurati da taj korisnikov unos ne može dovesti do SQL injekcije.

4. DRUGAČIJA KLASIFIKACIJA SQL NAREDBI; NEKE DEFINICIJE

Kako je rečeno u 2.točki, uobičajena je klasifikacija SQL (i PL/SQL) koda na statički i dinamički SQL. Već je navedeno da autori u [7] predlažu da se, umjesto pojma *statički SQL* radije upotrebljava pojam *ugrađeni SQL*. Ono što se inače uobičajeno naziva dinamički SQL (sa Oracle podvarijantama DBMS_SQL i NDS), u [7] se dalje dodatno razlaže.

Za potrebe preciznijeg objašnjenja o tome kako pisati kod koji je otporan na SQL injekciju, u [7] se definiraju neki pojmovi koji će ukratko biti prikazani u nastavku.

Predložak SQL sintakse (engl. SQL syntax template) izgleda na prvi pogled kao regularna SQL naredba, ali njegova namjena je da opiše skup svih mogućih SQL naredbi koje se mogu izvesti iz tog predloška (može se reći da su konkretne SQL naredbe instance tog predloška). Primjer jednog predloška je:

select &&1 from &&2 where &&3 = &4

Jedan znak *&* predstavlja **rezervirano mjesto za (određenu) vrijednost** (engl. value placeholder), a dva znaka *&&* predstavljaju **rezervirano mjesto za (jednostavno) SQL ime** (engl. simple SQL name placeholder). Pritom atribut *jednostavno* (ime) znači da ime nije složeno od dva ili više dijelova koji se zajedno nalaze unutar dvostrukih navodnika, što Oracle baza dozvoljava. Naime, u Oracle bazi moguće je imati ne-jednostavno ime, npr. ime tablice "Moja tablica". No, u nastavku ćemo uvijek raditi s jednostavnim imenima, jer se u [7] dozvoljavaju samo takva imena. U navedenom primjeru predloška vidimo tri rezervirana mjesta za imena (ime stupca u SELECT klauzuli, ime tablice u FROM klauzuli i ime stupca u WHERE klauzuli), te jedno rezervirano mjesto za vrijednost (&4).

Napomenimo da u ovoj notaciji (za predloške) rezervirano mjesto za vrijednost nije isto što i rezervirano mjesto za vrijednost u uobičajenoj SQL sintaksi (za SQL naredbe) – rezervirano mjesto za vrijednost u predlošku može predstavljati uobičajeno rezervirano mjesto za vrijednost u SQL naredbi, ili može označavati dobro oblikovan SQL literal (konstantu određenog tipa).

Na temelju zadanog "općeg" predloška SQL sintakse moguće je izvesti čitav niz "specijaliziranih" predložaka. Npr. na temelju prethodnog "općeg" predloška moguće je izvesti (i) ove "specijalizirane" predloške:

```
select c1 from &&1 where c2 = &1

select &&1 from t where &&2 = 99

select c1 from &&1 where c2 = :1

select c1 from t where c2 = :1
```

Intuitivno se vidi da je kod "specijalizacije" dopušteno da se na rezervirano mjesto za (jednostavno) ime stavi konkretno (jednostavno) ime, a na rezervirano mjesto za vrijednost može se staviti ili literal (konstanta) ili uobičajeno rezervirano mjesto za vrijednost u SQL naredbi (npr. :1 ; uobičajeno se to zove vezana varijabla, engl. bind variable).

Uz prethodni pojam (*predložak SQL sintakse*), u [7] se definiraju pojmovi **tekst SQL naredbe fiksiran kod kompajliranja** (engl. compile-time-fixed SQL statement text) i **tekst SQL naredbe kreiran kod izvođenja** (engl. run-time-created SQL statement text).

Kako samo ime govori, tekst SQL naredbe fiksiran kod kompajliranja ne može se mijenjati u toku izvođenja, i on se u cijelosti može saznati kod čitanja izvornog koda. Mora se naglasiti da u tekst SQL naredbe fiksiran kod kompajliranja spadaju i ugrađeni SQL (tj. ono što obično nazivamo *statički SQL*) i dinamički SQL koji se sastoji samo od "dijelova" koji su, zapravo, statički varchar2 izrazi. Takav dinamički SQL nije podložan SQL injekciji. No, niti sav dinamički SQL koji spada u drugu vrstu – u tekst SQL naredbe kreiran kod izvođenja - nije uvijek podložan SQL injekciji, kako će se vidjeti u nastavku!

U [7] se p*redložak SQL sintakse* dalje dijeli na dvije vrste – na **statički predložak** (preciznije - statički predložak SQL sintakse, engl. static SQL syntax template) i na **dinamički predložak** (preciznije - dinamički predložak SQL sintakse, engl. dynamic SQL syntax template). Treba naglasiti da statički predložak ne služi samo za dobivanje SQL naredbi čiji je tekst fiksiran kod kompajliranja, već u nekim slučajevima i za dobivanje SQL naredbi čiji je tekst

kreiran kod izvođenja. Slijedi jedan primjer (iz [7]) koji prikazuje taj slučaj. U ovom primjeru želi se napisati programski kod koji će zaključati određeni redak određene tablice, ali na način da se, ako je neka druga sesija baze već zaključala taj redak, na otključavanje čeka određeno vrijeme koje je definirano od strane korisnika. U Oracle bazi to se može napraviti jedino tako da se napiše programski kod koji će tekst SQL naredbe kreirati kod izvođenja (jer vrijeme čekanja za otključavanje ne može biti varijabla):

```
function f(
  PK in t.PK%type,
  Wait_Time in pls_integer)
return t.c1%type
authid Definer
is
  c1 t.c1%type;
  Stmt constant varchar2(32767) :=
    'select c1 from t where PK = :b
        for update wait '
    || Wait_Time;
begin
  execute immediate Stmt
    into c1 using PK;
  return c1;
end f;
```

Očito je da se u ovom primjeru tekst SQL naredbe kreira za vrijeme izvođenja, kad korisnik definira vrijednost parametra *Wait_Time*. Međutim, u ovom primjeru je ipak riječ o statičkom predlošku, jer se on može vidjeti iz izvornog koda i izgleda ovako:

```
select c1 from t where PK = :b
   for update wait &1
```

Treba primijetiti da u ovom slučaju nema opasnosti od SQL injekcije. No, to nije uvijek slučaj sa statičkim predlošcima koji kreiraju SQL naredbe kod izvođenja. Npr. u 3.točki imali smo, u 1.primjeru, statički predložak

```
select c2 from t where c1 = &1
```

koji je bio neotporan na SQL injekciju.

Za razliku od prethodnog primjera, gdje je predložak bio statički (iako je njime definiran dinamički SQL koji se kreira kod izvođenja), slijedi primjer dinamičkog predloška. U praksi ima takvih primjera gdje bi se teoretski mogao koristiti tekst SQL naredbe fiksiran kod kompajliranja (bilo statički ili dinamički), ali iz praktičnih razloga koristimo tekst SQL naredbe kreiran kod izvođenja, ali i dinamičke predloške. Npr. česti je zahtjev da broj stupaca u SELECT klauzuli može odrediti korisnik. Pretpostavimo da je riječ o samo jednoj tablici koja ima 20 stupaca. Ako želimo da korisnik može odabrati bilo koju varijantu od jednog do svih 20 stupaca, takvih varijanti ima 2**20 – 1 (više od milijun). Teoretski bi se mogao koristiti tekst SQL naredbe fiksiran kod kompajliranja (bilo statički ili dinamički) u kojem bi se (npr. pomoću IF ili CASE naredbi) odredila varijanta koju je odabrao korisnik, no praktično rješenje je tekst SQL naredbe kreirati kod izvođenja, i to pomoću dinamičkog predloška, npr. ovako:

```
procedure p(
  PK in t.PK%type, Wanted in cw)
is
  function Col_List return varchar2
  is ... end Col_List;

  Stmt constant varchar2(32767) :=
   'select '
   || Col_List()
   || ' Report from t where PK = :b';

  Report varchar2(32767);
begin
  execute immediate Stmt
    into Report using PK;
  DBMS_Output.Put_Line(Report);
end p;
```

Funkcija Col_List izgleda ovako:

```
function Col_List return varchar2 is
  type cn is varray(20)
    of varchar2(30);
  Col_Names constant cn :=
    cn('c1', 'c2', ... , 'c20');
  Seen_One boolean := false;
  List varchar2(32767);
begin
  for j in 1..Wanted.Count() loop
    if Wanted(j) then
      List :=
        List
        || case Seen_One
           when true then '||'
           else ''
           end
        || 'Rpad
        ('||Col_Names(j)||', 10)';
      Seen_One := true;
    end if;
  end loop;
  return List;
end Col_List;
```

Može se vidjeti da ovdje ne postoji statički predložak, već je predložak dinamički. No, gledajući izvorni kod može se zaključiti da u toku izvođenja možemo imati 20 različitih predložaka – sa 1 do 20 stupaca. Primijetimo da u ovom slučaju dinamički predložak nije stvorio opasnost od SQL injekcije, ali to nije uvijek tako!

Na temelju uvedenih pojmova, može se preciznije definirati SQL injekcija kao pojava da se kod izvršavanja PL/SQL potprograma (neka) SQL naredba po svojem predlošku razlikuje od onoga (predloška) koji je želio ostvariti autor potprograma, zato jer napadač injektira tekst u SQL naredbu. Injektirani tekst može doći u potprogram kroz formalne parametre potprograma (to je tzv. **napad prvog reda**), ili indirektno, npr. čitanjem tablice iz baze (to je tzv. **napad drugog reda**).

SQL injekcija ne može se desiti ako je tekst SQL naredbe fiksiran kod kompajliranja (bilo da je to statički ili dinamički SQL). Može se desiti samo ako je tekst SQL naredbe kreiran kod izvođenja, bez obzira da li je dobiven statičkim ili dinamičkim predloškom. Cilj je da se i takav tekst (tj. kreiran kod izvođenja) napiše na način da bude otporan na SQL injekciju, što se prikazuje u sljedećoj točki.

5. GARANTIRANJE SIGURNOSTI SQL LITERALA I (JEDNOSTAVNOG) SQL IMENA

U predlošku SQL sintakse, dvije vrste elemenata mogu biti zamijenjene dinamičkim tekstom – rezervirano mjesto za vrijednost može biti zamijenjeno literalom, te rezervirano mjesto za (jednostavno) SQL ime može biti zamijenjeno sa (jednostavnim) SQL imenom. Baš te zamjene predstavljaju rizik od SQL injekcije.

Kako osigurati SQL literal? Postoje tri vrste SQL literala: tekstualni, datumski (zapravo, on sadrži datum i vrijeme – datetime) i numerički. Svaki od njih se štiti na određeni način.

U 3.točki vidjeli smo primjer nesigurnog SQL tekstualnog literala. Sigurni SQL tekstualni literal gradi se na ovakav način:

- Svaka pojava pojedinačnog navodnika u PL/SQL (tekstualnoj) vrijednosti zamjenjuje se sa dva navodnika. Ovo osigurava da se ne desi run-time greška;
- Na početak i kraj vrijednosti dodaje se (konkatenira) po jedan navodnik;
- Dobivena vrijednost provjerava se pomoću Oracle funkcije DBMS_Assert.Enquote_Literal(). Ova provjera je krucijalna za sprečavanje SQL injekcije.

Prije nego se vidi kako se gradi sigurni SQL datumski literal, korisno je pokazati kako uopće datumski literal može biti nesiguran. Najbolje je to vidjeti na primjeru (iz [7]). Pretpostavimo da imamo sljedeću proceduru:

```
procedure p is
  q constant varchar2(1) := '''';

  d constant date :=
    To_Date('2008-09-22 17:30:00',
     'yyyy-mm-dd hh24:mi:ss');

  Stmt constant varchar2(32767) :=
    'select t.PK from t where t.d > '
    || q||d||q;
...
begin
  execute immediate Stmt
    bulk collect into Results;
...
```

Proceduru bismo uobičajeno pozvali tako da prije postavimo neki "normalni" datumski format, npr.:

```
alter session set NLS_Date_Format =
  'dd-Mon-yy hh24:mi:ss'
/
begin p(); end;
/
```

No, pretpostavimo da je napadač (koji ima pravo raditi ALTER SESSION) napravio sljedeće:

```
alter session set NLS_Date_Format =
  '"'' and Scott.Evil()=1 --"'
/
begin p(); end;
/
```

Kao rezultat, bit će kreirana sljedeća SQL naredba:

```
select t.PK from t where t.d > ''
   and Scott.Evil()=1 --'
```

Naredba očito nema semantiku SQL predloška iz kojeg je nastala - funkcija *Evil* može napraviti svašta! Ovaj način SQL injekcije obično se naziva **lateralna injekcija** - nije došla direktno kroz korisnikov unos, već sa strane (u ovom slučaju kroz NLS parametar).

Sigurni SQL datumski literal gradi se na sljedeći način:

- Koristi se funkcija To_Char(d, Fmt), koja datumski literal pretvara u tekstualni, pri čemu parametar Fmt (format) mora biti u skladu sa funkcionalnom specifikacijom;
- Na početak i kraj vrijednosti dodaje se (konkatenira) po jedan navodnik;

- Dobivena vrijednost provjerava se pomoću Oracle funkcije DBMS_Assert.Enquote_Literal();

- U SQL naredbi koristi se funkcija To_Date(t, Fmt) pri čemu je Fmt (format) jednak onome u funkciji To_Char.

Slijedi primjer koji prikazuje primjenu tog postupka:

```
procedure p_Safe(d in date) is
  q constant varchar2(1) := '''';

  -- Choose precision
  -- according to purpose.
  Fmt constant varchar2(32767) :=
    'J hh24:mi:ss';

  Safe_Date_Literal constant
    varchar2(32767) :=

  Sys.DBMS_Assert.Enquote_Literal
   (q||To_Char(d, Fmt)||q);

  Fmt_Literal constant
    varchar2(32767) := q||Fmt||q;

  Safe_Stmt constant varchar2(32767)
  :=
  ' insert into t(d) values(To_Date('
  || Safe_Date_Literal
  || ', '
  || Fmt_Literal
  || '))';
begin
  execute immediate Safe_Stmt;
...
```

Kod SQL numeričkog literala, do SQL injekcije može doći na sličan način kao kod datumskog literala - preko lateralne injekcije. Razlika je u tome što se kod numeričkog literala može zlouporabiti (pomoću ALTER SESSION SET) parametar NLS_Numeric_Characters, a kod datumskog literala to je bio parametar NLS_Date_Format.

Sigurni SQL numerički literal gradi se na sljedeći način:

- Koristi se funkcija To_Char (primjer će biti prikazan u sljedećoj točki);

- Eksplicitno se navodi vrijednost '.,' kao ona koja se postavlja u parametar NLS_Numeric_Characters.

Osim garantiranja sigurnosti SQL literala, potrebno je garantirati sigurnost (jednostavnog) SQL imena. SQL ime može se općenito prikazati u obliku a.b.c@db_link . Suština sigurnosti je u tome da se (dvije) točke i znak @ uvijek eksplicitno navedu u SQL predlošku, tj. da se kod zamjene mogu mijenjati samo dijelovi *a, b, c* i *db_link*, koji predstavljaju jednostavna imena. To se radi tako da se dizajnira odgovarajući API kroz koji će korisnik moći unijeti odgovarajuće dijelove (*a, b, c, db_link*), a API će na odgovarajuća mjesta dodati (dvije) točke i znak @. Tako dobivena vrijednost SQL imena provjerava se pomoću funkcije DBMS_Assert.Simple_Sql_Name().

6. PRAVILA ZA SPREČAVANJE SQL INJEKCIJE

Pravila koja slijede navedena su u [7] i vrijede samo za sprečavanje SQL injekcije unutar PL/SQL programa u Oracle bazi. Pravila se, barem za sada (u bazi Oracle 11g), ne mogu provjeravati automatski od strane baze, već je potrebna ručna provjera. U nekim kasnijim verzijama Oracle baze moguće je da će postojati alati koji će barem neka od tih pravila provjeravati automatski.

Izložiti bazu klijentima samo kroz PL/SQL API

Klijentu treba omogućiti da se poveže na bazu isključivo kroz korisnički račun baze (engl. database user) koji ima samo privatne sinonime koji pokazuju na PL/SQL programske module aplikacijske sheme (koja je, zapravo, neki drugi korisnički račun). Jedino pravo koje treba imati korisnički račun je pravo izvršavanja PL/SQL modula (EXECUTE) aplikacijske sheme. Na taj način, ranjivo mjesto baze mogu biti jedino ti PL/SQL moduli.

Koristiti samo tekst SQL naredbe koji je fiksiran kod kompajliranja – ako je moguće

Ako je tekst SQL naredbe fiksiran kod kompajliranja, tada je takva naredba potpuno zaštićena od SQL injekcije. Pritom je uvijek bolje koristiti ugrađeni (statički) SQL, a ako to nije moguće, onda dinamički SQL. Dinamički SQL (ako je nužan) treba biti napravljen pomoću naredbe EXECUTE IMMEDIATE koja koristi jednu PL/SQL konstantu, koja je (konstanta) sastavljena samo od statičkih VARCHAR2 izraza.

Koristiti statički predložak (SQL sintakse) za tekst SQL naredbe kreiran kod izvođenja – ako je moguće

Ako je uopće potrebno imati tekst SQL naredbe koji se kreira tek kod izvođenja, onda njega (ako je moguće) treba dobiti pomoću statičkog predloška. U statičkom predlošku se rezervirana mjesta za vrijednosti zamjenjuju literalom, te se rezervirana

mjesta za (jednostavno) SQL ime zamjenjuju (jednostavnim) SQL imenom. Ta se zamjena mora obaviti na siguran način, koji je opisan u prethodnoj točki.

U slučajevima složenijih zahtjeva, koristiti DBMS_SQL API

Postoje neki slučajevi, kao što je npr. zahtjev za kreiranjem WHERE klauzule čija kompozicija neće biti poznata do izvođenja programa, koji naizgled traže da se naredba nekontrolirano gradi kroz slobodno definirani tekst. Međutim, iako to nije uvijek jednostavno, takvi se zahtjevi u pravilu mogu realizirati na kontroliran način, pomoću DBMS_SQL API-a.

Formalizacija pravila za garantiranje sigurnosti (od SQL injekcije)

Kako je navedeno u [7], formalne definicije (koje slijede u nastavku) pomaže auditoru PL/SQL koda da lakše utvrdi sigurnost koda na SQL injekciju:

- **Statički tekst** je ili PL/SQL statički varchar2 izraz, ili izraz koji je dobiven konkatenacijom statičkih tekstova, ili lokalna varijabla kojoj je jasno pridružena vrijednost koja je statički tekst (primjećuje se da je definicija rekurzivna);

- **Dinamički tekst** je bilo koji tekst koji nije statički tekst. Primjeri su npr. formalni parametri, ili varijable deklarirane na vršnoj razini u paketu, a nemaju *constant* ključnu riječ;

- **Sigurni dinamički tekst** je izlaz funkcija
  DBMS_Assert.Enquote_Literal()
  ili
  To_Char (x f,
  'NLS_Numeric_Characters = ".,"')
  ili
  DBMS_Assert.Simple_Sql_Name()
  (kako je bilo navedeno u prethodnoj točki);

- **Sigurni tekst SQL naredbe** je bilo koja konkatenacija statičkog teksta i sigurnog dinamičkog teksta.

7. ZAKLJUČAK

Osiguranju otpornosti SQL koda na napad injekcijom mora se pristupiti kod pisanja koda. Stručnjaci firme Oracle napravili su preporuke za pisanje programskog koda koje, ako ih se držimo, garantiraju otpornost koda na SQL injekciju. U ovom je radu dat skraćeni i pojednostavljeni prikaz tih preporuka. Riječ je o preporukama koje se tiču isključivo pisanja PL/SQL programskog koda, i odnose se isključivo na PL/SQL kod koji se izvodi unutar Oracle baze.

Najvažnija pravila jesu:

- Izložiti bazu klijentima samo kroz PL/SQL API;

- Koristiti samo tekst SQL naredbe koji je fiksiran kod kompajliranja – ako je moguće;

- Ako prethodno nije moguće, koristiti statički predložak (SQL sintakse) za tekst SQL naredbe kreiran kod izvođenja, ako je moguće. Pritom osim statičkog teksta treba koristiti samo siguran dinamički tekst, tj. dinamički tekst koji je provjeren (tj. koji je izlaz) funkcija

  DBMS_Assert.Enquote_Literal()
  ili
  To_Char (x f,
  'NLS_Numeric_Characters = ".,"')
  ili
  DBMS_Assert.Simple_Sql_Name()

- U slučajevima složenijih zahtjeva, koristiti DBMS_SQL API umjesto nekontroliranog, slobodno definiranog teksta.

Literatura:

[1] Antognini, C. (2008): Troubleshooting Oracle Performance, Apress

[2] Ben-Natan, R. (2009): HOWTO Secure and Audit Oracle 10g and 11g, Auerbach Publications

[3] Feuerstein S. i Pribyl B. (2009): Oracle PL/SQL Programming (5.izdanje), O'Reilly

[4] Knox, D, C. (2004): Effective Oracle Database 10g Security by Design, McGraw-Hill

[5] Kyte, T. (2005): Expert Oracle Database Architecture, Apress

[6] Litchfield, D. (2007): The Oracle Hacker's Handbook: Hacking and Defending Oracle, Wiley

[7] Oracle Corporation (2008): How to write SQL injection proof PL/SQL, Oracle White Paper, http://www.oracle.com/technetwork/database/features/plsql/overview/how-to-write-injection-proof-plsql-1-129572.pdf (studeni 2010.)

[8] Oracle Corporation (2010): Oracle Database Security Guide 11g Release 2 (11.2), Oracle manual

Podaci o autoru:

Zlatko Sirotić, dipl.ing.
Istra informatički inženjering d.o.o., Pula
e-mail: zlatko.sirotic@iii.hr

Autor radi više od 25 godina na informatičkim poslovima, najviše od toga u poduzeću Istra informatički inženjering d.o.o, Pula, gdje radi i sada. Radio je na svim poslovima u izradi softvera: analiza, dizajn, programiranje, uvođenje, obuka korisnika, održavanje. Oracle proizvode (baza, Designer, Forms, Reports, JDeveloper) koristi više od 12 godina.

Objavljivao je stručne radove na kongresima "Hotelska kuća", na konferencijama CASE, KOM, HrOUG (Hrvatska udruga Oracle korisnika), u časopisima "InfoTrend" i "Ugostiteljstvo i turizam", a neka njegova programska rješenja objavljena su na web stranicama firmi Quest i Oracle.