Oracle Forensics Part 6: Examining Undo Segments, Flashback and the Oracle Recycle Bin

David Litchfield [davidl@ngssoftware.com]
16th August 2007

An NGSSoftware Insight Security Research (NISR) Publication

http://www.ngssoftware.com

Introduction

This paper examines the ways in which a forensic examiner or incident responder may look for evidence in those places and technologies designed by Oracle for disaster recovery purposes – namely Undo segments, Flashback and the Recycle Bin - of a compromise and the actions an attacker may have taken. Please note that the research conducted for this paper was performed on Oracle 10g Release 2 and the information therefore should only be considered as pertaining to that version. This paper, however, can act as a suitable guideline for researching other versions of Oracle. For more papers on Oracle Forensics please see http://www.databasesecurity.com/oracle-forensics.htm

Flashback Queries

Oracle 10g introduced new technology called Flashback which allows a user to query data from an older version or snapshot of a given table – in other words peer back into the past and see the table's data as it was then. When responding to a potential incident this can prove invaluable to an incident responder. For example, to see if anyone has recently been granted role membership or new privileges one can select the difference between the current data and the data as existed at a set time ago – in this case 3600 minutes:

```
SQL> SELECT GRANTEE#, PRIVILEGE# FROM SYS.SYSAUTH$ MINUS SELECT
GRANTEE#, PRIVILEGE# FROM SYS.SYSAUTH$ AS OF TIMESTAMP(SYSDATE -
INTERVAL '3600' MINUTE);

  GRANTEE# PRIVILEGE#
---------- ----------
         1        -15
         1          4
```

Here we can see that, indeed, there is a difference between the older data and the current data. In this case DBA has been granted to PUBLIC. To see what objects have been dropped and which have been created in a given timeframe one could use the following queries. To find new objects that aren't in the older data execute:

```
SQL> SELECT NAME FROM SYS.OBJ$ MINUS SELECT NAME FROM SYS.OBJ$ AS OF
TIMESTAMP(SYSDATE - INTERVAL '156' MINUTE);

NAME
------------------------------
TESTTEST
```

To find old objects that are no longer in the data – in other words recently dropped execute:

```
SQL> SELECT NAME FROM SYS.OBJ$ AS OF TIMESTAMP(SYSDATE - INTERVAL '156'
MINUTE) MINUS SELECT NAME FROM SYS.OBJ$;

NAME
------------------------------
GET_DBA_FUNCTION
```

In “Oracle Forensics Part 5: Finding Evidence of Data Theft in the Absence of Auditing” [http://www.databasesecurity.com/dbsec/OracleForensicsPt5.pdf] the need to build a baseline of table accesses for comparison with data in the COL_USAGE$ table is required. This baseline can be achieved via a flashback query:

```
SQL> SELECT NAME FROM SYS.OBJ$ WHERE OBJ# IN (SELECT OBJ# FROM
SYS.COL_USAGE$ MINUS SELECT OBJ# FROM SYS.COL_USAGE$ AS OF
TIMESTAMP(SYSDATE - INTERVAL '500' MINUTE));
```

Data for flashback queries is drawn from the undo data and the redo logs and in practice may not be available for long. On a “quiet” system data may linger for a day or two but considerably less so in a “busy” system. Suffice it to say that, provided an incident responder or DBA gets there in “time” they will be able to quickly ascertain what an attacker may or may not have done. If no useful evidence can be gathered this way then they will have to dig a little deeper – as explained in the following sections.

The Oracle Recycle Bin

Whenever a table is dropped, the table and any dependent objects such as indexes and triggers are moved to the Recycle Bin. This way, if it is decided that the table has been dropped in error, it can be recovered from the Recycle Bin using the UNDROP statement. The Recycle Bin is implemented as a table called RECYCLEBIN$ in the SYSTEM tablespace.

When a table is dropped all that basically happens is that the name of the table is changed in SYS.OBJ$ from its original to its dropped equivalent – i.e. a prefix of BIN$ followed by what appears to be a base64 encoded string and affixed with a double equals, a dollar and a number e.g. “BIN$tjjNZzJ2RSWgPAOcVwnmQg==$0”. With the exception of the MTIME and CTIME columns everything else, such as OBJ# and so on remains the same. Along with the table name change, a row is inserted into the RECYCLEBIN$. This row details, amongst other items, the original table name, the object ID, the owner, and the time at which the table was dropped. If the object is not a table, but rather a trigger or index, then the TYPE# column indicates what the object is. [See Appendix B for a table showing the mapping from type number to type.]

The SQL below shows the relationship between a dropped object’s row data in SYS.OBJ$ and SYS.RECYCLEBIN$:

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

Session altered.

SQL> SELECT DROPTIME, OBJ#, OWNER#, ORIGINAL_NAME FROM SYS.RECYCLEBIN$;

DROPTIME              OBJ#     OWNER#  ORIGINAL_NAME
--------------------- -------- ------- ------------------------------
2007-08-16 09:27:45   53137    104     FOOBAR

SQL> SELECT MTIME, OBJ#, OWNER#, NAME FROM SYS.OBJ$ WHERE OBJ#=53137;
```

```
MTIME                 OBJ#     OWNER#  NAME
--------------------- -------- ------- ------------------------------
2007-08-16 09:27:46   53137    104     BIN$tjjNZzJ2RSWgPAOcVwnmQg==$0
```

If the user wishes to restore the table (and any associated objects) they may do so and everything is restored to its original state.

An incident responder should query both the RECYLEBIN$ and the OBJ$ table – specifically looking for objects starting with BIN$ in the latter case:

```
SQL> SELECT MTIME, NAME, OWNER#, OBJ# FROM SYS.OBJ$ WHERE NAME LIKE
'BIN$%';
```

If the Recycle Bin has been purged, a forensic examiner can often still gain access to the original data. Just because the table has been dropped doesn't mean it has been wiped from the data files. The diagram below shows a hex dump of the entry for the dropped table in the SYSTEM tablespace's data file. It shows a data block used by the OBJ$ table (object ID 18 or 0x12 and highlighted in red at the top. The grey square (0x00FA) is an entry in the row directory pointing to the deleted row in OBJ$. This entry is an offset from the start of the row directory at 0x189E0044 to where the row data can be found. As can be seen this row is marked as deleted – the 5th bit in the row flags (0x3C) has been set.

```
189e0000h: 06 A2 00 00 F0 C4 40 00 5C 7F 11 00 00 00 01 06 ; .¢..ðÄ@.\.......
189e0010h: 52 DF 00 00 01 00 00 00 12 00 00 00 5B 7F 11 00 ; Rß..........[...
189e0020h: 00 00 00 00 01 00 03 00 F1 C4 40 00 02 00 01 00 ; ........ñÄ@.....
189e0030h: 2D 02 00 00 1E 0B 80 00 6C 01 0E 00 01 20 5D 00 ; -.....€.l.... ].
189e0040h: 5C 7F 11 00 00 01 41 00 08 00 94 00 FA 00 81 04 ; \.....A...".ú.....
189e0050h: E0 04 00 00 41 00 49 1F E6 1E C7 04 1A 1E AF 1D ; à...A.I.æ.Ç...¯.
189e0060h: 78 03 D3 1C 64 1C FF FF 80 1B F1 19 87 19 1D 19 ; x.Ó.d.ÿÿ€.ñ.‡...
189e0070h: B9 18 4F 18 E5 17 7E 17 17 17 AE 16 42 16 D6 15 ; ¹.O.å.~...®.B.Ö.
189e0080h: 71 15 09 15 A1 14 40 14 DC 13 78 13 07 13 93 12 ; q...¡.@.Ü.x...".
189e0090h: 1F 12 B3 11 44 11 D5 10 68 10 F8 0F 88 0F 24 0F ; ..³.D.Õ.h.ø.ˆ.$.
189e00a0h: BE 0E 58 0E F4 0D 8D 0D 26 0D C0 0C 57 0C EE 0B ; ¾.X.ô.....&.À.W.î.
189e00b0h: 89 0B 21 0B B9 0A 50 0A E4 09 78 09 05 09 8F 08 ; ‰.!.¹.P.ä.x.....
189e00c0h: 19 08 AD 07 3E 07 CF 06 6C 06 06 06 A0 05 35 05 ; ..-.>.Ï.l... .5.
189e00d0h: 59 04 EA 03 06 03 FA 00 00 00 00 00 00 00 00 00 ; Y.ê...ú.........
189e00e0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
189e00f0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
189e0100h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
189e0110h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
189e0120h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
189e0130h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 3C 01 ; ..............<.
189e0140h: 11 04 C3 06 20 26 04 C3 06 20 26 03 C2 02 05 1E ; ..Ã. &.Ã. &.Â...
189e0150h: 42 49 4E 24 74 6A 6A 4E 5A 7A 4A 32 52 53 57 67 ; BIN$tjjNZzJ2RSWg
189e0160h: 50 41 4F 63 56 77 6E 6D 51 67 3D 3D 24 30 02 C1 ; PAOcVwnmQg==$0.Á
189e0170h: 02 FF 02 C1 03 07 78 6B 08 10 0A 16 17 07 78 6B ; .ÿ.Á..xk......xk
189e0180h: 08 10 0A 1C 2F 07 78 6B 08 10 0A 1C 2F 02 C1 02 ; ..../.xk..../.Á.
189e0190h: FF FF 03 C2 02 1D FF 02 C1 07 02 C1 03 3C 01 11 ; ÿÿ.Â..ÿ.Á..Á.<..
```

Figure 1: Hex dump of OBJ$ block showing deleted entry

We can clearly see the name of the dropped table – "BIN$tjjNZzJ2RSWgPAOcVwnmQg==$0". From here it is possible to dump the data such as object ID and so on and work backwards to recover evidence. For more information about finding deleted data and dropped objects please see "Oracle Forensics Part 2: Locating Dropped Objects" [http://www.databasesecurity.com/dbsec/Locating-Dropped-Objects.pdf]

Automatic Undo Management

In Oracle the ability to rollback transactions is enabled by "undo". Whenever a transaction takes place, for example an INSERT, DELETE or UPDATE, an image of the data before is recorded in an undo segment in the undo tablespace. So in the case of an UPDATE, an entry is created in the undo segment detailing what the data was before the update took place. Only those columns which are updated are recorded. In the case of a DELETE, then a copy of the data that was deleted is stored as an entry in the undo segment. With an INSERT it's slightly different. As there was no data to begin with, an entry detailing the file number, row and slot is recorded in the undo segment. This is all the information necessary to undo or rollback the transaction.

Undo Segments

Oracle 9i introduced Automatic Undo Management where undo segments are managed by the system as well as the length of time to retain undo data. 10g Release 2 by default uses Automatic Undo Management. To do this an undo tablespace is maintained – this made up of one or more files – by default a single file called UNDOTBS01.DBF. This tablespace contains 10 undo segments. Recall that a segment is made up of one or more extents and each extent is made up of one or more data blocks. Data blocks in an extent are allocated in a contiguous block.

```
SQL> SELECT SEGMENT_NAME,HEADER_FILE,HEADER_BLOCK,EXTENTS,BLOCKS FROM
DBA_SEGMENTS WHERE SEGMENT_NAME LIKE '_SYSSMU%$';

SEGMENT_NAME    HEADER_FILE HEADER_BLOCK    EXTENTS     BLOCKS
--------------- ----------- ------------ ---------- ----------
_SYSSMU1$                 2            9          2         16
_SYSSMU2$                 2           25          3        144
_SYSSMU3$                 2           41          3        144
_SYSSMU4$                 2           57          4        272
_SYSSMU5$                 2           73          3         24
_SYSSMU6$                 2           89          3         24
_SYSSMU7$                 2          105          4         32
_SYSSMU8$                 2          121          5         40
_SYSSMU9$                 2          137          3        144
_SYSSMU10$                2          153          3        144

10 rows selected.
```

The Undo Header

Each segment has a header detailing the location of the extents that it is made up of. The

diagram below is a hex dump of an undo segment header. The header contains, amongst other things an extent map. This map contains a list of Database Block Addresses (DBAs) that point to the extents that make up the segment and their respective lengths – in other words the number of blocks that make up the extent.

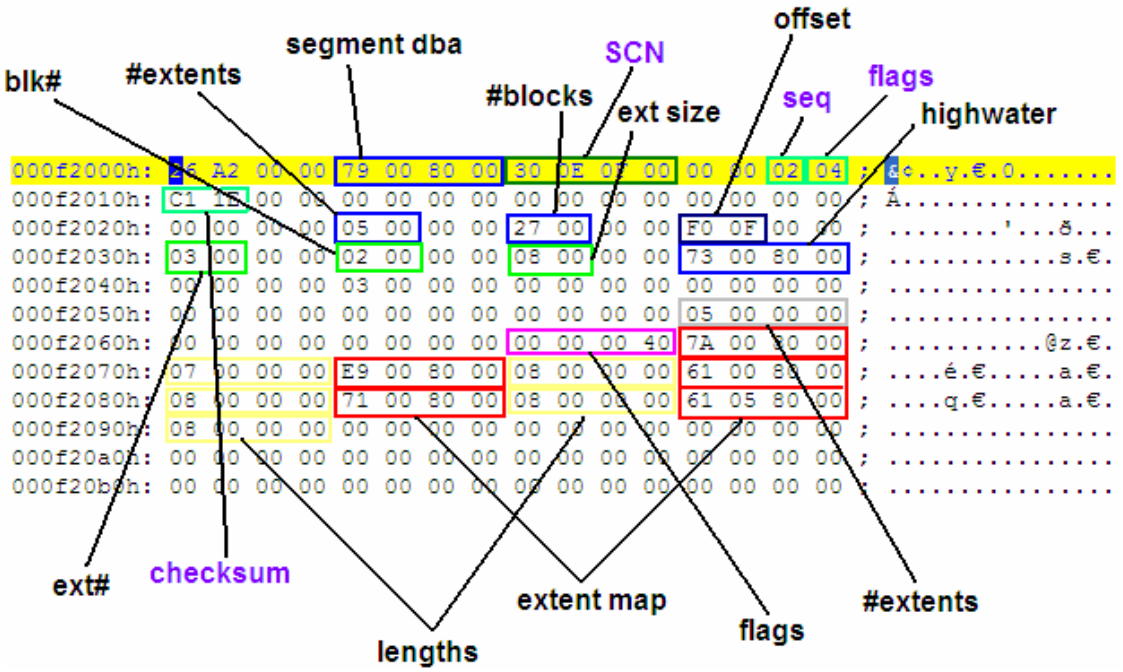

Figure 2: Hex Dump of Undo Segment Header

The first byte at 0x000F2000 is 0x26 which indicates that the block is a "KTU SMU HEADER BLOCK" – a System Managed Undo header. The header also contains a Retention Table 0x1000 bytes offset from the start. This table contains the time of the last commit for the extent. The time is stored in seconds since 1/1/1970 – Julian time. The diagram below shows a hex dump of a Retention Table.

```
000f3000h: 00 00 00 00 5D 7D C1 46 5D 7D C1 46 5D 7D C1 46 ; ....]}ÁF]}ÁF]}ÁF
000f3010h: 5D 7D C1 46 54 7D C1 46 00 00 00 00 00 00 00 00 ; ]}ÁFT}ÁF........
000f3020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000f3030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000f3040h: C5 6A AF 46 C5 6A AF 46 C5 6A AF 46 C5 6A AF 46 ; Åj¯FÅj¯FÅj¯FÅj¯F
000f3050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
```

Figure 3: Hex Dump of Retention Table

Undo Extents and Blocks

The extent map in the segment header contains entries that each point to the first block of a given extent. The number of blocks in the extent is indicated by the length entry in the extent map. Each block in the extent contains a header. The header contains a set of offsets (marked in green in the diagram below) that point to individual undo entries. The

offset is added to the block base plus twenty to locate the undo entry.

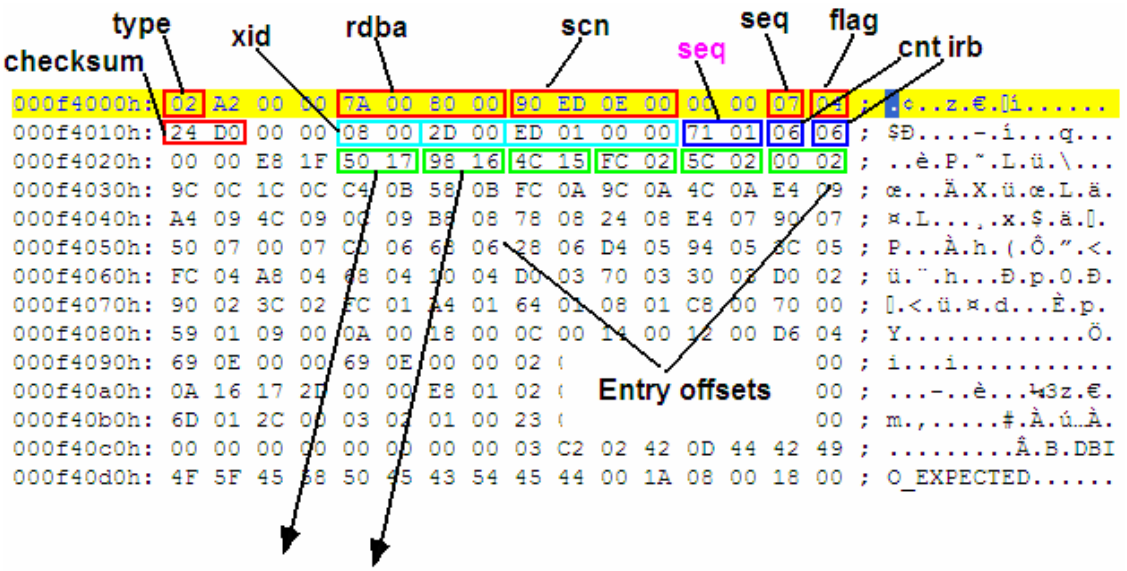

Diagram 4: Hex Dump of a Block Header

Undo Entries

Each undo entry has a header that indicates information about the entry such as the operation the undo entry is for. There are many different kinds of undo entries, denoted as Layers, though the most common actions relate to Row (10) and Index (11) operations. The undo entry header starts with a header size which is used to locate other items in the header in addition to a size located at bytes 3 and 4 specific to the entry type. Using these, the entry's layer, opcode, slot and the object ID the entry pertains to can be located. If the undo entry is for an Row operation then the opcode will be one of the following:

```
IUR [Interpret Undo Record]   1
IRP [Insert Row Piece]        2
DRP [Drop Row Piece]          3
LKR [Lock Row]                4
URP [Update Row Piece]        5
ORP [Overwrite Row Piece]     6
MFC [Manipulate First Column] 7
CFA [Change Forward Address]  8
CKI [Change Key index]        9
SKL [Set Key Links]           10
QMI [Quick Multi-Inserts]     11
QMD [Quick Multi-Deletes]     12
DSC                           14
LMN                           16
LLB                           17
SHK                           20
```

The C source code in Appendix A dumps the header all entries for each block in each segment. Below is a snippet of the output:

```
Number of extents: 17
Address: 0x0080000A Size: 0x00000007
Address: 0x00800081 Size: 0x00000008
Address: 0x00800061 Size: 0x00000008
Address: 0x00800071 Size: 0x00000008
Address: 0x008000B9 Size: 0x00000008
Address: 0x00800721 Size: 0x00000008
Address: 0x008000B1 Size: 0x00000008
Address: 0x008000D9 Size: 0x00000008
Address: 0x008000E1 Size: 0x00000008
Address: 0x00800109 Size: 0x00000008
Address: 0x00800119 Size: 0x00000008
Address: 0x00800131 Size: 0x00000008
Address: 0x00800D09 Size: 0x00000080
Address: 0x00800729 Size: 0x00000008
Address: 0x008000C1 Size: 0x00000008
Address: 0x008000D1 Size: 0x00000008
Address: 0x00800139 Size: 0x00000008

Dumping extents...
DBA: 0080000A
Number of blocks: 7
DBA: 0080000A
SCN: 1132624
XID: 0001.014.000001DF

Entries: 50 [32]

Rec #0x01: 1F44
Layer: 10 Index
opc: 21
rci: 0
ObjectID: 50529
Slot: 45 [2D]
Op: 5   ver: 1
…
…
Rec #0x17: 1568
Layer: 13 Transasction Segment
opc: 29
rci: 22
ObjectID: 50328
Slot: 22 [16]
Op: 235 ver: 103

Rec #0x18: 1508
Layer: 14 Transasction Extent
opc: 5
rci: 23
ObjectID: 1
Slot: 22 [16]
Op: 235 ver: 103

Rec #0x19: 142C
Layer: 11 Row
opc: 1
```

```
rci: 24
ObjectID: 14
Slot: 22 [16]
Op: 4   ver: 1
KDO Opcode: [05] URP [Update Row Piece]
```

Finding Evidence in the Undo Segments

Given that an attacker may perform actions that require a transaction, evidence of what the attacker did may appear in the undo segments. One can dump the undo tablespace using the alter system command:

```
SQL> SELECT FILE_ID, BLOCKS FROM DBA_DATA_FILES WHERE TABLESPACE_NAME =
'UNDOTBS1';

   FILE_ID     BLOCKS
---------- ----------
         2       4480

SQL> ALTER SYSTEM DUMP DATAFILE 2 BLOCK MIN 0 BLOCK MAX 4480;

System altered.
```

However, this, as its name indicates, alters the system and this could jeopardise evidence. Dumping an entire 36 Mb undo tablespace generates a 600 Mb file. Where possible an offline analysis should be performed. The C code presented in Appendix A can be used to help with this.

Conclusion

We have seen how technologies designed for disaster recovery can be used by a forensic examiner to locate evidence of an attack. As with all investigations if the closer in time to the event one can get when performing the investigation increases the likelihood of finding evidence. As we have seen on a busy system undo and flashback data may not be around for long but if you can find it, the evidence is telling.

Appendix A

```
#include <stdio.h>

int GetSegments();
int SetSegmentVariables();
int PrintExtentMap();
int DumpExtent(unsigned int address, unsigned int size);
int DumpBlock();
int DumpEntry(unsigned int rec);
int FindSMUHeaderBlock();
int SetupBuffer();
int CleanUp();
int ReadBlock();
unsigned int Peek();


unsigned int block_size = 0;
unsigned int number_of_extents = 0;
```

```
unsigned int number_of_blocks = 0;
FILE *fd = NULL;
unsigned char *block_buffer = NULL;
unsigned total_reads = 0;

int main(int argc, char *argv[])
{
      if(argc!=2)
      {
            printf("undoreader P.o.C.\n");
            printf("David Litchfield\n");
            printf("(david@databasesecurity.com)");
            printf("15th August 2007\n\n");
            return printf("c:\\>%s undo_file_name\n",argv[0]);
      }

      fd = fopen(argv[1],"rb");
      if(!fd)
            return printf("Failed to open %s\n",argv[1]);

      if(GetBlockSize()==0)
            return CleanUp();
      if(SetupBuffer()==0)
            return CleanUp();
      if(GetSegments()==0)
            return CleanUp();
      return CleanUp();
}

int GetSegments()
{

      while(1)
      {
            printf("************************************\n");
            if(FindSMUHeaderBlock()==0)
                  return 0;
            if(ReadBlock()==0)
                  return 0;
            // We now have SMU Header Block in block_buffer
            if(SetSegmentVariables()==0)
                  return 0;

      }
      return 1;

}

int SetSegmentVariables()
{
      //Dump();
      memmove(&number_of_extents,&block_buffer[92],4);
      printf("Number of extents: %d\n",number_of_extents);
      if(PrintExtentMap()==0)
            return 0;
      return 1;
}
```

```
int PrintExtentMap()
{
	unsigned int count = 0;
	unsigned int *address = NULL;
	unsigned int *size = NULL;
	unsigned int i = 0;
	unsigned int top = 0;
	unsigned int seccheck = 0;
	unsigned int *x = NULL;
	unsigned int *y = NULL;
	fpos_t pos;


	address = (unsigned int *)malloc(number_of_extents*4);
	if(!address)
		return 0;
	memset(address,0,number_of_extents*4);

	size = (unsigned int *)malloc(number_of_extents*4);
	if(!size)
	{
		free(address);
		return 0;
	}
	memset(size,0,number_of_extents*4);
	top = number_of_extents * 2;

	// Make sure we don't read beyond
	// bounds of buffer
	seccheck = (block_size - 108) / 4;
	if(top > seccheck)
	{
		free(address);
		free(size);
		return 0;
	}

	while(count < top)
	{
		memmove(address+i,&block_buffer[108+count*4],4);
		x = address+i;
		printf("Address: 0x%.8X ",*x);
		count++;
		memmove(size+i,&block_buffer[108+count*4],4);
		y = size+i;
		i++;
		printf("Size: 0x%.8X\n",*y);
		count ++;
	}

	count = 0;
	fgetpos(fd,&pos);
	printf("Dumping extents...\n");
	while(count < i)
	{
		x = address+count;
```

```
            y = size+count;
            DumpExtent(*x,*y);
            count ++;
      }
      fseek(fd,pos,SEEK_SET);

      free(address);
      free(size);


      return 1;
}

int DumpExtent(unsigned int address, unsigned int size)
{
      unsigned int res = 0;
      unsigned int addy=0;
      unsigned int count = 0;

      addy = address;
      addy = address ^ 0x00800000;
      addy = addy * block_size;

      printf("DBA: %.8X\n",address);
      printf("Number of blocks: %d\n",size);

      fseek(fd,addy,SEEK_SET);

      while(count < size)
      {
            if(ReadBlock()==0)
                  return 0;
            DumpBlock();
            count ++;
      }
      printf("\n");
      return 1;
}

int DumpBlock()
{
      unsigned int scn = 0;
      unsigned int dba = 0;
      unsigned short xidl = 0;
      unsigned short xidh = 0;
      unsigned int xid4 = 0;
      unsigned int entries = 0;
      unsigned int i = 0;
      unsigned int indx = 0;
      unsigned int head = 0;

      memmove(&dba, &block_buffer[4],4);
      memmove(&scn, &block_buffer[8],4);
      memmove(&xidl,&block_buffer[20],2);
      memmove(&xidh,&block_buffer[22],2);
      memmove(&xid4,&block_buffer[24],4);
      memmove(&entries,&block_buffer[30],1);
```

```
	printf("DBA: %.8X\n",dba);
	printf("SCN: %d\n",scn);
	printf("XID: %.4X.%.3X.%.8X\n",xidl,xidh,xid4);
	printf("Entries: %d [%.2X]\n",entries,entries);
	printf("\n");

	while(i < entries)
	{
		// Get offset to entry
		memmove(&indx,&block_buffer[36+i*2],2);

		printf("Rec #0x%.2X: %.4X",i+1,indx);
		DumpEntry(indx+20);
		i ++;
	}

	return 1;
}

int DumpEntry(unsigned int rec)
{
	unsigned int object_id = 0;
	unsigned int layer = 0;
	unsigned int slot = 0;
	unsigned int opcode = 0;
	unsigned int rci = 0;
	unsigned int sz = 0;
	unsigned int data_area_size = 0;
	unsigned int head = 0;
	unsigned int t = 0;
	unsigned int hs = 0;
	unsigned int op = 0;
	unsigned int ver = 0;

	printf("\n");
/*	Dump16Address(&block_buffer[rec]);
	Dump16Address(&block_buffer[rec+16]);
	Dump16Address(&block_buffer[rec+32]);
	Dump16Address(&block_buffer[rec+48]);
	Dump16Address(&block_buffer[rec+64]);
	Dump16Address(&block_buffer[rec+80]);
	Dump16Address(&block_buffer[rec+96]);
*/
	memmove(&sz,&block_buffer[rec],2);
	if(sz % 4 !=0)
		sz = sz + 2;
	memmove(&hs,&block_buffer[rec+2],2);
	memmove(&object_id,&block_buffer[rec+sz],4);
	memmove(&op,&block_buffer[rec+sz+hs],1);
	memmove(&ver,&block_buffer[rec+sz+hs+1],1);
	memmove(&layer,&block_buffer[rec+sz+16],1);
	memmove(&opcode,&block_buffer[rec+sz+17],1);
	memmove(&slot,&block_buffer[rec+sz+18],1);
	memmove(&rci,&block_buffer[rec+sz+19],1);
	memmove(&head,&block_buffer[rec+4],1);
```

```
printf("Layer: %d ",layer);
if (layer==5)
      printf("Transaction Undo\n");
else if (layer==10)
      printf("Index\n");
else if(layer == 11)
      printf("Row\n");
else if(layer == 13)
      printf("Transasction Segment\n");
else if(layer == 14)
      printf("Transasction Extent\n");
else if(layer == 22)
      printf("Tablespace Bitmapped file\n");
else
{
      printf("UNKNOWN Layer");
      getch();
}

printf("opc: %d\n",opcode);
printf("rci: %d\n",rci);

if(head == 0x00)
{
      //printf("Emtpy entry\n\n");
      printf("\n");
      return 1;
}


printf("ObjectID: %d\n",object_id);
printf("Slot: %d [%X]\n",slot,slot);
printf("Op: %d\tver: %d\n",op,ver);

// If not Transasction Segment

if(layer == 11)
{
      memmove(&t,&block_buffer[rec+sz+hs+head+10],1);
      printf("KDO Opcode: [%.2X] ",t);
      if(t==1)
            printf("IUR [Interpret Undo Record]\n");
      else if (t==2)
            printf("IRP [Insert Row Piece]\n");
      else if (t==3)
            printf("DRP [Drop Row Piece]\n");
      else if (t==4)
            printf("LKR [Lock Row]\n");
      else if (t==5)
            printf("URP [Update Row Piece]\n");
      else if (t==6)
            printf("ORP [Overwrite Row Piece]\n");
      else if (t==7)
            printf("MFC [Manipulate First Column]\n");
      else if (t==8)
```

```
                  printf("CFA [Change Forward Address]\n");
            else if (t==9)
                  printf("CKI [Change Key index]\n");
            else if (t==10)
                  printf("SKL [Set Key Links]\n");
            else if (t==11)
                  printf("QMI [Quick Multi-Inserts\n");
            else if (t==12)
                  printf("QMD [Quick Multi-Deletes]\n");
            else if (t==14)
                  printf("DSC\n");
            else if (t==16)
                  printf("LMN\n");
            else if (t==17)
                  printf("LLB\n");
            else if (t==20)
                  printf("SHK\n");
            else
                  printf("UNKNOWN\n");
      }

      if(layer == 5)
      {
            printf("5 [Transaction Undo]\n");
            printf("Opcode: %d ",opcode);

            if(opcode==3)
                  printf("[Rollout a transaction begin]\n");
            else if(opcode == 4)
                  printf("[Commit transaction/Transaction table update
- no undo record]");
            else if(opcode == 7)
                  printf("[Begin transaction/Transaction table
update]");
            else if(opcode == 8)
                  printf("[Mark transaction as dead]");
            else
                  printf("[Unknown]\n");

      }
      printf("\n");
      return 0;
}



int FindSMUHeaderBlock()
{
      unsigned int res = 0;
      while(1)
      {
            if(ReadBlock()==0)
                  return 0;
            res = Peek();
            if(res == 0)
                  return 0;
            res = res ^ 0x10000000;
```

```
            if(res == 0x26)
                  break;
      }
      return 1;
}
int SetupBuffer()
{
      if(block_size > 0x7FFFFFFE)
            return 0;
      block_buffer = (unsigned char *)malloc(block_size+1);
      if(!block_buffer)
            return 0;
      memset(block_buffer,0,block_size+1);
      return 1;
}

int CleanUp()
{
      fclose(fd);
      if(block_buffer)
            free(block_buffer);
      return 0;
}

int ReadBlock()
{
      int bytes_read = 0;
      memset(block_buffer,0,block_size);
      bytes_read = fread(block_buffer,1,block_size,fd);
      if(bytes_read = 0)
            return 0;
      if(feof(fd))
            return 0;
      total_reads ++;
      return 1;


}

unsigned int Peek()
{
      // Peek() returns 0x100000nn on success
      unsigned char buffer[8]="";
      unsigned int ret = 0x10000000;
      int bytes_read = 0;

      bytes_read = fread(buffer,1,4,fd);
      if(bytes_read = 0)
            return 0;
      fseek(fd,-4,SEEK_CUR);
      ret = ret + buffer[0];
      return ret;
}

int GetBlockSize()
{
      unsigned char buffer[200]="";
```

```
	int bytes_read = 0;
	bytes_read = fread(buffer,1,196,fd);
	if(bytes_read = 0)
		return 0;
	memmove(&block_size,&buffer[20],4);
	printf("Block Size = %d\n",block_size);
	fseek(fd,0,SEEK_SET);
	return 1;
}

int Dump()
{
	int count = 0;
	while(count < block_size)
	{
		printf("%.2X ",block_buffer[count]);
		count ++;
		if(count % 16 == 0)
			printf("\n");
	}
	return 0;
}

int Dump16()
{
	int count = 0;
	while(count < 16)
	{
		printf("%.2X ",block_buffer[count]);
		count ++;
	}
	printf("\n");
	return 0;
}

int Dump16Address(unsigned char *buf)
{
	int count = 0;
	while(count < 16)
	{
		printf("%.2X ",buf[count]);
		count ++;
	}
	printf("\n");
	return 0;
}
```

Appendix B

1 TABLE
2, 3 INDEX
4 NESTED TABLE
5 LOB

6	LOB INDEX
7	DOMAIN INDEX
8	IOT TOP INDEX
9	IOT OVERFLOW SEGMENT
10	IOT MAPPING TABLE
11	TRIGGER
12	CONSTRAINT
13	Table Partition
14	Table Composite Partition
15	Index Partition
16	Index Composite Partition
17	LOB Partition
18	LOB Composite Partition