



Waterford
Institute of
Technology

KDM INVESTIGATION

File System Forensics

ABSTRACT

Investigative report for recovery and analysis of a USB key, property of KDM Ltd.

[Samantha Sheehan](#)

Computer Forensics and Security

Contents

Introduction	3
Disclaimer - Maintaining Data Integrity	3
Initial Analysis	4
Directory Entries	4
I-node	4
Data Block	4
Block Group	5
Superblock	5
Group Descriptor Table	5
I-node Bitmap	5
Block Bitmap (Data Bitmap)	5
I-Node Table	5
Analysing the Data Structures	6
The Superblock	6
The Group Descriptor Table	8
The Block Bitmap	9
The I-node Bitmap	10
The I-node Table	10
Block pointers	11
Journal	12
Charting the Filesystem	14
The File System	14
The Block Groups	15
Block Group 0	15
Block Group 1	15
Block Group 2	15
Block Group 3	15
Block Group 4	15
Block Group 5	16
Block Group 6	16
Block Group 7	16
Exploring Files	17

Directories	17
Files	17
File Extraction	18
Deleted Files	19
Unknown Files	19
Investigation Conclusion	20
Bibliography	21

Introduction

As per the request from the HR department at KDM Ltd., a formal investigation was conducted regarding a USB key found at the desk of former employee, Anne O'Brien, suspected of theft of intellectual property.

The report contains a detailed description of the investigation conducted, the steps taken to recover the data and the files which were recovered.

I, Samantha Sheehan, conducted the investigation.

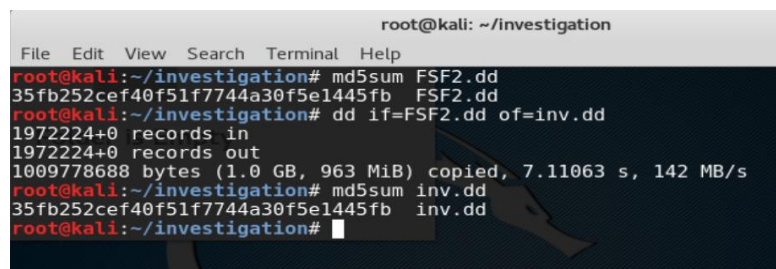
Disclaimer - Maintaining Data Integrity

Outlined below are the steps which were taken to ensure any actions taken by me during the course of the investigation did not edit or effect on any way the data recovered on the USB key.

Upon receipt of the evidence, I created an image of the file system on the USB key for analysis using a forensic imaging tool (ftk Imager). I then created a hash value using the same tool, which uses an algorithm called MD5sum to calculate a unique hexadecimal value for the image.

I then created a bit by bit duplicate image to analyse and verified the integrity of the copy by getting the MD5sum value for the duplicate and

comparing the two values. If the hash values are identical, this confirms that the images are exact duplicates, maintaining the integrity of the evidence.



```
root@kali: ~/investigation
File Edit View Search Terminal Help
root@kali:~/investigation# md5sum FSF2.dd
35fb252cef40f51f7744a30f5e1445fb FSF2.dd
root@kali:~/investigation# dd if=FSF2.dd of=inv.dd
1972224+0 records in
1972224+0 records out
1009778688 bytes (1.0 GB, 963 MiB) copied, 7.11063 s, 142 MB/s
root@kali:~/investigation# md5sum inv.dd
35fb252cef40f51f7744a30f5e1445fb inv.dd
root@kali:~/investigation#
```

It is important to note that this step was taken before the investigation began.

Initial Analysis

Using my duplicate copy (inv.dd), I used a forensic toolkit tsk (The Sleuth Kit) to investigate and recover information. I wanted to get an overview of the image, and some general information about the file system, so I used the tool fsstat to show me that information.

```
root@kali:~/investigation# fsstat inv.dd
FILE SYSTEM INFORMATION
-----
File System Type: Ext3
Volume Name: AOBrien USB Stic
Volume ID: adaf343235b6089228466b7c460dbc3f

Last Written at: 2012-04-17 16:01:44 (EDT)
Last Checked at: 2012-04-17 15:58:41 (EDT)
Last Mounted at: 2012-04-17 16:01:44 (EDT)
Unmounted properly
Last mounted on:

Source OS: Linux
Dynamic Structure
Compat Features: Journal, Ext Attributes, Resize Inode, Dir Index
InCompat Features: Filetype, Needs Recovery,
Read Only Compat Features: Sparse Super, Large File,

Journal ID: 00
Journal Inode: 8

METADATA INFORMATION
-----
Inode Range: 1 - 61697
Root Directory: 2
Free Inodes: 61685

CONTENT INFORMATION
-----
Block Range: 0 - 246527
Block Size: 4096
Free Blocks: 238239
```

This tool gave me general information such as the name of the USB key, AOBrien USB Stic, and also contained some important information such as the filesystem type: EXT3. The extended file system (EXT), was developed specifically for Linux in the nineties and has four variations: EXT, EXT2, EXT3 and EXT4. EXT3, or the third extended file system, is an extension of EXT2 with journaling, (technique used to prevent system corruption), as an additional feature.

Awareness of which type of file system is important when carrying out an investigation. There are many file systems and among them is a diversity in layout, protocols and data structures, for example while NTFS allocates data to clusters, addressed using cluster chains, EXT's addressing scheme uses structures known as i-nodes and data blocks, and where NTFS uses partitions, EXT uses block groups.

The fsstat commands also shows the general layout of the file system, and the data structures.

EXT systems use the following data structures:

Directory Entries

Directory entries are structures which store the file name, they also hold a pointer to the i-node of the file.

I-node

In an EXT system, each file is allocated an i-node. I-nodes store metadata information about the file (e.g. creation date, file type etc.), and usually stores a pointer to the memory location of the files content.

Data Block

A data block is the structure which stores a files content. It is made up of logical blocks of sequential memory space.

Block Group

A block group is just a group of blocks, the blocks are grouped together sequentially. Blocks and i-nodes are divided up and allocated to a block group. Each block group also holds information on the data contained within that block. Some blocks within a block group are reserved for the following purposes:

Superblock

The superblock contains information concerning the entire file system, there are redundant copies in multiple blocks. It keeps track of the structure and layout of the filesystem sometimes including information on features used in the file system(linfo.org, 2013).

Group Descriptor Table

The group descriptor table contains the location information on the important data structures of that block such as:

- Block address of i-node and data bitmap
- Starting block address of i-node table
- Number of unallocated blocks and i-nodes
- Number of directories

I-node Bitmap

The i-node bitmap is one block in size and contains binary used to record allocated or unallocated i-nodes in the block group. In the bitmap a zero would represent the corresponding i-node is unallocated, where a one would represent an allocated i-node (Layton, 2011).

```

BLOCK GROUP INFORMATION
-----
Number of Block Groups: 8
Inodes per group: 7712
Blocks per group: 32768
Music
Group: 0:
  Inode Range: 1 - 7712
  Block Range: 0 - 32767
  Layout:
    Super Block: 0 - 0
    Group Descriptor Table: 1 - 1
    Data bitmap: 62 - 62
    Inode bitmap: 63 - 63
    Inode Table: 64 - 545
    Data Blocks: 546 - 32767
  Free Inodes: 7692 (99%)
  Free Blocks: 31304 (95%)
  Total Directories: 2
    
```

Block Bitmap (Data Bitmap)

Like the i-node bitmap, the block bitmap stores allocation information about the data blocks.

I-Node Table

The i-node table is an array like structure which holds the i-nodes themselves (Carrier, File System Forensic Analysis).

To summarise the EXT3 systems physical structure usually looks like this:

Boot Sector	Block Group 0	Block Group 1	Block Group 2	Block Group ...	Block Group n
First 1024 bytes of block 0					

And inside each block group is structured like this:

Superblock	Group descriptor Table	Block Bitmap	I-node Bitmap	I-node Table	Data Blocks

Analysing the Data Structures

The Sleuth Kit is a well-used tool among forensics investigators, fsstat as well as other tools use the filesystem information to give an accurate report of the filesystem. However, it is important to know how The Sleuth Kit accesses and converts that information. I will work through the data structures of block 0 as an example and the data stored in the superblock, group descriptor table, the i-node and block bitmaps, and the i-node table showing their location in memory, how they are extracted, the data stored here and how to read the contents.

The Superblock

The first 1024 bytes of the EXT system are reserved for boot instructions, and the first copy of the superblock is located directly after this offset. It is also 1024 bytes in size. Keeping this in mind I used the dd tool of ftk (forensic imaging software) to create an image of the superblock, and printed a hexadecimal representation using the Linux command xxd.

First to explain the dd commands:

- if is the input file, the file to extract the data from (inv.dd).
- bs is the size in bytes of the block.
- skip is how many blocks of that size to skip before starting to store the information
- count is how many of those blocks to store.
- of is the output file, either creating a new file or use an existing file to store the data which has been extracted.

Here because the superblock is located 1024 bytes into the first block I set the bs to 1024 and skip 1 block.

```
root@kali:~/investigation# dd if=inv.dd bs=1024 skip=1 count=1 of=superblock.dd
1+0 records in
1+0 records out
1024 bytes (1.0 kB, 1.0 KiB) copied, 0.000355452 s, 2.9 MB/s
root@kali:~/investigation# xxd superblock.dd
00000000: 00f1 0000 00c3 0300 2630 0000 9fa2 0300  ....&0.....
00000010: f5f0 0000 0000 0000 0200 0000 0200 0000  ....
00000020: 0080 0000 0080 0000 201e 0000 28cc 8d4f  ....(..0
00000030: 28cc 8d4f 0100 2200 53ef 0100 0100 0000  (...0..".S.....
00000040: 71cb 8d4f 004e ed00 0000 0000 0100 0000  q..0.N.....
00000050: 0000 0000 0b00 0000 0001 0000 3c00 0000  .....<...
00000060: 0600 0000 0300 0000 3fbc 0d46 7c6b 4628  .....?..F|kF(
00000070: 9208 b635 3234 afad 414f 4272 6965 6e20  ...524..A0Brien
00000080: 5553 4220 5374 6963 0000 0000 0000 0000  USB Stic.....
00000090: 0000 0000 0000 0000 0000 0000 0000 0000  .....
000000a0: 0000 0000 0000 0000 0000 0000 0000 0000  .....
000000b0: 0000 0000 0000 0000 0000 0000 0000 0000  .....
000000c0: 0000 0000 0000 0000 0000 0000 0000 3c00  .....<.
000000d0: 0000 0000 0000 0000 0000 0000 0000 0000  .....
000000e0: 0800 0000 0000 0000 0000 0000 6580 d49b  .....e...
000000f0: cd5f 45be 8252 1117 b2fa 4f90 0101 0000  ._E..R....0....
00000100: 0000 0000 0000 0000 71cb 8d4f e401 0100  .....q..0....
00000110: e501 0100 e601 0100 e701 0100 e801 0100  .....
00000120: e901 0100 ea01 0100 eb01 0100 ec01 0100  .....
00000130: ed01 0100 ee01 0100 ef01 0100 f001 0100  .....
00000140: f105 0100 0000 0000 0000 0000 0000 0001  .....
00000150: 0000 0000 0000 0000 0000 0000 1c00 1c00  .....
00000160: 0100 0000 0000 0000 0000 0000 0000 0000  .....
00000170: 0000 0000 0000 0000 0000 0000 0000 0000  .....
00000180: 0000 0000 0000 0000 0000 0000 0000 0000  .....
```

The signature value (0xef53), located in bytes 56-57, is how I know that this is the super block.

The structure of the superblock is fixed, each byte is allocated a specific purpose:

Byte Range	Description	Essential		
0-3	Number of inodes in file system	Yes		
4-7	Number of blocks in file system	Yes		
8-11	Number of blocks reserved to prevent file system from filling up	No		
12-15	Number of unallocated blocks	No		
16-19	Number of unallocated inodes	No		
20-23	Block where block group 0 starts	Yes		
24-27	Block size (saved as the number of places to shift 1,024 to the left)	Yes		
28-31	Fragment size (saved as the number of bits to shift 1,024 to the left)	Yes		
32-35	Number of blocks in each block group	Yes		
36-39	Number of fragments in each block group	Yes		
40-43	Number of inodes in each block group	Yes		
44-47	Last mount time	No		
48-51	Last written time	No		
52-53	Current mount count	No		
54-55	Maximum mount count	No	92-95	Compatible feature flags (see Table 15.6)
56-57	Signature (0xfef53)	No	96-99	Incompatible feature flags (see Table 15.7)
58-59	File system state (see Table 15.2)	No	100-103	Read only feature flags (see Table 15.8)
60-61	Error handling method (see Table 15.3)	No	104-119	File system ID
62-63	Minor version	No	120-135	Volume name
64-67	Last consistency check time	No	136-199	Path where last mounted on
68-71	Interval between forced consistency checks	No	200-203	Algorithm usage bitmap
72-75	Creator OS (see Table 15.4)	No	204-204	Number of blocks to preallocate for files
76-79	Major version (see Table 15.5)	Yes	205-205	Number of blocks to preallocate for directories
80-81	UID that can use reserved blocks	No	206-207	Unused
82-83	GID that can use reserved blocks	No	208-223	Journal ID
84-87	First non-reserved inode in file system	No	224-227	Journal inode
88-89	Size of each inode structure	Yes	228-231	Journal device
90-91	Block group that this superblock is part of (if backup copy)	No	232-235	Head of orphan inode list
			236-1023	Unused

Taken directly from class notes created by John Sheppard, based on the book File System Forensic Analysis by Brian Carrier.

Following this guide, I can verify some important information such as:

	Byte Location	Hexadecimal Value	Conversion
Number of i-nodes in filesystem	0-3	00 f1 00 00	61,696
Number of Blocks in file system	4-7	00 c3 03 00	246,528
Number of unallocated blocks	12-15	9f a2 03 00	238,239
Number of unallocated i-nodes	16-19	f5 f0 00 00	4,085
Starting block of block group 0	20-23	00 00 00 00	0
Volume Name	120-135	41 4f 42 72 69 65 6e 20 55 53 42 20 53 74 69 63	AOBrien USB Stic
Journal i-node	224-227	08 00 00 00	8

The allocated bytes containing information sometimes must be converted differently, for example the volume name is converted using ascii characters. The other values shown in the table are stored in hex in little endian, this means the most significant bytes are located on the right-hand side, as opposed to the left in big endian. This simply means before converting we must reverse the order, byte by byte for example the value of the number of i-nodes in the system is stored in the superblock as 00f10000 which would convert to 15,794,176 in decimal, however if the bytes are reversed – 00 00 f1 00, the value becomes 61,696. Little endian is a common feature of addressing schemes, data structures or anywhere in computing where hex values are prevalent.

As is clear from the table on the previous page there is a lot more information to be gained from the superblock, the creator OS(Linux), and the version(EXT3) for example, the table above is just an example of the type of data stored in a superblock, and how this data is useful.

The Group Descriptor Table

The group descriptor table is located in the block superseding the superblock, for group zero this generally means the beginning of block 1, unless the block size is 1024 bytes then it's located at the beginning of block 2. The block size of our USB key is 4096 bytes as we have already seen from the fsstat tool.

CONTENT INFORMATION

```
-----
Block Range: 0 - 246527
Block Size: 4096
Free Blocks: 238239
```

Once again, we can use the dd tool from ftk (imaging software) and Linux command xxd to view the group descriptor table, however because the table is located at the beginning of the second block I change the bs to 4096, the actual size in bytes of each block, and again skip 1.

```
root@kali:~# cd investigation/
root@kali:~/investigation# dd if=inv.dd count=1 skip=1 bs=4096 of=groupdesc.dd
1+0 records in
1+0 records out
4096 bytes (4.1 kB, 4.0 KiB) copied, 0.0175563 s, 233 kB/s
root@kali:~/investigation# xxd groupdesc.dd
00000000: 3e00 0000 3f00 0000 4000 0000 487a 0c1e >...?...@...Hz..
00000010: 0200 0000 0000 0000 0000 0000 0000 0000 .....
00000020: 3e80 0000 3f80 0000 4080 0000 d97d 1b1e >...?...@....}..
00000030: 0300 0000 0000 0000 0000 0000 0000 0000 .....
00000040: 0000 0100 0100 0100 0200 0100 176e 201e .....n.
00000050: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000060: 3e80 0100 3f80 0100 4080 0100 de7d 201e >...?...@....} .
00000070: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000080: 0000 0200 0100 0200 0200 0200 1c7e 201e .....~.
00000090: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000a0: 3e80 0200 3f80 0200 4080 0200 de7d 201e >...?...@....} .
000000b0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000c0: 0000 0300 0100 0300 0200 0300 1c7e 201e .....~.
000000d0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000e0: 3e80 0300 3f80 0300 4080 0300 de40 201e >...?...@....@.
000000f0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000100: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000110: 0000 0000 0000 0000 0000 0000 0000 0000 .....
```

The group descriptor table holds 32 byte (2 rows on the table) entries on each block group, eight in total for this system.

The group descriptor table also has a fixed structure with bytes allocated for a specific purpose:

Byte Range	Description	Essential
0-3	Starting block address of block bitmap	Yes
4-7	Starting block address of inode bitmap	Yes
8-11	Starting block address of inode table	Yes
12-13	Number of unallocated blocks in group	No
14-15	Number of unallocated inodes in group	No
16-17	Number of directories in group	No
18-31	Unused	No

Taken directly from class notes created by John Sheppard, based on the book File System Forensic Analysis by Brian Carrier.

Using the information provided as a guide I can convert the data in the group descriptor table for block 0:

	Byte Location	Hexadecimal Value	Conversion
Starting block address of block bitmap	0-3	3e 00 00 00	62
Starting block address of i-node bitmap	4-7	3f 00 00 00	63
Starting block address of i-node table	8-11	40 00 00 00	64
Unallocated blocks in the group	12-13	48 7a	31,304
Unallocated i-nodes in the group	14-15	0c 1e	7,692
Directories in the group	16-17	02 00	2
Unused Space	18-31		

Again, the conversion is done using little endian. Each 2 rows in the descriptor table consists of one 32-byte entry representing one block group. Each block group could be analysed this way using the same table.

The Block Bitmap

The block bitmap is one byte in size. It is a string of binary which dictates the allocation status of a block, one means its allocated and 0 means unallocated. Each bit represents one block, the addressing based on the blocks address relative to the start of the block, for example block 2 is a third sequential block in block zero, so the third bit in the bitmap for block group 0 represents block 2. The group descriptor table has told me that the block bitmap is located in block 62 so I can again extract it using dd. I've used the binary function of xxd (-b) to print the bitmap in binary:

```
root@kali:~/investigation# dd if=inv.dd count=1 skip=62 bs=4096 of=blockbit.dd
1+0 records in
1+0 records out
4096 bytes (4.1 kB, 4.0 KiB) copied, 0.000360144 s, 11.4 MB/s
root@kali:~/investigation# xxd -b blockbit.dd
00000000: 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 .....
00000006: 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 .....
0000000c: 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 .....
00000012: 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 .....
00000018: 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 .....
0000001e: 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 .....
00000024: 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 .....
0000002a: 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 .....
00000030: 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 .....
00000036: 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 .....
0000003c: 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 .....
00000042: 11111111 11111111 11111111 00000000 00000000 00000000 00000000 00000000 .....
00000048: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 .....
0000004e: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 .....
00000054: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 .....
0000005a: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 .....
00000060: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 .....
```

As can be seen here the first 69 bytes are allocated blocks. 69 bytes is 552 bits, which means the first 552 blocks are allocated. These are not the only allocated blocks in the system, this is just to explain how the information is stored and accessed.

The I-node Bitmap.

The i-node bitmap work in exactly the same way as the block bitmap, it is 1 block in size and is a binary string representing the i-node allocation status. The group descriptor table told e the i-node table for group 0 was located in block 63, so I extracted the block and printed the binary. The first 10 i-nodes are reserved and so are always allocated. The superblock is usually the first unreserved i-node.

```
root@kali:~/investigation# dd if=inv.dd count=1 skip=63 bs=4096 of=inodebit.dd
1+0 records in
1+0 records out
4096 bytes (4.1 kB, 4.0 KiB) copied, 0.031477 s, 130 kB/s
root@kali:~/investigation# xxd -b inodebit.dd
00000000: 11111111 11111111 00001111 00000000 00000000 00000000 .....
00000006: 00000000 00000000 00000000 00000000 00000000 00000000 .....
0000000c: 00000000 00000000 00000000 00000000 00000000 00000000 .....
00000012: 00000000 00000000 00000000 00000000 00000000 00000000 .....
00000018: 00000000 00000000 00000000 00000000 00000000 00000000 .....
0000001e: 00000000 00000000 00000000 00000000 00000000 00000000 .....
00000024: 00000000 00000000 00000000 00000000 00000000 00000000 .....
0000002a: 00000000 00000000 00000000 00000000 00000000 00000000 .....
00000030: 00000000 00000000 00000000 00000000 00000000 00000000 .....
00000036: 00000000 00000000 00000000 00000000 00000000 00000000 .....
```

The I-node Table

The i-node table is the structure which holds all the i-nodes in a group. When printed using xxd it simply prints all the i-nodes sequentially. I used dd to extract the i-node table, and xxd to print a hex dump.

```
root@kali:~/investigation# dd if=inv.dd count=1 skip=64 bs=4096 of=inodetable.dd
1+0 records in
1+0 records out
4096 bytes (4.1 kB, 4.0 KiB) copied, 0.000352381 s, 11.6 MB/s
root@kali:~/investigation# xxd inodetable.dd
00000000: 0000 0000 0000 0000 71cb 8d4f 71cb 8d4f .....q..0q..0
00000010: 71cb 8d4f 0000 0000 0000 0000 0000 0000 q..0.....
00000020: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000030: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000040: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000050: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000060: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000070: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000080: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000090: 0000 0000 0000 0000 0000 0000 0000 0000 .....
```

I only extracted the first block here to show an example however, the i-node table is over 400 blocks in size.

The first ten i-nodes are reserved by the system and are usually empty, so I travelled farther down the table to find an allocated i-node to convert:

This is i-node 13, it belongs to group zero. The i-node in this system are 256 bytes in size however only the first 128 bits are used to store information.

```
00000c00: ed81 0000 1181 0500 32d0 8d4f 79d0 8d4f .....
00000c10: be24 8d4f 0000 0000 0000 0100 d002 0000 $.0....
00000c20: 0000 0000 0000 0000 6a30 0000 6b30 0000 .....
00000c30: 6c30 0000 6d30 0000 6e30 0000 6f30 0000 l0..m0..f
00000c40: 7030 0000 7130 0000 7230 0000 7330 0000 p0..q0..l
00000c50: 7430 0000 7530 0000 7630 0000 0000 0000 t0..u0..v
00000c60: 0000 0000 54e5 2dc3 0000 0000 0000 0000 ....T.-..
00000c70: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000c80: 0400 0000 0000 0000 0000 0000 0000 0000 .....
00000c90: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000ca0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000cb0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000cc0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000cd0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000ce0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000cf0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000d00: ed81 0000 cd43 0600 1ad0 8d4f 1ad0 8d4f .....G
```

Similar to the previous data structures, the i-node contains values in fixed locations:

To read the i-node table we need to understand the structure of the i-node.

Byte Range	Description	Essential
0-1	File mode (type and permissions) (see Tables 15.11, 15.12, and 15.13)	Yes
2-3	Lower 16 bits of user ID	No
4-7	Lower 32 bits of size in bytes	Yes
8-11	Access Time	No
12-15	Change Time	96-99 1 triple indirect block pointer Yes
16-19	Modification time	100-103 Generation number (NFS) No
20-23	Deletion time	104-107 Extended attribute block (File ACL) No
24-25	Lower 16 bits of group ID	108-111 Upper 32 bits of size / Directory ACL Yes/ No
26-27	Link count	112-115 Block address of fragment No
28-31	Sector count	116-116 Fragment index in block No
32-35	Flags (see Table 15.14)	117-117 Fragment size No
36-39	Unused	118-119 Unused No
40-87	12 direct block pointers	120-121 Upper 16 bits of user ID No
88-91	1 single indirect block pointer	122-123 Upper 16 bits of group ID No
92-95	1 double indirect block pointer	124-127 Unused No

Taken directly from class notes created by John Sheppard, based on the book File System Forensic Analysis by Brian Carrier.

The i-node holds the metadata of the file, which can be useful to gain information like the creation or access time of the file or the group it's located in, the file size and also holds pointers to the files location in memory.

Block pointers

Block pointers in EXT are memory locations stored in the i-node structure which point to the blocks where the file's content is stored. If the file requires 12 or fewer blocks of memory the information is stored as direct block pointers. Files larger than 12 blocks allocate a single indirect pointer. Larger files will require also a double indirect pointer, if the double indirect pointer field is filled, and the file requires more space a triple indirect pointer is created (Layton, 2011).

I have chosen below some of the more common fields to convert:

	Byte Location	Hexadecimal Value	Conversion
Lower 32 bits of size (bytes)	4-7	11 81 05 00	Total bits: 00 00 00 00 00 05 81 11
Upper 32 bits of size (bytes)	108-111	00 00 00 00	360,721 bytes
Access Time	8-11	32 d0 8d 4f	17/04/2012, 9:18:58 PM
Modification Time	16-19	be 24 8d 4f	17/04/2012, 9:07:26 AM
Direct block pointers (2 of 12)	40-87	6a 30,6b 30	12,394, 12,395
Lower 16 bits of group I.D.	24-25	00 00	
Upper 16 bits of group I.D.	122-123	00 00	Block Group 0

Once again the hex values are stored in little endian. The date and time values are stored in MAC time (Modify, Access, Create), also known as epoch time, which is the minute value in relation to Jan 1 1970.

Sometimes it is necessary to check the information manually and so it is important to know and understand the data structures and features of the file system.

Journal

An additional data structure used in EXT3 is the journal. Some file systems, including EXT3, use a data structure called a journal to keep track of changes made which haven't been saved to the system yet. This process is known as journaling and is used to recover lost information if the system crashes or something happens during the write process.

There are two categories of journaling, logical journals and physical journals. Logical journals store metadata only (i-nodes) while physical journals store all updates to the system, including content (data blocks). Updates are done using a sequence of transaction numbers.

The journals i-node is stored in the super block and is usually, but not always i-node 8. We saw that the journal in this system is allocated i-node 8 when we decoded the superblock (Carrier, 2005).

First, we have to locate the first block of the journal. We can do this by viewing the i-node table and finding the direct block pointer value for i-node 8:

```
00000700: 8081 0000 0000 0001 71cb 8d4f 71cb 8d4f ...
00000710: 71cb 8d4f 0000 0000 0000 0100 2880 0000 q..
00000720: 0000 0000 0000 0000 e401 0100 e501 0100 ...
00000730: e601 0100 e701 0100 e801 0100 e901 0100 ...
00000740: ea01 0100 eb01 0100 ec01 0100 ed01 0100 ...
00000750: ee01 0100 ef01 0100 f001 0100 f105 0100 ...
00000760: 0000 0000 0000 0000 0000 0000 0000 0000 ...
00000770: 0000 0000 0000 0000 0000 0000 0000 0000 ...
```

The circled value is the first block location stored in little endian, converting to 000101e4, and then 66,020. This is the value of the first block location.

Now I can use dd to extract the file and xxd to view the hex dump:

```
root@kali:~/investigation# dd if=inv.dd bs=4096 count=1 skip=66020 | xxd
1+0 records in
1+0 records out
00000000: c03b 3998 0000 0004 0000 0000 0000 1000  .;9.....
00000010: 0000 1000 0000 0001 0000 0002 0000 0001  .....
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  .....
00000030: 3fbc 0d46 7c6b 4628 9208 b635 3234 afad  ?..F|kF(...524..
00000040: 0000 0001 0000 0000 0000 0000 0000 0000  .....
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  .....
00000060: 0000 0000 0000 0000 0000 0000 0000 0000  .....
00000070: 0000 0000 0000 0000 0000 0000 0000 0000  .....
00000080: 0000 0000 0000 0000 0000 0000 0000 0000  .....
```

The first four bytes `c0 3b 39 98` is the journal's signature entry.

The first 12 bytes are the journals header:

Byte Range	Description	Essential
0-3	Signature (0xC03B3998)	Yes
4-7	Block type (see Table 15.29)	Yes
8-11	Sequence Number	Yes

Table 15.29. Values for the type field in the journal header fields.

Value	Description
1	Descriptor block
2	Commit block
3	Superblock version 1
4	Superblock version 2
5	Revoke block

Taken directly from notes created by John Sheppard, based on the book File System Forensic Analysis by Brian Carrier.

Signature	Block Type	Sequence No
C0 3b 39 98	4- superblock version 2	00 00 00 00

This header is part of the journal's superblock which has the following structure:

Table 15.30. Data structure for version 1 and 2 of the journal superblock.

Byte Range	Description	Essential
0–11	Standard header (see Table 15.28)	Yes
12–15	Journal block size	Yes
16–19	Number of journal blocks	Yes
20–23	Journal block where the journal actually starts	Yes
24–27	Sequence number of first transaction	Yes
28–31	Journal block of first transaction	Yes
32–35	Error number	No

Table 15.31. Data structure for the remainder of the version 2 journal superblock.

Byte Range	Description	Essential
36–39	Compatible Features	No
40–43	Incompatible Features	No
44–47	Read only compatible Features	No
48–63	Journal UUID	No
64–67	Number of file systems using journal	No
68–71	Location of superblock copy	No
72–75	Max journal blocks per transaction	No
76–79	Max file system blocks per transaction	No
80–255	Unused	No
256–1023	16-byte IDs of file systems using the journal	No

Taken directly from class notes created by John Sheppard based on the book File System Forensic Analysis written by Brian Carrier.

As can be seen from the header information, this is a version 2 superblock, so we use the entire table, I will do some conversions as an example:

	Byte Location	Hexadecimal Value	Conversion
Journal Block Size	12-15	00 00 10 00	4096
No of Journal Blocks	16-19	00 00 10 00	4096
Sequence no of First Transaction	24-27	00 00 00 02	

There are four data structures in the journal, the superblock, the descriptor, commit and revoke. Similar to the block groups each structure holds different information about the journal and can be read in the same way as the superblock.

```

root@kali:~# cd investigation/
root@kali:~/investigation# jls inv.dd
JBlk  Description
0:    Superblock (seq: 0)
sb version: 4
sb version: 4
sb feature_compat flags 0x00000000
sb feature_incompat flags 0x00000000
sb feature_ro_incompat flags 0x00000000
1:    Allocated Descriptor Block (seq: 2)
2:    Allocated FS Block 64
3:    Allocated Commit Block (seq: 2, sec: 0.0)
4:    Allocated Descriptor Block (seq: 3)
5:    Allocated FS Block 64
6:    Allocated Commit Block (seq: 3, sec: 0.0)
7:    Allocated Descriptor Block (seq: 4)
8:    Allocated FS Block 64
9:    Allocated Commit Block (seq: 4, sec: 0.0)
10:   Allocated Descriptor Block (seq: 5)
11:   Allocated FS Block 64

```

Journals are massive files and difficult to read but there is a tool in The Sleuth Kit toolkit which locates and lays out the journal, jls.exe lays out journal in a more readable and understandable fashion.

Charting the Filesystem

Now that we know what is happening in the system and how the tools in The Sleuth Kit are working, we can use these tools in our investigation.

In order to get a good overview of what is happening in the system it's important to know where everything is. Fsstat gives a detailed overview of the system. Making a record of the information provided allows us to easily check for anomalies in the data, which may lead to the discovery of hidden files. Recording the design and structure of the system also provides a starting point for the investigation, and allows me to see where certain data structures are located so I can more easily and accurately recover files such as the superblock for example.

Block Group	Block Group 0	Block Group 1	Block Group 2	Block Group 3	Block Group 4	Block Group 5	Block Group 6	Block Group 7
I-node range	1-7712	7713-15424	15425-23136	23137-30848	30849-38560	38561-46272	46273-53984	53985-61696
I-nodes per group	7712							
Free i-nodes	7692	7707	7712	7712	7712	7712	7712	7712
I-nodes in use	20	5	0	0	0	0	0	0
Block range	0-32767	32768-65535	65536-98303	98304-131071	131072-1683839	163840-196607	196608-229375	229376-246527
Blocks per group	32768							
Free Blocks	31304	32217	28183	32222	32284	32222	32284	16606
Blocks in use	1464	551	4585	546	484	546	484	16162
Directories	2	3	0	0	0	0	0	0

The File System

Even though there is no boot sector because this file system image was taken from a USB key, which is not a bootable device, the first 1024 bytes are still reserved for boot code.

As can be seen from the table above there are eight block groups in the image, however only two currently contain directories, Block Group 0 and Block Group 1.

The first thing I notice here is the discrepancy in Block Group 2. There seems to be a large number of blocks in use here where there are no directories stored.

The same issue is prevalent in Block Group 7. I will investigate this further.

I may find some insight in the data structures within the block groups.

The Block Groups

It is necessary to record structural information regarding the data structures inside each block for a more complete overview of the system. Knowing and recording memory locations of these data structures allows me to access the files directly and analyse that data.

Block Group 0

Data Structure	Superblock	Group descriptor Table	Block Bitmap	I-node Bitmap	I-node Table	Data Blocks
Memory Location(block)	0-0	1-1	62-62	63-63	64-545	546-32767

Block Group 1

Data Structure	Superblock	Group descriptor Table	Block Bitmap	I-node Bitmap	I-node Table	Data Blocks
Memory Location(block)	32768-32768	32769-32769	32830-32830	32831-32831	32832-33313	33314-65535

Block Group 2

Data Structure	Superblock	Group descriptor Table	Block Bitmap	I-node Bitmap	I-node Table	Data Blocks
Memory Location(block)			65536-65536	65537-65537	65538-66019	65538-65537, 66020-98303

Block Group 3

Data Structure	Superblock	Group descriptor Table	Block Bitmap	I-node Bitmap	I-node Table	Data Blocks
Memory Location(block)	98304-98304	98305-98305	98366-98366	98367-98367	98368-98849	98850-131071

Block Group 4

Data Structure	Superblock	Group descriptor Table	Block Bitmap	I-node Bitmap	I-node Table	Data Blocks
Memory Location(block)			131072-131072	131073-131073	131074-131555	131074-131073, 131556-163839

Block Group 5

Data Structure	Superblock	Group descriptor Table	Block Bitmap	I-node Bitmap	I-node Table	Data Blocks
Memory Location(block)	163840-163840	163841-163841	163902-163902	163903-163903	163904-164385	164386-196607

Block Group 6

Data Structure	Superblock	Group descriptor Table	Block Bitmap	I-node Bitmap	I-node Table	Data Blocks
Memory Location(block)			196608-196608	196609-196609	196610-197091	196610-196609, 197092-229375

Block Group 7

Data Structure	Superblock	Group descriptor Table	Block Bitmap	I-node Bitmap	I-node Table	Data Blocks
Memory Location(block)	229376-229376	229377-229377	229438-229438	229439-229439	229440-229921	229922-246527

The memory allocations of the block groups appear to be continuous and consistent. Block groups 2, 4, and 6 do not contain either a superblock or group descriptor table, this is a feature of newer versions of EXT2, and EXT3 with the increase likelihood of thousands of data groups on larger systems. This realisation made the redundant nature of reserve copies seem wasteful and it was decided to reduce the number of copies stored by the system.

Once the file system layout has been recorded I can begin investigating the content.

Exploring Files

Everything in EXT, every data structure is treated as a file (special files) and every piece of data or metadata is stored in a file. User created files are in two forms, directories and files.

Directories

Directories in EXT are similar to files however their i-node holds a signature value which differentiates them. Directories are annotated in The Sleuth Kit using d/d.

Files

Files stored in EXT have no signature value, this field is reserved for file extension values. User created files are annotated in The Sleuth Kit using r/r (Carrier, File System Forensic Analysis).

The fls.exe tool gives us an overview of files in the system:

```
root@kali: ~/inv  
File Edit View Search Terminal Help  
root@kali:~/investigation# fls inv.dd  
d/d 11: lost+found  
r/r 12: United States Patent? 7156436.pdf  
r/r * 13(realloc): United States Patent? 6745949.pdf  
r/r 14: United States Patent? 7195269.pdf  
r/r * 15(realloc): United States Patent? 8066186.pdf  
r/r 16: United States Patent? 7866013.pdf  
r/r 17: United States Patent? 7913713.pdf  
r/r 18: United States Patent? 7957830.pdf  
r/r 19: United States Patent? 8105034.pdf  
r/r 20: United States Patent? D537759.pdf  
d/d 7713: .Trash-0  
d/d 61697: $OrphanFiles  
root@kali:~/investigation#
```

*The fls
reference using
i-nodes.*

*The * notation
represents a
deleted file.*

The file at i-node 11, lost+found is a system file which stores open program files if the system crashes. We can use fls to look inside a directory also:

```
root@kali:~/investigation# fls inv.dd 11  
root@kali:~/investigation#
```

*There is no output, meaning the directory
contains no files (is empty).*

The orphan files folder is also a system file. This holds files that have been left over after a parent application has been uninstalled from the system. The orphan files contain Orphan File-7716.

If we look in the .Trash-0 directory contains two more directories:

```
root@kali:~/investigation# fls inv.dd 7713  
d/d 7714: info  
d/d 7715: files  
root@kali:~/investigation#
```

Inside the directories are more files:


```

root@kali:~/investigation# fls inv.dd 7714
r/r 7717: United States Patent? 6745949.pdf.trashinfo
r/r 7718: United States Patent? 8066186.pdf.trashinfo
r/r * 7718(realloc): United States Patent? 8066186.pdf.trashinfo.SA9ICW
root@kali:~/investigation# fls inv.dd 7715
r/r 13: United States Patent? 6745949.pdf
r/r 15: United States Patent? 8066186.pdf
root@kali:~/investigation#

```

File Extraction

Once I know where all the files are I can start to extract them using the `icat.exe` tool. First I have to check the file information using the `istat.exe` tool. Also to be safe I will use the `blkcat.exe` tool to do a hex dump of the first block to check the file extension. I will show the process using the file allocated i-node 12, named United States Patent? 715636.pdf.

```

root@kali:~/investigation# istat inv.dd 12
inode: 12
Allocated
Group: 0
Generation Id: 3274564947
uid / gid: 0 / 0
mode: rwxrwxrwx
size: 428670
num of links: 1

Inode Times:
Accessed: 2012-04-17 16:18:44 (EDT)
File Modified: 2012-04-17 04:05:56 (EDT)
Inode Modified: 2012-04-17 16:18:44 (EDT)

Direct Blocks:
12288 12289 12290 12291 12292 12293 12294 12295
12296 12297 12298 12299 12301 12302 12303 12304
12305 12306 12307 12308 12309 12310 12311 12312
12313 12314 12315 12316 12317 12318 12319 12320
12321 12322 12323 12324 12325 0
12329 12330 12331 12332 12333 16
12337 12338 12339 12340 12341 32
12345 12346 12347 12348 12349 48
12353 12354 12355 12356 12357 64
12361 12362 12363 12364 12365 80
12369 12370 12371 12372 12373 96
12377 12378 12379 12380 12381 12382 12383 12384
12385 12386 12387 12388 12389 12390 12391 12392
12393

```

I stat lays out the file information associated with the i-node, from here I can see permissions and file size, if the file is allocated or unallocated and the data blocks it occupies.

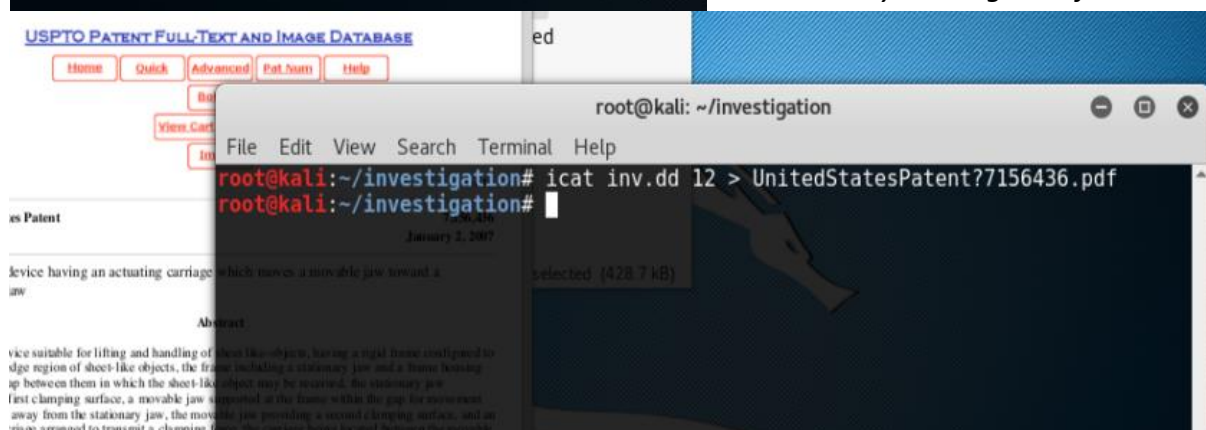
In the first data block, I can check the file extension before I extract the file.

```

root@kali:~/investigation# blkcat -h inv.dd 12288
25504446 2d312e35 0d25e2e3 cfd30d0a %PDF -1.5 .%.. ....
31343020 30206f62 6a0d3c3c 2f4c696e 149 0 ob j.<< /Lin
65617269 7a656420 312f4c20 34323836 eari zed 1/L 4286
37302f4f 20313531 2f452036 36323538 70/0 151 /E 6 6258
2f4e2031 312f5420 34323831 33372f48 /N 1 1/T 4281 37/H
205b2035 31362032 37325d3e 3e0d656e [ 5 16 2 72]> >.en

```

I then extract the file as a .pdf and store it in my investigation folder.



I repeated this process for all the allocated files on the system.

Deleted Files

Files which have been deleted are notated by the asterisk symbol. When a file is deleted the data stays in the system but the data blocks previously marked allocated in the bitmap are marked unallocated and are available for use. This means some deleted information may be recovered unless it's been overwritten. The icat tool has a command (-r) which recovers data which is unallocated and not yet overwritten.

This USB key contains no deleted files, the * notation is followed by (realloc), meaning the file was moved not erased.

For example in the file allocated i-node 13:

```
r/r 12: United States Patent? 7156436.pdf
r/r * 13(realloc): United States Patent
r/r 14: United States Patent? 7195269.pdf
r/r * 15(realloc): United States Patent
```

Was moved to the .Trash-0\files folder:

```
root@kali:~/investigation# fls inv.dd 7715
r/r 13: United States Patent? 6745949.pdf
r/r 15: United States Patent? 8066186.pdf
root@kali:~/investigation#
```

So to recover the files is the same process, istat to locate the first block, check the file extension with blkcat, and use the icat tool to restore the file.

Unknown Files

The .Trash-0 directory had two files with extension .trashinfo which I wasn't sure about, so I used istat and blkcat to take a closer look:

```
root@kali:~/investigation# istat inv.dd 7718
inode: 7718
Allocated
Group: 1
Generation Id: 3274564962
uid / gid: 0 / 0
mode: rrw-r--r--
size: 85
num of links: 1

Inode Times:
Accessed: 2012-04-17 16:20:22 (EDT)
File Modified: 2012-04-17 16:20:22 (EDT)
Inode Modified: 2012-04-17 16:20:22 (EDT)

Direct Blocks:
47108
root@kali:~/investigation#

root@kali:~/investigation# istat inv.dd 7717
inode: 7717
Allocated
Group: 1
Generation Id: 3274564960
uid / gid: 0 / 0
mode: rrw-r--r--
size: 85
num of links: 1

Inode Times:
Accessed: 2012-04-17 16:20:09 (EDT)
File Modified: 2012-04-17 16:20:09 (EDT)
Inode Modified: 2012-04-17 16:20:09 (EDT)

Direct Blocks:
47107
root@kali:~/investigation#
```



```
root@kali:~/investigation# blkcat inv.dd 47107
[Trash Info]
Path=United States Patent? 6745949.pdf
DeletionDate=2012-04-17T21:20:09
root@kali:~/investigation# blkcat inv.dd 47108
[Trash Info]
Path=United States Patent? 8066186.pdf
DeletionDate=2012-04-17T21:20:22
root@kali:~/investigation#
```

I discovered the files are some kind of script which stores deletion information.

Investigation Conclusion

I also performed other checks such as checking empty spaces for hidden data, like in the first 1024 bytes, reserved for boot code:

```
root@kali:~/investigation# dd if=inv.dd count=1 skip=0 bs=1024 | xxd
1+0 records in
1+0 records out
00000000: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000010: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000020: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000030: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000040: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000050: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000060: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000070: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000080: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000090: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000a0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000b0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000c0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000d0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000e0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000f0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000100: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000110: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000120: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000130: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000140: 0000 0000 0000 0000 0000 0000 0000 0000 .....
```

I realised the questions I had about the large number of blocks in use in Block Group 2 is because these are the blocks in use by the journal , 66,020 onward for over 400 blocks.

In total I recovered nine files, labelled patents, which are attached with this document.

Bibliography

Carrier, Brian, File System Forensic Analysis, published 2005.

Carrier, Brian, <http://www.sleuthkit.org/sleuthkit/man/fsstat.html>, published 2014, accessed 14/03/2017

Layton, Jeffrey B. , <http://www.linux-mag.com/id/8658/> ,published Thursday, June 9th, 2011, accessed 15/03/2017

linfo.org, <http://www.linfo.org/superblock> , published Apr 2013, accessed 18/03/2017

macaal, <http://www.linuxjournal.com/content/dd-image-and-how> published Jun 28, 2009 , accessed 15/03/2017

Pillai, Sarath ,<http://www.slashroot.in/understanding-file-system-superblock-linux> , published Mon, 09/07/2015, accessed 19/03/2017

Pillai, Sarath , <http://www.slashroot.in/inode-and-its-structure-linux> , published Sat, 12/01/2012 accessed 14/03/2017

Pomeranz, Hal, <https://digital-forensics.sans.org/blog/2008/12/24/understanding-indirect-blocks-in-unix-file-systems> , published 28 Dec 2008, accessed 20/03/2017.

Smith, James, <http://140.120.7.21/LinuxKernel/LinuxKernel/node17.html>, published Jun 2016, accessed 20/03/2017.