

## SQL Injection and Oracle, Part Two

Pete Finnigan 2002-11-28

### SQL Injection and Oracle, Part Two

byPete Finnigan

last updated November 28, 2002

---

This is the second part of a two-part article that will examine SQL injection attacks against Oracle databases. The [first installment](#) offered an overview of SQL injection and looked at how Oracle database applications are vulnerable to this attack, and looked at some examples. This segment will look at enumerating the privileges, detecting SQL injection attacks, and protecting against SQL injection.

#### Enumerating the Privileges

Access to SQL inject an Oracle database is great, but what would an attacker look for to gain an advantage or a potential step up. He would, of course, need to enumerate the user he had access to and see what that user can see and do. I will show a few examples here to give the reader an idea of what is possible.

In this example, we are logged in as the user *dbsnmp* and the `get_cust` procedure has been modified to select three columns from our sample table. If we use a union to extend an existing `select` statement then the new SQL in the union must select the same number of columns and data types as the existing hijacked `select` otherwise an error occurs, see the following:

```
SQL> exec get_cust('x' union select 1,'Y' from sys.dual where 'x'='x');
debug:select customer_phone,customer_forname,customer_surname from customers
where customer_surname='x' union select 1,'Y' from sys.dual where 'x'='x'
-1789ORA-01789: query block has incorrect number of result columns
```

The main `select` has three `varchar` columns but we select two columns and one is a number; as a result, an error occurs. Back to enumeration, first get the objects that the user we are logged in as can see:

```
SQL> exec get_cust('x' union select object_name,object_type,'x' from user_obj
ects where 'x'='x');
debug:select customer_phone,customer_forname,customer_surname from customers
where customer_surname='x' union select object_name,object_type,'x' from
user_objects where 'x'='x'
::CUSTOMERS:TABLE:x
::DBA_DATA_FILES:SYNONYM:x
::DBA_FREE_SPACE:SYNONYM:x
::DBA_SEGMENTS:SYNONYM:x
::DBA_TABLESPACES:SYNONYM:x
::GET_CUST:PROCEDURE:x
::GET_CUST2:PROCEDURE:x
::GET_CUST_BIND:PROCEDURE:x
::PLSQ:DATABASE LINK:x
```

Then get the roles that have been allocated directly to the user:

```
SQL> exec get_cust('x' union select granted_role,admin_option,default_role from
  user_role_privs where 'x'='x');
debug:select customer_phone,customer_forname,customer_surname from customers
where customer_surname='x' union select granted_role,admin_option,default_role
from user_role_privs where 'x'='x'
::CONNECT:NO:YES
::RESOURCE:NO:YES
::SNMPAGENT:NO:YES
```

Then find out the system privileges that are granted directly to the user:

```
SQL> exec get_cust('x' union select privilege,admin_option,'X' from user_sys_
privs where 'x'='x');
debug:select customer_phone,customer_forname,customer_surname from customers
where customer_surname='x' union select privilege,admin_option,'X' from
user_sys_privs where 'x'='x'
::CREATE PUBLIC SYNONYM:NO:X
::UNLIMITED TABLESPACE:NO:X
```

Selecting from the table USER\_TAB\_PRIVS will give the privileges granted directly to the user on objects.

There are many system views that start USER\_%, these show objects and privileges that are granted to the current user as well as details about objects owned by the user. For instance, there are 168 views or tables in Oracle 8.1.7, so this gives an indication of the amount of detail that can be learned about the user you are logged in as. These USER\_% views do not include all the many privileges and options available to the current user; however, besides those specifically granted, any user also can include all of the objects that have permissions granted to PUBLIC.

PUBLIC is a catch-all that is available to all users in the Oracle database. There is a good set of views, known as the ALL\_% views, that is similar in construction to the USER\_% views. These include every item available to the current user, including PUBLIC ones. A good place to start is the view ALL\_OBJECTS, as it has a similar structure to USER\_OBJECTS and will display every object and its type available to the current user. A good query to see all of the objects, their types and owner available would be:

```
select count(*),object_type,owner
from all_objects
group by object_type,owner
```

The V\$ views is also a good set of views, provided they are available to the user. These give information about the current instance, performance, parameters, and the like. V\$PARAMETER, which gives all of the database instance initialization parameters, including details of the UTL\_FILE directories is a good example. V\$PROCESS and V\$SESSION are another pair of views that will give details of current sessions and processes. These will tell the user who is logged on, where they are logged in from, and what program they are using, etc.

In conclusion to this exploration section it is worth mentioning that because I wanted to make easy examples that anyone with a copy of the Oracle RDBMS could try out, I used a PL/SQL procedure to demonstrate the techniques and obviously I had access to my source code. It made it easy for me to understand exactly the SQL I could send successfully without causing errors.

In the real world, in a Web-based environment, or in a network-based application, the source code would probably not be available. As a result, working out how to get successful SQL to send will probably require trial and error. If error messages are returned to the user either directly from the Oracle RDBMS or from the application, then it is usually possible to work out where to change the SQL. An absence of error messages makes it harder but not impossible. All of the Oracle error messages are quite well documented and are available on-line on a Unix system with the `oerr` command or with the HTML documentation provided with Oracle CDs on any platform. (Remember anyone can get a copy of Oracle to use to learn the product.) They are also on-line, along with the complete Oracle documentation, at <http://tahiti.oracle.com/>.

Having knowledge of Oracle and of the schema of the user being used is also a great advantage. Quite obviously, some of this knowledge is not hard to learn, so the lesson is that in case anyone is able to SQL inject into your database then you need to minimize what they can do, see, or access.

## Detecting SQL Injection

Oracle is a large product and is applied in many diverse uses, so to say that SQL injection can be detected would be wrong; however, in some cases, it should be possible for the DBA or security admin to spot whether or not this technique is being used. If abuse is thought to be taking place then forensic investigations can be done using the redo logs. A GUI tool called [Log Miner](#) is available from Oracle to allow the redo logs to be analysed. However, this has serious restrictions: until version 9i, select statements could not be retrieved. The redo logs allow Oracle to replay all of the events that altered data in the database, this is part of the recovery functionality. It is possible to see all statements and data that has been altered. Two PL/SQL standard packages, `DBMS_LOGMNR` and `DBMS_LOGMNR_D`, are available, these packages allow the redo logs to be queried from the command line for all statements processed.

The extensive Oracle audit functionality can be utilized but, again, unless you know what you are looking for, finding evidence of SQL injection taking place could like finding a needle in a haystack. The principle of least privilege should be observed in any Oracle database so that only those privileges that are actually needed are granted to the application database users. This simplifies (minimizes) what can be legally done and, as a result, makes any actions outside the scope of these users easier to spot. For instance, if the application user should have access to seven tables and three procedures and nothing else, then using Oracle audit to record select failures on all other tables would enable an administrator to spot any attempted access to any table outside the applications realm. This can be done, for example, for one table with the following audit command:

```
SQL> audit select on dbsnmp.customers by access whenever not successful;
```

Audit succeeded.

A simple script can be built to generate the audit statements for the tables needed. There should be no real performance issues with this audit, as no other tables should be accessed by the application. As a result, it should not therefore generate audit. Of course, if someone successfully accesses a table outside the realm, it would not be captured. This is merely intended as a first step.

The same audit principles can be used to audit DDL, inserts and update failures or successes. The new SANS guide (see references) has a whole chapter on audit.

Another idea could be to watch the SQL executed and look for any dodgy SQL. A good script called *peep.sql* can be used to access the SQL executed from the SGA is one called from <http://www.oriolo.com/frameindexSA.html>, search down the list of free scripts and get it. The script gives the SQL statements in the SGA with the worst performance times. It can be easily modified to remove the execution time restraints and bring back all SQL in the SGA. A script such as this can be scheduled on a regular basis and then the SQL that is returned can be used to *guess* if any SQL injection has been attempted. I say "guess" because it is virtually impossible to know all legal pieces of SQL an application generates; therefore, the same applies to spotting illegal ones. A good first step would be to identify all statements with "union" included or `or` statements with `'x'='x'` type lines. There could be performance issues with extracting all of the SQL from the SGA regularly!

The best cure of course is prevention!

## Protecting against SQL Injection

On the surface, protection against SQL injection appears to be easy to implement but, in fact, it is not as easy as it looks. The solutions fall into two distinct areas:

- Do not allow dynamic SQL that uses concatenation, or at least filter the input values and check for special characters such as quote symbols.
- Use the principle of least privilege and ensure that the users created for the applications have the privileges needed and all extra privileges (such as PUBLIC ones) are not available.

This section cannot go into great detail; such a discussion would constitute an entire article in itself. However, certain basic measures can be taken. These actions fall into two sections:

- Review the application source code. The code can be written in many different languages, such as PHP, JSP, java, PL/SQL VB, etc., so the solutions vary. However, they all follow a similar pattern. Review the source code for dynamic SQL where concatenation is used. Find the call that parses the SQL or executes it. Check back to where values are entered. Ensure that input values are validated and that

quotes are matched and metacharacters are checked. Reviewing source code is a task that is specific to the language used.

- Secure the database and ensure that all excess privileges are revoked.

Some other simple tips to follow include:

- If possible, do not use dynamic PL/SQL. The potential for damage is much greater than for dynamic SQL, as then there is scope to execute any SQL, DDL, PL/SQL etc.
- If dynamic PL/SQL is necessary then use bind variables.
- If PL/SQL is used use AUTHID CURRENT\_USER so that the PL/SQL runs as the logged in user and not the creator of the procedure, function or package.
- If concatenation is necessary then use numeric values for the concatenation part. This way strings cannot be passed in to add SQL.
- If concatenation is necessary then check the input for malicious code, i.e. check for union in the string passed in or metacharacters such as quotes.
- For dynamic SQL if it is necessary use bind variables. An example is shown below:

We first need to alter our simple procedure to allow the dynamic part passed in to use a bind variable. This is shown here:

```
create or replace procedure get_cust_bind (lv_surname in varchar2)
is
    type cv_typ is ref cursor;
    cv cv_typ;
    lv_phone      customers.customer_phone%type;
    lv_stmt       varchar2(32767):='select customer_phone ' ||
                    'from customers ' ||
                    'where customer_surname=:surname';
begin
    dbms_output.put_line('debug:' || lv_stmt);
    open cv for lv_stmt using lv_surname;
    loop
        fetch cv into lv_phone;
        exit when cv%notfound;
        dbms_output.put_line('::' || lv_phone);
    end loop;
    close cv;
exception
    when others then
        dbms_output.put_line(sqlcode || sqlerrm);
end get_cust_bind;
/
```

First we execute with a genuine value, in this case "Clark", to show that the correct records are returned. We then we try to SQL inject this procedure and find it doesn't work:

```
SQL> exec get_cust_bind('Clark');
debug:select customer_phone from customers where customer_surname=:surname
::999444888
::999777888
```

PL/SQL procedure successfully completed.

```
SQL> exec get_cust_bind('x' union select username from all_users where 'x'='
x');
debug:select customer_phone from customers where customer_surname=:surname
```

Some more pointers:

- Encrypt sensitive data so that it cannot be viewed.
- Revoke all PUBLIC privileges where possible from the database
- Do not allow access to UTL\_FILE, DBMS\_LOB, DBMS\_PIPE, DBMS\_OUTPUT, UTL\_HTTP,UTL\_SMTP or any other standard or application packages that allow access to the O/S.
- Change database default passwords.
- Run the listener as a non-privileged user.
- Ensure that minimum privileges are granted to application users.
- Restrict PL/SQL packages that can be accessed from apache.
- Remove all example scripts and programs from the Oracle install.

### **Final thoughts**

I hope that this article has given an overview of some of the possibilities of SQL injecting Oracle and done so with simple examples that most readers can try. Again, SQL injection is a relatively simple technique and on the surface protecting against it should be fairly simple; however, auditing all of the source code and protecting dynamic input is not trivial, neither is reducing the permissions of all applications users in the database itself. Be vigilant, grant what is needed, and try and reduce dynamic SQL to the minimum.

*Pete Finnigan is a freelance consultant specialising in Oracle and security of Oracle. Pete is currently working in the UK financial sector and has recently completed the new [Oracle security step-by-step guide](#) for the [SANS institute](#). Pete has many years of development and administration experience in many languages. Pete is regarded as one of the worlds leading experts on Oracle security.*

*Watch for the forthcoming book [The SANS Institute Oracle Security Step-by-step - A survival guide for Oracle security](#) written by Pete Finnigan with consensus achieved by experts from over 53 organizations with over 230 years of Oracle and security experience. Due to be published in the next few weeks by the SANS Institute.*

## Relevant Links

All of the code from this paper is available from the author's Web site from the scripts menu - SQL and PL/SQL option.  
[www.petefinnigan.com](http://www.petefinnigan.com)

Protecting Oracle Databases Whitepaper

*Aaron Newman, Application Security Inc*

Hackproofing Oracle Application Servers

*David Litchfield, NGSSoftware Insight Security Research*

*Rain Forest Puppy*

RFPlutonium to fuel your PHP-Nuke

*Rain Forest Puppy*

NT Web Technologies Vulnerabilities

*Rain Forest Puppy*

[Privacy Statement](#)

Copyright 2006, SecurityFocus