Oracle Forensics Part 2:
Locating dropped objects

David Litchfield [davidl@ngssoftware.com]
24th March 2007

An NGSSoftware Insight Security Research (NISR) Publication

http://www.ngssoftware.com

Introduction
After a successful compromise of a database server an attacker will usually attempt to hide their activities and this may include the dropping and purging of objects that they have created along the way, for example tables, functions and procedures. As this second paper in the Oracle Forensics series will show, even when an object has been dropped and purged from the system there will be, in the vast majority of cases, fragments left "lying around" which can be sewn together to build an accurate picture of what the actions the attacker took – or at least some of their actions. Perhaps, depending upon how quickly an investigation takes place from the incident in question, even all data pertaining to the dropped object or objects may still be found.

Oracle Data Blocks
Before beginning our search for dropped objects we should first discuss Oracle data blocks. Each data file is divided into blocks. The size of each block is determined by the database server's DB_BLOCK_SIZE initialization parameter. Each data file has a header and this block size can be found 20 bytes into this header; further the number of blocks in the data file can be found 24 bytes into the file. There are different kinds of blocks – some which store table data, others which store index data, cluster data and so on. Each block is sub-divided into different sections. The diagram represents a block and its sections.

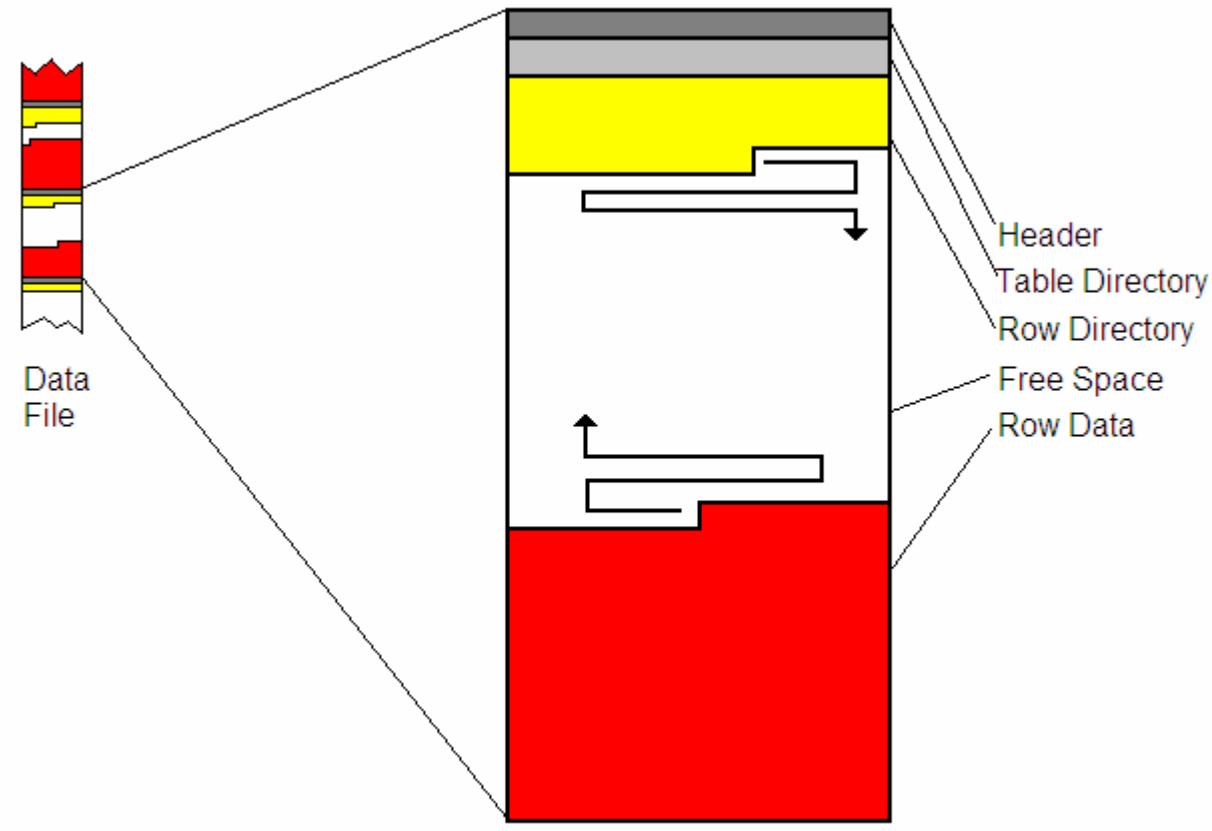

Figure 1 – An Oracle data block

Firstly there is the header which contains information such as, amongst other things, the block type, the Object ID of the table, index or cluster it's assigned to and a checksum. Below this is a table directory and below this a row directory. Both the

table and row directories are of variable size. The row directory contains information about how many rows of data are in the block and for each of these rows there is a two byte entry that acts as a pointer to the actual data of the row. This pointer is added to the offset into the file of the start of the block's row directory to give an offset to the location of the row of data. Row data itself is written to the block from the bottom up and eats into the free space from below. As new rows are added a new entry is added to the row directory and it eats into the free space from above. When the block is filled up, the server starts filling a new block. Each row in the block has a three byte header. The first byte is a marker and contains a set of flags to indicate the row's state. For example, if the row has been deleted the 5th bit of the byte is set. For example, a common set of flags value for a marker is 0x2C – which becomes 0x3C when the "deleted" flag is set. This is an important point to remember as it is a key indicator when looking for dropped objects. The second byte of the row header is used to determine lock status and the third byte indicates the total amount of data in the row. If the total amount is greater than 255 bytes then the row header is four bytes allowing for up to 65536 bytes. After the row header is the data itself. Each column of the row data is preceded with a byte indicating the size. If there is no data for a given column, in other words it is null, then it is represented with a 0xFF. As an example consider the following hex dump:

```
189d3790h:                                              3C 01 11
189d37a0h: 04 C3 06 13 2F 04 C3 06 13 2F 02 C1 37 0D 4D 59
189d37b0h: 5F 54 45 4D 50 5F 54 41 42 4C 45 02 C1 02 FF 02
189d37c0h: C1 03 07 78 6B 03 17 12 08 38 07 78 6B 03 17 12
189d37d0h: 08 38 07 78 6B 03 17 12 08 38 02 C1 02 FF FF 01
189d37e0h: 80 FF 02 C1 07 02 C1 02
```

Starting with the row header (3C 01 11) we can see that we're dealing with a deleted row as the 5th bit is set in 0x3C and we can see that the number of columns is 0x11 or 17 in decimal. Immediately after the row header we take the next byte as a length counter, count that many bytes which will take us to the next length counter and we do this until the number of columns, 17 in this case, has been reached. This breaks out the following:

```
Col 1       04 C3 06 13 2F
Col 2       04 C3 06 13 2F
Col 3       02 C1 37
Col 4       0D 4D 59 5F 54 45 4D 50 5F 54 41 42 4C 45
Col 5       02 C1 02
Col 6       FF
Col 7       02 C1 03
Col 8       07 78 6B 03 17 12 08 38
Col 9       07 78 6B 03 17 12 08 38
Col 10      07 78 6B 03 17 12 08 38
Col 11      02 C1 02
Col 12      FF
Col 13      FF
Col 14      01 80
Col 15      FF
Col 16      02 C1 07
Col 17      02 C1 02
```

What this data represents is not relevant right now but we will come back to it and break it down later on. When a row is deleted the space the row took up becomes available again as too does the entry in the row directory. However, until such time

that they are reused the information is still there and recoverable. For the purposes of this paper I'll call these deleted entries "linked" – because there is a link from the row directory entry to the deleted row itself. Linked deleted data is easy to find as we'll discuss. You may often find a row directory entry has been reused but not the space it used to point to – thus you can often find a deleted row without a corresponding entry in the row directory. For the purposes of this paper I'll call these "floating" – because there are not anchored to any entry in the row directory. These are a little harder to find than linked rows in terms of it requiring a little more effort.

Locating Deleted Rows

It's important to be able to spot a deleted row when looking for dropped objects because that's essentially what happens: when the object is dropped all row data pertaining to the object is deleted. Locating a deleted row is quite easy. For a given Oracle block find the row directory and for each entry workout the full offset of the row of data the entry points to. Using the first byte of the row header for each row, determine if the 5th bit is set – if so the row is deleted. [Even if the 5th bit is not set the row still may be deleted in the case of a deleted function in the OBJ$ table – which we'll deal with later – for now just put it down as an Oracle oddity!] For each row, visually "block out" the data. As per the details in the preceding section on Oracle Data Blocks you can do this by using the last byte of the row header (the number of columns in the row data) and cycling through until all data is accounted for. Once all linked rows are blocked out any data which is not blocked out is free floating – and is more than likely either deleted or left over from a previous UPDATE to the data. This should be repeated for each block in the data file that has been assigned to tables – or rather objects of interest.

Locating blocks assigned to "objects of interest"

There are a number of tables of interest when it comes to locating dropped objects – the OBJ$ table for example. There are indexes and clusters which we'll also be looking at. Regardless, locating blocks that are used by the table, index or cluster is the same. As stated earlier, each block has a header. This header contains the ID of object the block is assigned to at offset 24. Locating "objects of interest" is as easy as opening the data file and for each block checking the object ID. If it matches an object we're interested in then an analysis is performed on the block – if not then we move on to the next block.

Creating and Dropping Objects

It's important to understand what happens when an object is created as this will determine many of the locations where we will look later on. When an object is created a row is inserted in the OBJ$ table. This table has three indexes, I_OBJ1 to I_OBJ3, and an entry is also inserted into each of these indexes when the object is created. One of these, I_OBJ2, indexes the name of the object and the owner's ID. Depending on the object being created more rows are inserted into other tables and indexes. For example, when a table is created a row is inserted into the TAB$ table and one or more rows into the COL$ table. Underneath the covers both the TAB$ and COL$ tables exist in the C_OBJ# cluster and this is where the rows are actually created. The COL$ table has an index called I_COL1 which indexes the table's object ID and the table's column names so data about the table is inserted into this index as well. When a function or procedure is created, one or more rows are inserted into the SOURCE$, IDL_UB1$ and IDL_CHAR$ tables. Little bits of information about the

object are stored all across the database and it is these that will eventually provide the clues when attempting locate dropped objects. All of these tables, indexes and clusters are “items of interest” and should be thoroughly examined.

So that’s what happens when an object is created but what about when it is dropped? When an object is dropped the row in the OBJ$ is marked as deleted, by setting the row marker from 0x2C to 0x3C – or rather – the 5th bit of the byte is set. The data, such as the object name and ID etc, all still persist, just that it’s marked as deleted. The same goes for any of the other tables that have stored information about the object, such as TAB$ and COL$ in the C_OBJ# cluster in the case of a table. Likewise, information stored in the I_OBJ2 index is still “there” – it’s just marked as deleted. Further to this, if the server is running 10g or later and flash back is enabled, which it is by default, a row is created in the RECYCLEBIN$. If, at some later point, the recycle bin is purged or the object in question is purged from the recycle bin, then the row in the RECYCLEBIN$ is marked as deleted. Again, the data is still there, just the marker is set from 0x2C to 0x3C. With any of the rows that have been marked as deleted at some point the space may, of course, be re-used which has the side effect of destroying evidence. However, given that there are many locations where information about a given object is stored there is a good chance that not all deleted rows pertaining to the dropped object will have been re-used when it comes to investigating the compromised server.

Tracking down the evidence

Scenario: A DBA believes that one of his development servers has been compromised. No auditing was enabled. Is there any evidence to support a compromise occurred?

Locating dropped tables

Consider the hex dump of a block below. This is taken from the SYSTEM01.DBF file which is used by the SYSTEM tablespace. As we can see in the blue highlighted rectangle the block is assigned to object ID 0x12 – or 18 in decimal. This is the OBJ$ table. The hex dump cuts out the free space between 0x189D2080 and 0x189D3790. At 0x189D379D we have a row header marked as deleted – 0x3C.

```
189d2000h: 06 A2 00 00 E9 C4 40 00 2B 9B 0F 00 00 00 01 06 ; .¢..éÄ@.+›......
189d2010h: 8F 01 00 00 01 00 00 00 12 00 00 00 20 9B 0F 00 ; ▯........... ›..
189d2020h: 00 00 00 00 01 00 03 00 EA C4 40 00 01 00 24 00 ; ........êÄ@...$.
189d2030h: 08 02 00 00 0E 00 80 00 E6 01 11 00 01 20 00 00 ; ......€.æ.... ..
189d2040h: 2B 9B 0F 00 00 01 17 00 FF FF 40 00 31 15 E2 16 ; +›......ÿÿ@.1.â.
189d2050h: E2 16 00 00 17 00 42 1F D6 1E 67 1E F8 1D 95 1D ; â.....B.Ö.g.ø.•.
189d2060h: 2F 1D C9 1C 5E 1C F0 1B 82 1B 13 1B A1 1A 2F 1A ; /.É.^.ð.‚...¡./.
189d2070h: 31 15 1C 19 95 18 0D 17 A4 17 C1 16 75 16 25 16 ; 1...•...¤.Á.u.%.
189d2080h: C7 15 7C 15 00 00 00 00 00 00 00 00 00 00 00 00 ; Ç.|.............


189d3790h: C1 02 FF FF 01 80 FF 02 C1 07 02 C1 02 3C 01 11 ; Á.ÿÿ.€ÿ.Á..Á.<..
189d37a0h: 04 C3 06 13 2F 04 C3 06 13 2F 02 C1 37 0D 4D 59 ; .Ã../.Ã../.Á7.MY
189d37b0h: 5F 54 45 4D 50 5F 54 41 42 4C 45 02 C1 02 FF 02 ; _TEMP_TABLE.Á.ÿ.
189d37c0h: C1 03 07 78 6B 03 17 12 08 38 07 78 6B 03 17 12 ; Á..xk....8.xk...
189d37d0h: 08 38 07 78 6B 03 17 12 08 38 02 C1 02 FF FF 01 ; .8.xk....8.Á.ÿÿ.
189d37e0h: 80 FF 02 C1 07 02 C1 02 2C 00 11 04 C3 06 13 2A ; €ÿ.Á..Á.,...Ã..*
189d37f0h: 04 C3 06 13 2A 02 C1 37 1E 42 49 4E 24 4C 64 59 ; .Ã..*.Á7.BIN$LdY
```

Figure 2 – Hex Dump of Block assigned to OBJ$

When we map all the entries in the row directory and block out all the data for each entry we find that there are a couple of "gaps". These gaps are the "floating" deleted rows – in other words there is row data present but no entry for it in the row directory. The 0x3C at offset 0x189D379D is a floating deleted row. With the start of the row directory being at 0x189D2044 there should be an entry of 0x1759 but there is not – thus confirming it's a free floating entry. Although we can see the name in the ASCII on the right of the dump, MY_TEMP_TABLE, we need to extract the rest of the data.

Extracting the data
The row header is 3C 01 11. The last byte indicates there are 0x11 (17) columns.

```
189d3790h:                                              3C 01 11
189d37a0h: 04 C3 06 13 2F 04 C3 06 13 2F 02 C1 37 0D 4D 59
189d37b0h: 5F 54 45 4D 50 5F 54 41 42 4C 45 02 C1 02 FF 02
189d37c0h: C1 03 07 78 6B 03 17 12 08 38 07 78 6B 03 17 12
189d37d0h: 08 38 07 78 6B 03 17 12 08 38 02 C1 02 FF FF 01
189d37e0h: 80 FF 02 C1 07 02 C1 02
```

Recall that the column data is preceded with the length of the data. If there is no data for a given column, in other words it is null, then it is represented with a 0xFF. So, starting from the row header we take the next byte as a length counter, count that many bytes which will take us to the next length counter and we do this until the number of columns, 17 in this case, has been reached. This breaks out the following:

```
04 C3 06 13 2F
04 C3 06 13 2F
02 C1 37
0D 4D 59 5F 54 45 4D 50 5F 54 41 42 4C 45
02 C1 02
FF
02 C1 03
07 78 6B 03 17 12 08 38
07 78 6B 03 17 12 08 38
07 78 6B 03 17 12 08 38
02 C1 02
FF
FF
01 80
FF
02 C1 07
02 C1 02
```

To convert this into human understandable data we need to know the column types for the OBJ$ table:

```
OBJ#            NUMBER
DATAOBJ#        NUMBER
OWNER#          NUMBER
NAME            VARCHAR2(30)
NAMESPACE       NUMBER
SUBNAME         VARCHAR2(30)
TYPE#           NUMBER
CTIME           DATE
MTIME           DATE
```

```
STIME           DATE
STATUS          NUMBER
REMOTEOWNER     VARCHAR2(30)
LINKNAME        VARCHAR2(128)
FLAGS           NUMBER
OID$            RAW(16)
SPARE1          NUMBER
SPARE2          NUMBER
```

Knowing this we can begin to dump the data [see Appendix A]

```
04 C3 06 13 2F = ((6-1)*10000) + ((19-1)*100) + (47-1) = 51846
04 C3 06 13 2F = ((6-1)*10000) + ((19-1)*100) + (47-1) = 51846
02 C1 37 = 55
0D 4D 59 5F 54 45 4D 50 5F 54 41 42 4C 45 = MY_TEMP_TABLE
02 C1 02 = 1
FF = NULL
02 C1 03 = 2
07 78 6B 03 17 12 08 38 = 23/03/2007 17:07:37
07 78 6B 03 17 12 08 38 = 23/03/2007 17:07:37
07 78 6B 03 17 12 08 38 = 23/03/2007 17:07:37
02 C1 02 = 1
FF = NULL
FF = NULL
01 80 = 0
FF = NULL
02 C1 07 = 6
02 C1 02 = 1
```

We can see then that a table (2) called MY_TEMP_TABLE was created by the user with ID 55 at 17:07:37 on the 23rd March 2007. The table has an object ID of 51846. Searching for 04 C3 06 13 2F (the encoded object ID) elsewhere in the data file reveals that evidence of the table can also be found in the MON_MODS$ table, the I_MON_MODS$_OBJ and RECYCLEBIN$_OBJ indexes and the C_FILE#_BLOCK# cluster. Once we've located other items that have been dropped by the attacker we'll then locate the dropped table itself and see what, if anything it contains.

Locating dropped functions

Oddly enough when a function is dropped the row header in the block for the OBJ$ table is not modified – in other words the 5th bit of the marker byte is not set. Interestingly, however, the last two bytes of the STIME column are set to 0x3C. 0x3C is the same value for the vast majority of markers when a row has been deleted – the 5th bit is set. Why this is so I don't know – but regardless – it does provide an easy mechanism for spotting dropped functions. Let's assume though, to make things more interesting that, there is no evidence of a dropped function in any of the blocks assigned to the OBJ$ table. Does that mean that the attacker never created and dropped a function or does it mean that the evidence has been overwritten? By checking the SOURCE$ and IDL_UB1$ tables we can get closer to the truth.

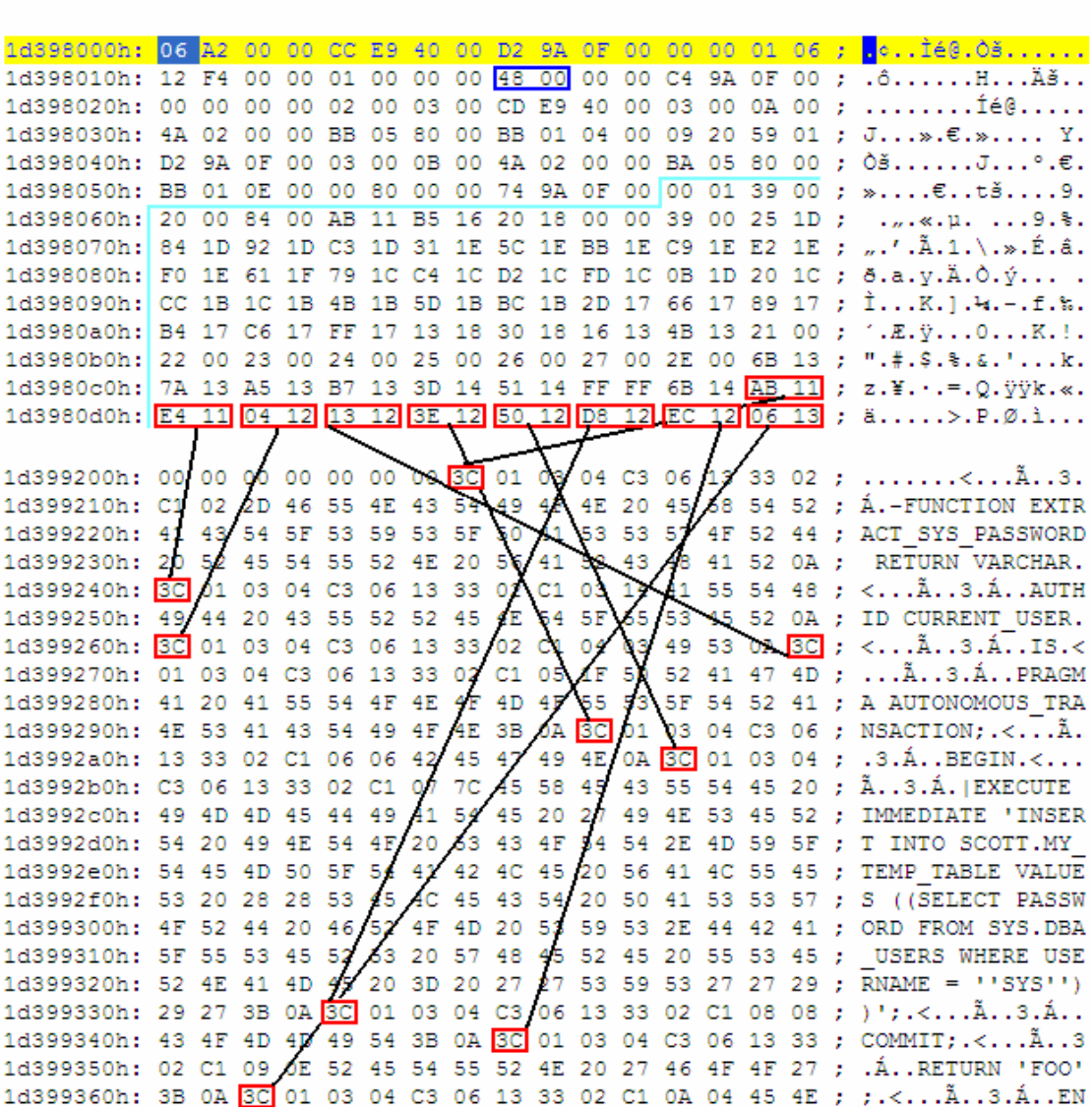

Figure 3 – Hex dump of block assigned to SOURCE$

In the hex dump above of a block assigned to the SOURCE$ table we find a large number of linked deleted rows:

0x1D39805C + 0x1306 = 0x1D399362
0x1D39805C + 0x12EC = 0x1D399348
0x1D39805C + 0x12D8 = 0x1D399334
0x1D39805C + 0x1250 = 0x1D3992AC
0x1D39805C + 0x123E = 0x1D39929A
0x1D39805C + 0x1213 = 0x1D39926F
0x1D39805C + 0x1204 = 0x1D399260
0x1D39805C + 0x11E4 = 0x1D399240
0x1D39805C + 0x11AB = 0x1D399207

Each of these relate to the same Object ID – C3 06 13 33 – or decoded – 51850. We can extract the text from the deleted data as well:

```
FUNCTION EXTRACT_SYS_PASSWORD RETURN VARCHAR
```

```
AUTHID CURRENT_USER
IS
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
EXECUTE IMMEDIATE 'INSERT INTO SCOTT.MY_TEMP_TABLE VALUES ((SELECT
PASSWORD FROM SYS.DBA_USERS WHERE USERNAME = ''SYS''))';
COMMIT;
RETURN 'FOO';
```

This looks pretty nefarious. This is the code of a function called EXTRACT_SYS_PASSWORD. Whilst we don't know who created the function – as there is no evidence in the blocks allocated to OBJ$ we can see that it inserts into SCOTT.MY_TEMP_TABLE table we noticed was dropped earlier. What's more, it selects the password hash for the SYS user and inserts it into this table? Was the attack successful – we'll just have to wait and see because we'll check the IDL_UB1$ table first. The IDL_UB1$ table has an object ID of 73. We dump all blocks that have been assigned to object ID 73 and soon come across the following:

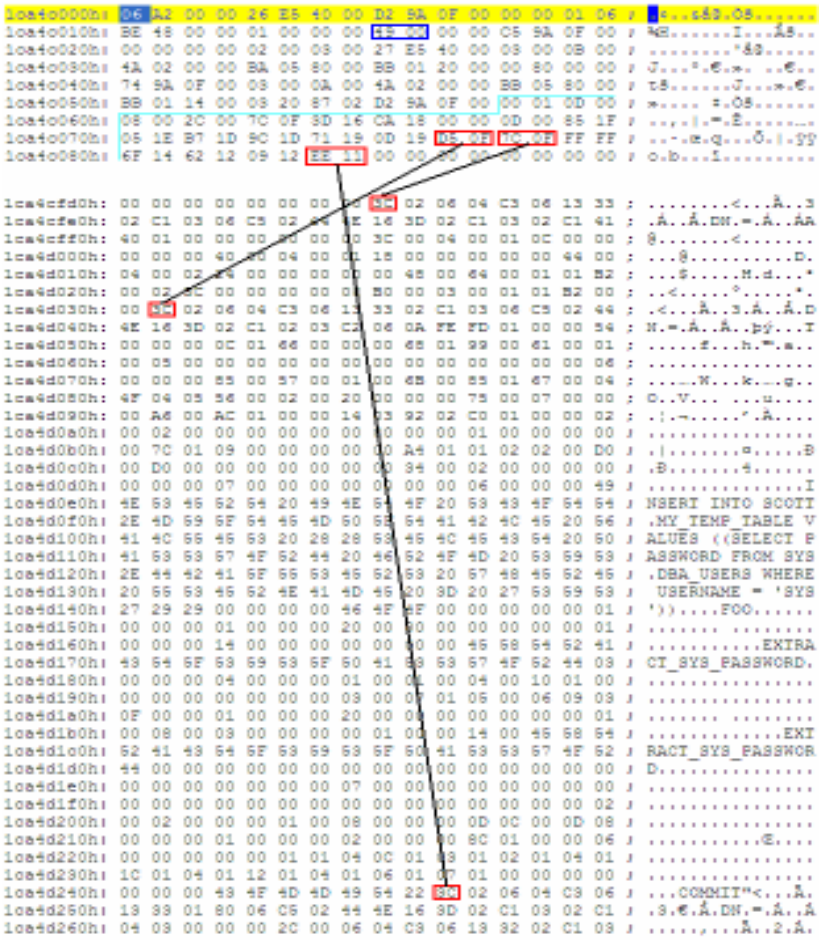

Figure 4 – Hex dump of block assigned to IDL_UB1$

Here we have some more linked deleted entries pertaining to the same object ID. We can also see large tracts of the text of the function – certainly the meat of it – the insertion of the password hash of the SYS user into the MY_TEMP_TABLE.

Locating the MY_TEMP_TABLE
From the data extracted from OBJ$ table we know that the object ID for the dropped table is 51846. We locate blocks for this table in the data files belonging to the USERS tablespace by looking for the Object ID 24 bytes into the block header. The hex dump below shows such a block and as the row directory is not empty there is data. We find this data at the end of the block and see that it is the hash for SYS password:

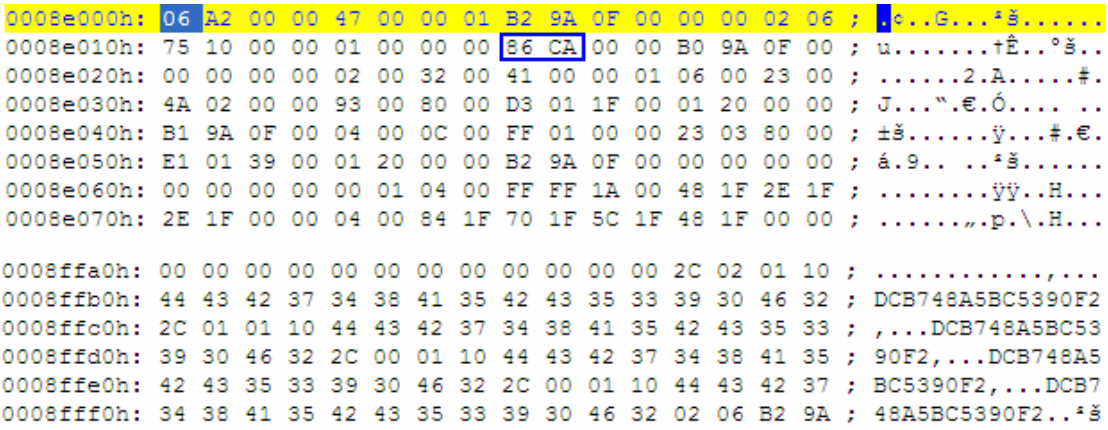

Figure 5 – Hex dump of block assigned to MY_TEMP_TABLE

So what can we say? We can say that at 17:07:37 on the 23rd of March 2007, the user SCOTT created a table called MY_TEMP_TABLE. We can also say that there was, at some point, a function called EXTRACT_SYS_PASSWORD that was coded to use this table to insert to SYS password hash. The table does have the SYS password hash in it and whilst it is extremely likely that the function was used to get the SYS password hash, we can't guarantee that it was used to get the hash. Other than the text of the function there is no direct link between the two. That said, there's a very strong likelihood that the EXTRACT_SYS_PASSWORD was created as an auxiliary inject function for a PL/SQL injection attack against a SYS owned package. Perhaps the redo logs [1] might show some more detail.

Anti-Forensics
A professional attacker is likely to leave the smallest of footprints and, as such, is unlikely to be creating and or dropping many objects. Indeed, by using the cursor injection attack method discussed in [2], an attacker doesn't need to create any objects at all. Given knowledge of how the block checksum is created and verified [1] it is not beyond the technical means for an attacker to modify the block on the fly using UTL_FILE or Java from within the database itself.

Conclusion
This paper has shown that despite the fact that an attacker may drop objects that they have used for ill purpose, a forensic examiner may still be able to recover evidence directly from the data files of the compromised server.

[1] Oracle Forensics Part 1 – Dissecting the Redo Logs, David Litchfield
http://www.databasesecurity.com/dbsec/oracle-forensics-pt1.pdf

[2] Cursor Injection, David Litchfield

http://www.databasesecurity.com/dbsec/cursor-injection.pdf