Oracle Forensics Part 5:
Finding Evidence of Data Theft
in the
Absence of Auditing

David Litchfield [davidl@ngssoftware.com]
10th August 2007

An NGSSoftware Insight Security Research (NISR) Publication

http://www.ngssoftware.com

Introduction

The forensic analysis of a compromised database server presents its own unique challenges. In other areas of computer forensics it's often obvious that a crime has been committed: pornographic images are discovered on a hard drive; a rootkit has been installed; a system has been trashed. In the case of a database intrusion however it may appear at first glance that nothing untoward has happened - prima facie evidence appears absent. In the physical world if something is stolen it is gone and by that the theft becomes obvious but with computers, and specifically database servers, when data is stolen, only a copy is taken and the original remains. As such, it is not immediately apparent that a theft has occurred and, in the absence of a suitable audit trail, it becomes even harder for the investigator to determine whether a breach has occurred. According to the National Conference of State Legislatures [1] 35 states in the U.S. have enacted security breach notification laws such as the California Database Security Breach Notification Act, Senate Bill 1386. Many of these laws only require notification in the event of an attacker gaining access to personally identifiable information (PII) and some require organizations to only notify in the case of an attacker gaining access to unencrypted PII. Knowing whether an attacker has gained access or not is critical when it comes to making the decision as to notify or not. This paper will show how an incident responder may determine if a such a breach of an Oracle database server has occurred in the event that there is no audit trail but it is suspected that an attacker has gain unauthorized SELECT access to data.

If an attacker breaks into a database and creates objects such as functions or tables and even deletes them afterwards in an attempt to hide their activities then they can be easily spotted [2]. If an attacker however breaks in and simply silently SELECTs some data, for example usernames, passwords or credit card details, and then slinks away then they can be much more difficult to spot. In a system that doesn't have auditing enabled how can one tell if such an attack has taken place? Pinning down evidence of this kind of compromise can be difficult but, as this paper will show, there are a few places where evidence of SELECT queries can be found. These include tables used by the Automatic Workload Repository (AWR), the Cost-Based Optimizer (CBO) and fixed views in the shared pool area of memory. In the case of SQL injection attacks through a web server there may of course be evidence in the web server's log files – this paper however concentrates on evidence in the database server itself.

N.B. This paper details information about Oracle 10g Release 2 only and should be used as a guideline for investigating other versions of Oracle as no guarantees or assertions can be made about other versions. For more papers on Oracle Forensics please see http://www.databasesecurity.com/oracle-forensics.htm

The Cost-Based Optimizer

Whenever a user executes an SQL query, the server needs to compile the query into an execution plan. The best way to do this is determined by the Cost-Based Optimizer (CBO) which attempts to reduce the amount of system resources required to service the

query. Statistics about the CBO are recorded in tables by the System Monitor (SMON) background process. One such table, the COL_USAGE$ table, is used to record information about predicates used in SELECT queries or, in other words, the columns used in a WHERE clause and the type of predicate such as equals, like, range and so on. In 10g Release 2, this table is updated by the SMON process every twenty minutes. Information in this table can be used by a forensic examiner or incident responder to infer details about SELECT queries that have been executed in the database which can indicate whether data may have been stolen or not. Before showing how, let's look at the table's definition.

```
SQL> DESC COL_USAGE$
 Name                                      Null?    Type
 ----------------------------------------- -------- ---------------
 OBJ#                                               NUMBER
 INTCOL#                                            NUMBER
 EQUALITY_PREDS                                     NUMBER
 EQUIJOIN_PREDS                                     NUMBER
 NONEQUIJOIN_PREDS                                  NUMBER
 RANGE_PREDS                                        NUMBER
 LIKE_PREDS                                         NUMBER
 NULL_PREDS                                         NUMBER
 TIMESTAMP                                          DATE
```

The OBJ# column holds the object ID of the table being queried and the INTCOL# column holds the column number, as taken from the COL# column in the COL$ table, used with the predicate. So, for example, if Z is defined as the 3rd column in COL$ on the COLTEST table and a query of '`SELECT * FROM COLTEST WHERE Z = 0`' is executed then the INTCOL# column in COL_USAGE$ will be 3. The TIMESTAMP column records to the nearest second when the entry was added to the COL_USAGE$ table – not when the query being recorded was executed. This is important to note when building a time line of events. As the SMON process writes changes to the COL_USAGE$ table every 20 minutes there can be a maximum skew of 20 minutes for a given entry. Another point to note with the TIMESTAMP column is that, if a new query on the same table using the same predicate is executed, the TIMESTAMP is updated; thus older records will be overwritten. The remaining columns indicate the type of predicate. For example, a row entry for EQUALITY_PREDS would be generated after a query of '`SELECT X FROM COLTEST WHERE Z = 5`'. A RANGE_PREDS row would be generated after a query of '`SELECT X FROM COLTEST WHERE Z > 0 AND Z < 100`'. A NULL_PREDS row would be generated after a query of '`SELECT Y FROM COLTEST WHERE Y IS NULL`' and a LIKE_PREDS row generated after a query of '`SELECT Y FROM COLTEST WHERE Y LIKE '%A%'`'.

By dumping the contents of this table, a full picture of which tables have been selected from and which columns have been used with which predicates can be gained. So as to format the results more clearly each predicate type can be queried separately. For example, dumping the LIKE_PREDs can be done as follows:

```
SQL> ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY-MM-DD HH24:MI:SS';
```

```
Session altered.

SQL> SELECT C.TIMESTAMP, O.NAME, C.INTCOL#, C.LIKE_PREDS FROM
COL_USAGE$ C, OBJ$ O WHERE C.OBJ#=O.OBJ# AND C.LIKE_PREDS > 0;

TIMESTAMP           NAME           INTCOL# LIKE_PREDS
------------------- -------------- ------- ----------
2007-08-08 06:10:27 COL$           6          1
2007-08-09 18:06:55 OBJ$           4          2
…
…
```

If we look at the first row of data we can see that someone has SELECTed from the COL$ table using the 6th column with a "like" predicate. As it happens, this row was created in COL_USAGE$ after an attacker, whilst looking for interesting tables from which to dump data, executed the following SQL:

```
SQL> SELECT TABLE_NAME FROM DBA_TAB_COLS WHERE COLUMN_NAME LIKE
'%CREDITCARD%';
```

DBA_TAB_COLS is a view that maps onto the COL$ table – COLUMN_NAME in the view corresponds to the 6th column in the COL$ table – which is "NAME" – in other words the name of the column. One thing to note here is that the OBJ# in COL_USAGE$ relates not to the view's object ID but that of the underlying table.

Clearly for this method to be useful one needs to be able acquire a baseline with which current data can be compared against. This baseline can be established in a number of ways. If the COL_USAGE$ table happens to have been backed up then comparisons with can be made these, provided of course the backups were made during a time outside of the intrusion. If no backups of COL_USAGE$ are available then it may be possible through examinations of the organization's database applications and conversations with DBAs and developers to determine which tables should appear in the COL_USAGE$ table and under what conditions. For example, if the organization's applications only ever query tables 1, 2 and 3 but not 4 and 4 happens to appear in the COL_USAGE$ data then one may infer that this is outside the bounds of what is "normal", warranting further investigation. There are further limitations when it comes to inferring details of attacks using COL_USAGE$ data. It is evident that if an attacker does not predicate their query with a column from the table in question then no entry will be created in the COL_USAGE$ table. Thus if an attacker queries '`SELECT PASSWORD FROM SYS.USER$ WHERE NAME = 'SYS'`' then a row will be created however no row will be created for the following query, '`SELECT PASSWORD FROM SYS.USER$`'.

If an attacker can run arbitrary SQL with DBA privileges, for example by exploiting a PL/SQL injection flaw, then the attacker could DELETE from this table. In doing so, however, evidence of this DELETE would be left in the redo logs as well as the data files themselves [2].

Fixed V$ views in the Shared Pool

There are a number of virtual tables and views that Oracle maintains for performance purposes. These views are accessible to DBAs and can often contain evidence of attacks. Two of these views are of particular interest – V$SQL and V$DB_OBJECT_CACHE.

The V$SQL fixed view contains a list of recently executed SQL. It is a circular buffer so as it fills up new information pushes out old information. Depending upon the size of the shared pool, and depending upon the length of each query the buffer can hold a large number of queries. A default install of 10g Release 2 can hold up to about 7000 queries before older entries are overwritten.

```
SQL> SET LONG 3000000
SQL> SELECT LAST_ACTIVE_TIME, PARSING_USER_ID, SQL_FULLTEXT FROM V$SQL;
```

Evidence of an attacker's activities may be found in this fixed view and careful examination of the SQL_TEXT should reveal this. It must be stressed that if an attacker can find a way to execute arbitrary SQL as DBA, of which there are many, then they can clear the SQL from this view by executing 'ALTER SYSTEM FLUSH SHARED_POOL'.

V$DB_OBJECT_CACHE contains details about objects in the library cache. There are two points of interest with regards to this particular view. Firstly, if an object exists in the cache then it has probably been accessed recently and secondly this view can contain snippets of recently executed SQL.

```
SQL> DESC V$DB_OBJECT_CACHE
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------
 OWNER                                              VARCHAR2(64)
 NAME                                               VARCHAR2(1000)
 DB_LINK                                            VARCHAR2(64)
 NAMESPACE                                          VARCHAR2(28)
 TYPE                                               VARCHAR2(28)
 SHARABLE_MEM                                       NUMBER
 LOADS                                              NUMBER
 EXECUTIONS                                         NUMBER
 LOCKS                                              NUMBER
 PINS                                               NUMBER
 KEPT                                               VARCHAR2(3)
 CHILD_LATCH                                        NUMBER
 INVALIDATIONS                                      NUMBER
```

The type of row data stored in the NAME column depends on the NAMESPACE column. If a row's NAMESPACE column is 'CURSOR' then NAME holds SQL data – if NAMESPACE is 'TABLE/PROCEDURE' then NAME holds a recently accessed table or procedure. Thus, to dump a list of recently executed queries one can execute

```
SQL> SELECT NAME FROM V$DB_OBJECT_CACHE WHERE NAMESPACE = 'CURSOR';
```

To access a list of recently accessed tables and procedures one can execute

```
SQL> SELECT OWNER, NAME FROM V$DB_OBJECT_CACHE WHERE NAMESPACE =
```

```
'TABLE/PROCEDURE' ORDER BY 1;
```

By examining this data an incident responder can determine if it contains evidence of an attack. This fixed view offers certain advantages over the V$SQL view. As already indicated, an attacker can clear the V$SQL view by executing 'ALTER SYSTEM FLUSH SHARED_POOL'. This is not true of data in the V$DB_OBJECT_CACHE view however – it is not cleared.

Automatic Workload Repository

The Automatic Workload Repository (AWR) is used to collect statistics related to performance. Records detailing information about SQL queries and object access are kept by the AWR and this can reveal what actions an attacker may have taken. AWR provides certain benefits over the V$ views as AWR data is persistent where as V$ data is lost when the database is shutdown and restarted but one of the drawbacks of using AWR data is that AWR needs to take a snapshot, which it does every second, at the same time that the attack is taking place. Assuming of course that the some of the attacker's actions have been captured by AWR then they can be found executing the following query:

```
SQL> SELECT ST.PARSING_SCHEMA_ID, TX.SQL_TEXT FROM WRH$_SQLSTAT ST,
WRH$_SQLTEXT TX WHERE TX.SQL_ID = ST.SQL_ID;
```

On locating "interesting" queries a timestamp can be collected by referencing the SQL_ID:

```
SQL> SELECT TIMESTAMP FROM WRH$_SQL_PLAN WHERE SQL_ID = 'b7v16a01s0f86'
```

It must be noted that on occasions some rows in the WRH$_SQLTEXT have no corresponding SQL_ID in the WRH$_SQLSTAT table. For example, on one of the test machines used in the lab for this document the following query produces 6 rows:

```
SQL> SELECT SQL_TEXT FROM WRH$_SQLTEXT WHERE SQL_ID NOT IN (SELECT
DISTINCT SQL_ID FROM WRH$_SQLSTAT);
```

Wrapping Up

This paper has shown where evidence of instances of data theft may be found in Oracle 10g Release 2 – specifically the tables maintained for performances purposes covering the Cost Based Optimizer, the dynamic performance V$ fixed views and the Automatic Workload Repository. In the absence of an audit trail these tables can help indicate whether an attack has occurred. In the opinion of the author it must be stressed that absence of evidence in these tables does not constitute proof that an attack did not take place – it can be seen that an attacker with the appropriate access can take steps to cover their tracks, even though in doing so they may leave evidence elsewhere.

[1] http://www.ncsl.org/programs/lis/cip/priv/breachlaws.htm
[2] http://www.databasesecurity.com/dbsec/Locating-Dropped-Objects.pdf