

# Detecting SQL Injection in Oracle

Pete Finnigan 2003-07-22

## Introduction

Last year I wrote a two-part paper about SQL Injection and Oracle. That paper explored which SQL injection techniques are possible with Oracle, gave some simple examples on how SQL injection works and some suggestions on how to prevent attackers and malicious employees using these methods. Those SQL Injection papers can be found here:

- ["SQL injection and Oracle - part one"](#)
- ["SQL injection and Oracle - part two"](#)

This paper takes the subject further and investigates the possibilities for the Oracle Database Administrator (DBA) to detect SQL injection in the wild against her Oracle database. Is it possible to detect SQL injection happening? If so what tools and techniques can be employed to achieve this?

The main focus of this paper is to explore some simple techniques in extracting logging and trace data that could be employed for monitoring. The aim is to show the reader what data is readily available so they can make their own mind up about what can be useful. The paper will not cover commercial solutions. Because a true SQL injection tool would involve writing a parser or filter to analyse the SQL statements a fully featured tool is unfortunately beyond the scope of a short paper - I leave the implementation of such a tool to interested readers.

Example code given in this paper can be obtained from <http://www.petefinnigan.com/sql.htm>.

## Can SQL Injection be detected?

The short answer is definitely yes... err... well err... probably... that is, yes it is possible to detect SQL injection but probably not all of the time for all cases and not always in real time. The reasons for this are many and complicated:

- There are many different forms of SQL injection attacks that can take place - these are limited only by the hacker's imagination and the DBA's foresight (or lack thereof) to protect the database and provide the *least privileges necessary*.
- Identifying SQL that shouldn't be there is not simple. The reason SQL injection is possible is because of the use of dynamic SQL in applications. This intended dynamic SQL means that the set of all legal SQL statements is harder if impossible to define. If the legal statements are impossible to define then so are the illegal ones.
- Distinguishing normal administration from an attacker is not always easy as an attacker can steal an administrator's account.
- Detecting SQL injection inevitably involves parsing the SQL statement for possible additions or

truncations to it. Table names and view names need to be extracted and checked to see if they should be altered.

- For a technique to be useful it should not affect the performance of the database too much.
- Corroborating data such as usernames and timestamps are also need to be extracted at the same time.
- Many more...

It is possible to detect SQL injection attempts in general and specifically against Oracle. How can we do this, and what data is available? This paper attempts to explore these questions.

The first step is to define the boundary conditions, or what actions are to be detected and then to look at the possible free solutions within Oracle and how these can be used to good effect.

## Some possible commercial solutions

There are no real commercial solutions that specifically detect SQL injection attempts against an Oracle database. There are a reasonable number of firewall products that incorporate an Oracle proxy and a few IDS tools that claim to support Oracle. A number of companies are at present seriously looking into the design and development of a true application IDS for Oracle, and perhaps these tools will detect SQL injection. At present most of the commercial tools to be used properly would need rules and signatures to be defined for the specific Oracle cases.

## Some free solutions

The ideal list of all possible SQL injection types or signatures is impossible to define in totality but a good starting point would cover the following possibilities:

- SQL where the addition of a *union* has enabled the reading of a second table or view
- SQL where an unintentional *sub-select* has been added.
- SQL where the *where clause* has been *short-circuited* by the addition of a line such as 'a'='a' or 1=1
- SQL where built-in or bespoke package procedures are called where they should not be.
- SQL where access is made to system tables and/or application user and authentication tables.
- SQL where the *where clause* has been truncated with a comment i.e --
- Analysis of certain classes of errors - such as multiple errors indicating that select classes have the wrong number of items or wrong data types. This would indicate someone trying to create an extra select using a *union*.

The key is to keep it simple at first; trying to do something too complicated with ad-hoc and built in tools will never work efficiently and effectively. It is important to not get too clever with SQL and the assumptions about what is legal SQL and what is hacker-created SQL. Beware of the false positives. Keep it simple and be proactive - use more than one method if possible and extend and learn.

Any tool or system employed to detect SQL injection could identify most of the above list of possibilities. In

trying to identify where the data can come from to analyse SQL, the following steps should be possible for one or more of the techniques:

- *Grab* the SQL as it is sent to the database or as soon after as possible
- Analyse the SQL to check for some or all of the above cases that indicate SQL injection
- Obtain user and timestamp data

Concentrating on grabbing the SQL and whether it is possible to get timestamp and user info as well as possible further analysis leads to the following list of possibilities:

- Pre-existing packet sniffers / IDS tools such as *snort* (not included in the experiments below)
- A free packet sniffer such as *snoop*
- Oracle networking trace files
- Oracle server trace files
- Extracting the SQL from the Oracle server memory (SGA)
- Use of a tool such as Oracle *Log Miner* and possibly the raw analysis of redo logs
- Oracle Audit
- Database Triggers
- Fine grained audit (FGA)

There are some issues to be aware of. The audit facilities can rarely be used for more than a *smoking gun*. If Oracle advanced options are used to encrypt network traffic then extracting the SQL from the network will be difficult. If trace facilities are used they tend to generate huge amounts of data and consume system resources. Any method that does not allow the detection of *select* statements whilst trapping others is really not useful.

If it is not possible to detect SQL injection taking place in real time, it is better to know after the fact than to not know it is happening at all.

## Worked examples

Next we can work through some simple examples of a SQL injection attempt using one of the examples from my previous papers. The first step is to create the sample customer table, add data, and also create the demonstration procedure *get\_cust*.

An example SQL injection attempt that will be used below to see if it is detected is:

```
SQL> exec get_cust('x' union select username from all_users where 'x'='x');
debug:select customer_phone from customers where customer_surname='x' union
select username from all_users where 'x'='x'
::AURORA$JIS$UTILITY$
::AURORA$ORB$UNAUTHENTICATED
::CTXSYS
```

: :DBSNMP

: :EMIL

: :FRED

Let us now explore what trace, packet, audit and internal information is available that records any evidence of running this query.

## Log Miner

Oracle provides two database package procedures DBMS\_LOGMNR and DBMS\_LOGMNR\_D that allow archive logs and on-line redo logs to be analysed. The redo logs contain all the information to replay every action in the database. These are used for point in time recovery and for transaction and data consistency. However there are some serious problems with the *Log Miner* functionality. These can be listed as follows:

- If an MTS database is used, *Log Miner* cannot be used due to the internal memory allocation of this tool. *Log Miner* uses PGA memory which would not be visible to each thread used in Multi Threaded Server (MTS).
- The tool doesn't properly support chained and migrated rows and also objects are not fully supported. Analysis of index-only tables and clusters are also not supported. The tool can still be used even though the output has gaps in it.
- The SQL generated by *Log Miner* is not the same SQL executed by the user. This is because the redo logs store enough data to change the data at row and column level and so the original compound statements cannot be reproduced.

Some advantages of *Log Miner* are:

- Analysis doesn't have to be done in the source database so archive logs could be moved to a dedicated database for the whole organisation and analysed offline.
- There is a GUI tool available via the Oracle Enterprise Manager (OEM)

Also, to make the use of this tool realistic the database has to be in ARCHIVELOGMODE and *transaction\_auditing* needs to be *true* in the initialisation file for user information to be included.

This is a very effective tool for after the fact analysis and forensics to find out exactly when some event occurred within the database and who did it. It can be used successfully to help recover, for instance, a table deleted by accident.

Redo logs can also be analysed by hand the hard way. A good paper demonstrating this can be found [here](#) (PDF).

Now we can run through the example and explore the contents of the archive logs. First check if the database is in ARCHIVELOGMODE, determine where the archive logs are written to, and finally that username auditing is

on.

```
SQL> select log_mode from v$database;
```

```
LOG_MODE
```

```
-----
```

```
ARCHIVELOG
```

```
SQL> select name,value from v$parameter
       2  where name in('log_archive_start','log_archive_dest');
```

```
NAME
```

```
-----
```

```
VALUE
```

```
-----
```

```
log_archive_start
```

```
TRUE
```

```
log_archive_dest
```

```
/export/home/u01/app/oracle/admin/emil/archive
```

To detect which user executed a command:

```
SQL> select name,value from v$parameter
       2  where name = 'transaction_auditing';
```

```
NAME
```

```
-----
```

```
VALUE
```

```
-----
```

```
transaction_auditing
```

```
TRUE
```

Now execute the SQL injection attempt and then use *Log Miner* to see what is recorded. To make the analysis easier for this example, the archive log is saved before and after to ensure only this command is in the log:

```
SQL> connect sys as sysdba
```

```
Enter password:
```

```
Connected.
```

```
SQL> alter system archive log current;
```

```
System altered.
```

```
SQL>
SQL> connect dbsnmp/dbsnmp@emil
Connected.
SQL> set serveroutput on size 100000
SQL> exec get_cust('x' union select username from all_users where 'x'='x');
debug:select customer_phone from customers where customer_surname='x' union
select username from all_users where 'x'='x'
::AURORA$JIS$UTILITY$
::AURORA$ORB$UNAUTHENTICATED
::CTXSYS
::DBSNMP
::EMIL
<records snipped>
::SYS
::SYSTEM
::WKSYS
::ZULIA
```

PL/SQL procedure successfully completed.

```
SQL> connect sys as sysdba
Enter password:
Connected.
SQL> alter system archive log current;

System altered.
```

SQL>

First create the *Log Miner* dictionary:

```
SQL> set serveroutput on size 1000000
SQL> exec dbms_logmnr_d.build('logmnr.dat','/tmp');
LogMnr Dictionary Procedure started
LogMnr Dictionary File Opened
TABLE: OBJ$ recorded in LogMnr Dictionary File
TABLE: TAB$ recorded in LogMnr Dictionary File
TABLE: COL$ recorded in LogMnr Dictionary File
TABLE: TS$ recorded in LogMnr Dictionary File
<output snipped>
Procedure executed successfully - LogMnr Dictionary Created
```

PL/SQL procedure successfully completed.

SQL>

Find the correct archive log file:

```
SQL> select name
      2  from v$archived_log
      3  where completion_time=(select max(completion_time) from v$archived_log);
```

NAME

```
-----
/export/home/u01/app/oracle/admin/emil/archive/1_7.dbf
```

SQL>

Now load the archive log file into *Log Miner*:

```
SQL> exec dbms_logmnr.add_logfile('/export/home/u01/app/oracle/admin/emil/archive/1_7.dbf',sys.
dbms_logmnr.NEW);
```

PL/SQL procedure successfully completed.

```
SQL> exec dbms_logmnr.start_logmnr(dictFileName => '/tmp/logmnr.dat');
```

PL/SQL procedure successfully completed.

SQL>

Finally, search the results:

```
SQL> select scn,username,timestamp,sql_redo
      2  from v$logmnr_contents
```

SQL>

<snipped>

SCN	USERNAME	TIMESTAMP	SQL_REDO
253533	DBSNMP	16-JUN-03	set transaction read write;
253533	DBSNMP	16-JUN-03	update "SYS"."AUD\$" set "ACTION#" = '101', "RETURNCODE" = '0', "LOGOFF\$LREAD" = '228',

```
"LOGOFF$PREAD" = '0',
"LOGOFF$LWRITE" = '10',
"LOGOFF$DEAD" = '0',
"LOGOFF$TIME" =
TO_DATE('16-JUN-2003
12:16:12', 'DD-MON-YYYY
```

```
SCN USERNAME          TIMESTAMP SQL_REDO
-----
HH24:MI:SS'), "SESSIONCPU" =
'5' where "ACTION#" = '100'
and "RETURNCODE" = '0' and
"LOGOFF$LREAD" IS NULL and
"LOGOFF$PREAD" IS NULL and
"LOGOFF$LWRITE" IS NULL and
"LOGOFF$DEAD" IS NULL and
"LOGOFF$TIME" IS NULL and
"SESSIONCPU" IS NULL and ROWID
= 'AAAABiAABAAAAEWAAX';
```

```
SCN USERNAME          TIMESTAMP SQL_REDO
-----
253534 DBSNMP          16-JUN-03 commit;
```

<snipped output>

The first thing that can be noticed is that *Log Miner* does not process select statements and display the output in 9i. The *Log Miner* package doesn't support selects as they are not stored in the redo logs. It is possible to use *Log Miner* to read on-line redo logs but I will leave that to the reader to experiment with. Even though SQL injection can be detected in insert, delete and update statements, *Log Miner* is not suitable for detecting SQL injection. This is due to its lack of being able to detect select statements as well as some of the other issues mentioned above.

**Packet sniffing**

The main issue with packet sniffing for connections to the Oracle database is that the Oracle network protocol is proprietary and not published. Does that matter for trying to ascertain if SQL injection attempts have taken place? Probably yes, as access to the protocol would allow a better design and efficient tool for this task. Without access to the protocol and no wish to reverse engineer it, the task is limited to grabbing ASCII text strings from the wire. There are both advantages and disadvantages with this method:

## Advantages:

- A system could be implemented on a separate server allowing real time analysis without impacting the source database.

## Disadvantages:

- This method can be resource intensive.
- The packets need to be sniffed close to the source and on the same subnet to ensure all packets pass in front of the sniffer.
- If the Oracle advanced security options for encrypting of network packets or any other third party solution is used to encrypt, sniffing packets will not work.
- If, as in our example, the SQL injection attempt is passed as a call to a package procedure then the true internal dynamic SQL will not be visible. Instead you will simply see the typed in command.
- Packet sniffing every packet will generate a huge amount of data. Piping the packets through a filter program is a solution to mitigate this issue.

To demonstrate, we will use *snoop* on Solaris to see what is visible within network packets. Start up *snoop* and fire the SQL from a SQL\*Plus session:

```
root:jupiter> snoop -t a -x 0 jupiter and port 1521 | strings
Using device /dev/hme (promiscuous mode)
15:06:34.31348 172.16.240.3 -> jupiter      TCP D=1521 S=1404      Ack=299902194 Seq=26460609
Len=174 Win=8413
   0: 0800 2092 9d88 00a0 ccd3 a550 0800 4500      .. .....P..E.
  16: 00d6 6884 4000 8006 596d ac10 f003 ac10      ..h.@...Ym.....
  32: f00b 057c 05f1 0193 c1c1 11e0 24f2 5018      ...|.....$.P.
  48: 20dd 0f36 0000 00ae 0000 0600 0000 0000      ..6.....
  64: 1169 36a4 61de 0001 0101 0303 5e37 0304      .i6.a.....^7..
  80: 0021 0078 71de 0001 52bc 39de 0001 0a00      .!.xq...R.9.....
  96: 0000 00e0 39de 0000 0101 0000 0000 0000      ....9.....
 112: 0000 0000 0000 0000 0000 0000 0000 0000      .....
 128: e239 de00 4245 4749 4e20 6765 745f 6375      .9..BEGIN get_cu
 144: 7374 2827 7827 2720 756e 696f 6e20 7365      st('x' union se
 160: 6c65 6374 2075 7365 726e 616d 6520 6672      lect username fr
 176: 6f6d 2061 6c6c 5f75 7365 7273 2077 6865      om all_users whe
 192: 7265 2027 2778 2727 3d27 2778 2729 3b20      re 'x'='x');
 208: 454e 443b 0a00 0101 0101 0000 0000 0001      END;.....
 224: 0800 0105      ....
15:06:34.33281      jupiter -> 172.16.240.3 TCP D=1404 S=1521      Ack=26460783
Seq=299902194 Len=54 Win=24820
   0: 00a0 ccd3 a550 0800 2092 9d88 0800 4500      .....P.. .....E.
```

```

16: 005e 094a 4000 4006 f91f ac10 f00b ac10    .^.J@.@.....
32: f003 05f1 057c 11e0 24f2 0193 c26f 5018    .....|..$....oP.

```

An immediate success with this method is apparent, because the SQL is captured with the SQL injection attempt in it. As with other methods in this article, to take this further a few things would need to be done:

- A filter program would need to be implemented that could parse out the SQL statements and determine if any potential SQL injection attempt has taken place.
- To be of any real use the filter program would also need to extract the timestamp from the packet header as well as the source IP address.
- Extracting the database user would be extremely difficult as previous packets would need to be inspected to extract login information. This would also suggest the need to store previous packets or information about them and their sequence to do this.

As a simple solution, packet sniffing would appear to be an option provided a reasonably simple filter/parser program can be written in Perl or C. The goal would be to output the possible wrongdoing by extracting the SQL, timestamp and src and dest IP from the packet stream.

### Sniffing packets within oracle (sqlnet trace)

Extracting network information closer to Oracle is possible by using the trace facility of SQL\*Net, Net\*8 or Oracle networking (whichever name is relevant to your database release).

Trace facilities are available for most of the Oracle networking tools such as the listener, Oracle names, connection manager, names control utility and of course the Oracle networking client and server. In this example we can concentrate on the server trace. It is possible to trace from the client end but it would be necessary to do so for all clients and thus harder to manage.

There are some disadvantages to using Oracle networking trace files as a tool to look for SQL injection:

- The trace files can grow very quickly and use an enormous amount of disk space. If not managed correctly there is a danger of filling a disk and a possible denial of service taking place.
- There is an overhead involved in writing the trace files.
- Even though it is possible to define a unique trace file name and location, Oracle appends a process ID (PID) to the trace file name. This is the operating system PID of the shadow process. The pid can be seen with the following SQL:

```

SQL> select p.spid,s.username
       2  from v$session s,v$process p
       3  where s.paddr=p.addr;

```

```

SPID          USERNAME

```

```
-----
<records snipped>
```

```
616          DBSNMP
556          SYSTEM
```

```
9 rows selected.
```

```
SQL>
```

To enable trace simply add the following lines to the \$ORACLE\_HOME/network/admin/sqlnet.ora file:

```
TRACE_FILE_SERVER=pf_trace.trc
TRACE_DIRECTORY_SERVER=/tmp
TRACE_LEVEL_SERVER=SUPPORT
```

The parameters define where the trace is to be written, what it is called and also the level. There are four levels that can be used: OFF, USER, ADMIN and SUPPORT. They rise in detail from OFF to SUPPORT; the SUPPORT level includes the contents of the network packets.

Let's run our example again from SQL\*Plus on a Windows client and see what is generated in the trace file. The trace file is as expected, pf\_trace\_616.trc. Wow, this file is 4005 lines in length just from connecting and executing the following:

```
SQL> exec get_cust('x' union select username from all_users where 'x'='x');
```

```
PL/SQL procedure successfully completed.
```

Searching in the trace for the packet dump we can find the SQL injection attempt sent to the database:

```
nsprecv: 165 bytes from transport
nsprecv: tlen=165, plen=165, type=6
nsprecv: packet dump
nsprecv: 00 A5 00 00 06 00 00 00 |.....|
nsprecv: 00 00 03 5E 35 03 04 00 |...^5...|
nsprecv: 21 00 78 71 DE 00 01 52 |!.xq...R|
nsprecv: BC 39 DE 00 01 0A 00 00 |.9.....|
nsprecv: 00 00 E0 39 DE 00 00 01 |...9....|
nsprecv: 01 00 00 00 00 00 00 00 |.....|
nsprecv: 00 00 00 00 00 00 00 00 |.....|
nsprecv: 00 00 00 00 00 00 00 E2 |.....|
nsprecv: 39 DE 00 42 45 47 49 4E |9..BEGIN|
nsprecv: 20 67 65 74 5F 63 75 73 | get_cus|
```

```

nsprecv: 74 28 27 78 27 27 20 75 |t('x' u|
nsprecv: 6E 69 6F 6E 20 73 65 6C |nion sel|
nsprecv: 65 63 74 20 75 73 65 72 |ect user|

nsprecv: 6E 61 6D 65 20 66 72 6F |name fro|
nsprecv: 6D 20 61 6C 6C 5F 75 73 |m all_us|
nsprecv: 65 72 73 20 77 68 65 72 |ers wher|
nsprecv: 65 20 27 27 78 27 27 3D |e 'x'='|
nsprecv: 27 27 78 27 29 3B 20 45 |'x'); E|
nsprecv: 4E 44 3B 0A 00 01 01 01 |ND;.....|

```

Whilst using SQL\*Net trace files are a possibility, there are a number of issues besides the items listed above:

- Once again a parser would be needed to extract the SQL text from the packet dumps and to then identify from a set of rules whether the SQL might be a SQL injection attempt.
- The PID used as part of the trace file name is not known until the user connects and the shadow process it started. If the filenames were not unique then like *snoop* the trace could be piped through a filter and the disk usage issue would go away.
- The trace files could be monitored regularly with a simple shell script that checks for trace files where there is no longer a shadow process running. These can then be processed and removed. Long running sessions would generate huge trace files and not be ideally managed in this way.
- It is easier to determine the OS user and database user involved with SQL\*Net traces as this information can be found from the v\$process and v\$session views with knowledge of the OS PID.

This method is not practical for implementing a simple SQL injection utility in an organisation. The resource issues alone would make it difficult to manage and work with. It could be used sparingly when a suspected incident has occurred and where trace usage could be controlled to a narrow time band or number of clients.

### Internal Oracle trace files

Oracle can also generate trace files from the Oracle kernel when SQL is executed. This can be enabled at the RDBMS level by setting the initialisation parameter SQL\_TRACE=true in the *init.ora* file. It can also be turned on just for the instance by doing:

```
ALTER SYSTEM SET SQL_TRACE=TRUE;
```

Trace can also be turned on for the current session with:

```
ALTER SESSION SET SQL_TRACE=TRUE;
```

Before trace is turned on the parameter *timed\_statistics* should be set to TRUE in the initialisation file. Trace files will be written to the file pointed at by the *user\_dump\_dest* parameter in the initialisation file.

Let's create a trace file and analyse it for our simple SQL injection example as follows:

```
SQL> alter session set sql_trace=true;
```

Session altered.

```
SQL> exec get_cust('x' union select username from all_users where 'x'='x');
```

PL/SQL procedure successfully completed.

```
SQL>
```

A trace file has been created with a name format of {SID}\_ora\_{PID}.trc. The format is platform specific. The PID is the OS PID as for the SQL\*Net trace file and again it can be determined from querying the views v\$sqlsession and v\$sqlprocess. The raw trace file can be read and the SQL in question can be seen as follows:

```
*** 2003-06-16 16:54:01.429
*** SESSION ID:(8.29) 2003-06-16 16:54:01.408
APPNAME mod='SQL*Plus' mh=3669949024 act='' ah=4029777240
=====
PARSING IN CURSOR #3 len=33 dep=0 uid=17 oct=42 lid=17 tim=1055782441429190 hv=3
732290820 ad='846c85c0'
alter session set sql_trace=true
END OF STMT
EXEC #3:c=0,e=1,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=4,tim=1055782441407935
*** 2003-06-16 16:54:34.876
=====
PARSING IN CURSOR #3 len=82 dep=0 uid=17 oct=47 lid=17 tim=1055782474876639 hv=3
204430447 ad='8482555c'
BEGIN get_cust('x' union select username from all_users where 'x'='x'); END;
END OF STMT
PARSE #3:c=0,e=6430,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=4,tim=1055782474876611
=====
```

A better way is to post process the trace file with a utility called *tkprof* as follows:

```
oracle:jupiter> tkprof emil_ora_616.trc output.trc sys=yes
```

```
TKPROF: Release 9.0.1.0.0 - Production on Mon Jun 16 16:59:50 2003
```

(c) Copyright 2001 Oracle Corporation. All rights reserved.

The generated file contains quite a lot of information, including timing and user ID. The timing would need to be read from the raw trace file but the user ID can be found from the *tkprof* output. The processed output shows the PL/SQL package being called and also the dynamic SQL string:

<output snipped>

```
BEGIN get_cust('x' union select username from all_users where 'x'='x');
      END;
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	1
Fetch	0	0.00	0.00	0	0	0	0
total	2	0.00	0.01	0	0	0	1

Misses in library cache during parse: 1

Optimizer goal: CHOOSE

Parsing user id: 17

<output snipped>

```
select customer_phone
from
  customers where customer_surname='x' union select username from all_users
  where 'x'='x'
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.01	0.02	0	0	0	0
Execute	1	0.01	0.00	0	0	1	0
Fetch	37	0.01	0.00	0	184	5	36
total	39	0.03	0.03	0	184	6	36

Misses in library cache during parse: 1

Optimizer goal: CHOOSE

Parsing user id: 17 (recursive depth: 1)

Rows Row Source Operation

```
36 SORT UNIQUE
```

```
36 UNION-ALL
```

```
<output snipped>
```

The user ID can be read from the database as:

```
SQL> select username,user_id
2 from dba_users
3 where user_id=17;
```

```
USERNAME                                USER_ID
-----                                -
DBSNMP                                  17
```

```
SQL>
```

Trace files clearly have promise for implementing a SQL injection detection system but they also have some serious problems:

- Trace would need to be turned on globally all of the time.
- Trace generation would consume system resources; how much depends on the type of database and application.
- A huge amount of trace files would quickly consume disk space. A denial of service attack would be easy to achieve.
- As with other methods, a parser or filter program would be needed to extract the SQL, user and timing information and then to decide if the SQL was a SQL injection attempt.
- Because trace files are again generated based on OS PID, managing them would be tricky in real time to ensure that resources are not overused. Any long running sessions could easily fill a disk.

The information in trace files is good and usable but the problems with managing the trace files and the performance issues with generating trace constantly would suggest this method might not be usable. Again, as with SQL\*net these files can be used sparingly.

### Reading SQL from the System Global Area (SGA)

This should be the most promising method to extract SQL and analyse if a SQL injection attempt has been made. The reason is purely because this is the *heart* of the Oracle RDBMS, and all SQL and PL/SQL executed spends some time in the SQL area in the SGA. There are a couple of issues to be aware of with this, however. The first is that querying the SQL area can be resource intensive and could affect performance on a critical production system and secondly it could be possible to miss SQL that has been executed. If a database runs thousands of pieces of SQL and all are different and the memory allocated for the SQL area is not large, then little used SQL (once or twice) could be aged out of the area very quickly.

Querying the SQL regularly is the key to monitoring for abuse. Too often this could affect performance and not often enough you could miss something. If an organisation were to use this method to check for abuse, start with checks maybe two or three times a day, monitor it and adjust as more is learnt.

Once again, as with the other sources of information a filter or parser is really needed to analyse the SQL extracted to give some indication as to whether it is legal or not. Start with a basic script like the one below that just checks for the existence of a *union* in the SQL, filter out some users perhaps, save the results to a summary table using a *create table as select* statement and further filter for specific tables involved. A good first approximation would be to highlight any SQL issued by a non SYS user with a *union* that also accesses a table or view owned by sys. Our example queries the view *all\_users* owned by sys.

Here is a simple query that extracts the SQL from the SGA where there is a *union* included. This is to give the reader an idea of what data can be read. Further filtering can be done as described above.

```
select  a.address address,
        s.hash_value hash_value,
        s.piece piece,
        s.sql_text sql_text,
        u.username parsing_user_id,
        c.username parsing_schema_id
from    v$sqlarea a,
        v$sqltext_with_newlines s,
        dba_users u,
        dba_users c
where   a.address=s.address
and     a.hash_value=s.hash_value
and     a.parsing_user_id=u.user_id
and     a.parsing_schema_id=c.user_id
and     exists (select 'x'
               from v$sqltext_with_newlines x
               where x.address=a.address
               and x.hash_value=a.hash_value
               and upper(x.sql_text) like '%UNION%')
order  by 1,2,3
/
```

Running this gives:

<output snipped>

```
QL_TEXT                                PARSING_USER_ID PARSING_SCHEMA_
-----
```

```

and a.parsing_schema_id=c.user_id
and upper(s
.sql_text) like '%UNION%'          SYSTEM          SYSTEM
order by 1,2,3

BEGIN dbsnmp.get_cust('x' union    SYS            SYS
select username from all_users
where 'x'='x'); END;              SYS            SYS

select customer_phone from        SYS            SYS
customers where
customer_surname='x'
union select username from        SYS            SYS
all_users where 'x'='x'

BEGIN get_cust('x' union select    DBSNMP         DBSNMP
username from all_users where '
'x'='x'); END;                    DBSNMP         DBSNMP

select                                SYS            SYS
tc.type#,tc.intcol#,tc.position#,c.
type#, c.length,c.scal
<output snipped>

```

The example has been run as both the SYS user and DBSNMP user whilst this database has been up so it appears a number of times in the output. This method so far seems to be the simplest to use and offers probably the easiest way to implement a complete tool/script to analyse for SQL injection. It also has the least downside effects!

## Using Oracle audit

The features available with standard Oracle audit are rich and varied but there are a number of problems with using it to detect SQL injection.

- Audit doesn't work at Row level only at session or access level.
- The SQL that was used at run time cannot be captured.
- There is no easy way to detect the use of a *union* (our example) or any of the other cases defined above that are only recognisable by parsing the SQL text.

Irrespective of any of the other methods, audit should be used. It is a robust and useful system and can reap benefits when used to detect abuse.

It is not easily possible to detect SQL injection with audit but rather to see the *smoking gun*. For instance if audit were enabled for *select access* on the tables `dbsnmp.customers` and `sys.all_users` then the audit trail should show entries when the SQL injection attempt as described in our example is used. We cannot definitely say a *union* was employed but access by DBSNMP to ALL\_USERS should be visible.

When a hacker is attempting to guess what SQL he can add to an existing string, how to add a union or sub-select, or adding comments to truncate, SQL errors can be generated when he gets it wrong. Again it is a *smoking gun* and detailed analysis of errors in the audit trail and other actions by the same user in the same session can indicate what was happening.

Here is an example of using audit whilst the SQL injection example is run. First turn on audit on the ALL\_USERS view and CUSTOMERS table.

```
SQL> audit select on sys.all_users by access;
```

```
Audit succeeded.
```

```
SQL> audit select on dbsnmp.customers by access;
```

```
Audit succeeded.
```

Check the audit trail with the following SQL:

```
col username for a8
col timestamp for a20
col action_name for a15
col obj_name for a10
col owner for a8
col sessionid for 999
col returncode for 999
select  username,
        to_char(timestamp,'DD-MON-YYYY HH24:MI:SS') timestamp,
        action_name,
        obj_name,
        owner,
        sessionid,
        returncode
from    dba_audit_object
order  by timestamp;
```

The results are:

USERNAME	TIMESTAMP	ACTION_NAME	OBJ_NAME	OWNER	SESSIONID	RETURNCODE
DBSNMP	16-JUN-2003 20:21:08	SELECT	ALL_USERS	SYS	227	0
DBSNMP	16-JUN-2003 20:21:08	SELECT	CUSTOMERS	DBSNMP	227	0
DBSNMP	16-JUN-2003 20:21:08	SELECT	CUSTOMERS	DBSNMP	227	0

This shows access to the ALL\_USERS view and CUSTOMERS table by the user DBSNMP in the same session. It is simple also to write a piece of SQL to analyse errors in the audit trail.

Audit can clearly be used to assist in detecting SQL injection but would require considerable effort to define "good" SQL queries and audit settings to reliably "guess" at SQL injection attempts.

For more information see the [previous article](#) by this author about Oracle audit for a brief introduction to its use.

## Database triggers

Database triggers are usually the next line of defence when Oracle's internal audit is used. The normal audit facilities operate at *object* level or privilege level and are not useful for determining what happened at the row level. Triggers can be used for this. To use them involves programming a trigger for each table and for each action such as *insert*, *update* or *delete*. The main failing with triggers is that they cannot be written to fire when a *select* takes place against a table or view.

Extracting the actual SQL used to do the *update*, *delete* or *insert* is not possible. It is possible to create SQL that is used to alter the table the trigger fires on by reading before and after values for each row and creating the trigger to fire for each row.

In view of these restrictions, database triggers are not really useful in the quest to find SQL injection attempts.

## Fine grained auditing

With version 9i, Oracle introduced *fine grained auditing* (FGA). This functionality is based on internal triggers that fire every time an SQL statement is parsed. As with fine grained access control it is based on defining predicates to limit the SQL that will be audited. By using audit policies, auditing can be focused on a small subset of activities performed against a defined set of data.

The one advantage that this functionality brings is the ability to finally monitor select statements at the row level. The standard handler function also captures SQL, so this looks like it could be a good tool to check for SQL injection.

Indeed Oracle states in their on-line documentation for this package that it is an ideal tool to implement an IDS system with. However it remains to be seen as to whether anyone has successfully used this package and policies as a base for an IDS tool.

Let's set up a simple policy and see if our example injection can be caught. The first requirement is that the tables with FGA set up against them need to have statistics generated by analysing. Also the cost-based optimiser must be used. There are a number of issues with FGA that cause the trigger to not fire correctly, for instance if a RULE hint is used in the SQL.

First check the optimiser and analyse the sample table:

```
SQL> analyze table dbsnmp.customers compute statistics;
```

Table analyzed.

```
SQL> select name,value from v$parameter
       2  where name='optimizer_mode' ;
```

NAME

-----  
VALUE  
-----

optimizer\_mode

CHOOSE

Next, create a policy that will execute every time a statement is run against the table (in this case DBSNMP.CUSTOMERS).

```
1  begin
2  dbms_fga.add_policy(object_schema=>'DBSNMP' ,
3                      object_name=>'CUSTOMERS' ,
4                      policy_name=>'SQL_INJECT' ,
5                      audit_condition=>'1=1' ,
6                      audit_column=>null ,
7                      handler_schema=>null ,
8                      handler_module=>null ,
9                      enable=>true);
10* end;
SQL> /
```

PL/SQL procedure successfully completed.

SQL>

Next, execute the SQL injection example and then check the audit trail for any entries:

```
SQL> set serveroutput on size 100000
SQL> exec get_cust('x' union select username from all_users where 'x'='x');
debug:select customer_phone from customers where customer_surname='x' union
select username from all_users where 'x'='x'
::AURORA$JIS$UTILITY$
::AURORA$ORB$UNAUTHENTICATED
::CTXSYS
::DBSNMP
::EMIL
<records snipped>
::SYS
::SYSTEM
::WKSYS
::ZULIA
```

PL/SQL procedure successfully completed.

```
SQL> col db_user for a15
SQL> col object_name for a20
SQL> col sql_text for a30 word_wrapped
SQL> select db_user,timestamp,object_name,sql_text
       2  from dba_fga_audit_trail
       3  order by timestamp;
```

DB_USER	TIMESTAMP	OBJECT_NAME	SQL_TEXT
DBSNMP	16-JUN-03	CUSTOMERS	select customer_phone from customers where customer_surname='x' union select username from all_users where 'x'='x'

SQL>

Okay, success. The dynamic SQL is shown including the *union* join to the ALL\_USERS system table. The use of FGA would need to be expanded to every table in the application schema. Then reports would need to be written that returned only entries that violate some of the basic rules we defined at the beginning, such as SQL with a *union* in it or SQL with a line such as 'x'='x'.

All of the policies could be easily generated using SQL such as:

```
select 'exec dbms_fga.add_policy(object_schema=>'||owner||',object_name=>'||
||table_name||',policy_name=>'||table_name||',audit_condition=>'1=1',aud
it_column=>null,handler_schema=>null,handler_module=>null,enable=>true);'
from dba_tables
where owner not in ('SYS','SYSTEM')
/
```

Of course the policies would probably need better names to avoid name clashes and it would be prudent to include some logic in a handler function to analyse the SQL for abuses.

## Protection is better than detection

Some solutions for protecting against SQL injection were given in the previous papers but for completeness a few of the main ideas are included here again:

- Do not use dynamic SQL that uses concatenation. If it is absolutely necessary then filter the input carefully.
- If possible do not use dynamic PL/SQL anywhere in an application. Find another solution. If dynamic PL/SQL is necessary then use *bind* variables.
- Use AUTHID CURRENT\_USER in PL/SQL so that it runs as the current user and not the owner.
- Use *least privilege principle* and allow only the privileges necessary

## Conclusions

SQL injection is a relatively new phenomenon and is being embraced by attackers with gusto. No figures are yet available to quantify how big the problem is for Oracle-based systems. At present more exposure in the press, technical sites and publications is given to other database products. I believe this is because it is slightly harder to SQL inject an Oracle database than other products. However, that doesn't mean there is not a problem with Oracle databases.

The other reason there are no accurate figures is that most companies probably would not even know anyone is using SQL injection against their Oracle database. I hope that this paper has given some insight to the issue and some simple ideas for DBAs and security managers to monitor for this problem.

As I have stated before, one simple solution is to not connect your Oracle database to the Internet (or intranet) if it is not necessary. Secondly, do not use dynamic SQL or PL/SQL and if you do, use bind variables. Also audit and secure the data with as much thought as possible to the OS and network security.

The simple test cases above have shown that there are indeed a number of trails left in the database or

network trace files when SQL injection is attempted. Therefore it should be possible for the DBA to use some of the above sources as a basis for a detection policy. Some of the methods such as trace files are clearly resource hogs. The fine grained audit and reading SQL from the SGA look like good candidates, as does the consistent use of audit.

Of course, the best form of defence is to audit your database security and avoid dynamic SQL!!

## References

- Oracle security step-by-step "A survival guide for Oracle security" - Pete Finnigan 2003, published by SANS Institute
- Oracle security handbook - Aaron Newman and Marlene Theriault - published by Oracle Press.
- Expert One-on-one - Tom Kyte - wrox press
- <http://www.orafaq.com/papers/redolog.pdf>
- [www.petefinnigan.com/orasec.htm](http://www.petefinnigan.com/orasec.htm)
- Oracle Net8 configuration and troubleshooting - Toledo and Gennick , O'Reilly
- Oracle in a nutshell - Greenwald and Kreines - O'Reilly
- [Introduction to simple Oracle auditing](#)
- ["SQL injection and Oracle - part one"](#)
- ["SQL injection and Oracle - part two"](#)
- <http://www.interealm.com/technotes/robby/fga.htm>

## About the author

Pete Finnigan is the author of the recently published book "Oracle security step-by-step - A survival guide to Oracle security" published in January 2003 by the [SANS Institute](#). Pete Finnigan is the founder and CTO of [PeteFinnigan.com Limited](#) a UK based company that specialises in auditing and securing Oracle databases for clients world wide.

## Search SecurityFocus

Show all [SecurityFocus articles by Pete Finnigan](#).

[Privacy Statement](#)

Copyright 2006, SecurityFocus