# Global Information Assurance Certification Paper

# *Taking advantage of Ext3 journaling file system in a forensic investigation*

*GCFA Gold Certification*
Author: Gregorio Narváez, gnarvae@yahoo.com
Adviser: Paul Wright

Accepted: December 30th, 2007

## *TABLE OF CONTENT*

# 1. Introduction

The Ext3 file system has become the default for most Linux distributions and thus is of great importance for any practitioner of forensics to understand how Ext3 handles files differently from the previous standard (Ext2) and how the knowledge of these differences can be applied to recover evidence as deleted files, and file activity. There is still the misconception that it is not possible to recover a deleted file from an Ext3 file system. This is what one of the developers, Andreas Dilger, said about it: *"In order to ensure that ext3 can safely resume an unlink after a crash, it actually zeros out the block pointers in the inode, whereas ext2 just marks these blocks as unused in the block bitmaps and marks the inode as "deleted" and leaves the block pointers alone."* [Linux Ext3 FAQ 2004]. The objective of this paper is show that it's not impossible to recover a deleted file from an Ext3, but also will show a couple of strategies that will allow accomplish several tasks that a forensic investigator faces on every case and that Ext3 nature offers interesting options:

- File recovery on files using the metadata stored by the journal on Ext3 including advantages, disadvantages and limitations.

- What happens with an Ext3 file system with its journal on an external device and how to deal with this situation? In this scenario TSK tools like jcat, jls will not work.

- An improved "Time Machine" that will allow an analyst to track down file activity beyond its last set of MACtimes.

With these three objectives in mind let's start.

## 1.1 Lab Setup

For the following demonstrations several file systems were created on a laptop running Fedora 5 (Kernel 2.6.20-1.2320) and using an external usb 2.0 40GB hard drive:

- `/dev/sda1` is an Ext2 FS called "workbench" that was used for storing the images of the other file systems.
- `/dev/sda5` is an Ext2 FS that served as a baseline, labeled "baseline"
- `/dev/sda6` is an Ext3 FS that has its own journal and mounted with default options (mode=ordered) and named "ext3default"
- `/dev/sda8` is an Ext3 FS that has its journal on an external device, in this case sda9
- `/dev/sda9` holds sda8 journal

Before creating the images of the file systems using dd we mount them with their default options, copied on them several pdf files, open them, deleted some of them and finally unmounting the file systems. The images created with dd were stored on sda1 mounted on /media/workbench, and the images file name have the following notation: `<dev>img.dd`, so `sda5img.dd` is the forensic image for /dev/sda5.

## 2. Ext3fs Journal Fundamentals

What we are going to explore in this research paper are some unique options that a forensic investigator has on Ext3 files systems such as how to recover those files that were deleted either by mistake or intentionally, or tracking down historical file activity. But before starting to explore those options a quick review of the Ext3 journal is in order to better understand the methods that will be described. It's assumed that the reader has basic notions of the internal structure of Ext2/Ext3 file system, if that's not the case, two excellent references that can helpful to better understand some of the concepts mentioned through this document are `Brian Carrier's File System Forensic Analysis` and `Forensic Discovery by Dan Farmer & Wietse Venema`.

The Ext3 file system created by Dr. Stephen C. Tweedie in 1999; is a journaled version of the old Ext2 file system standard. A Journaling File System is a type of file system that allows the OS to keep a log of all file system changes before writing the data to disk. This log is called a `journal`, and it is usually a circular log in an especially-allocated area of the file system. This type of file system offers better probabilities to avoid corruption in case of a power outage or system crash. Just keep in mind that Ext3 is not the only file system to offer journaling capabilities, other file systems like NTFS, JFS, JFS2 and ReiserFS offers similar capabilities.

For a better understanding of how a file system works a five layer model will be used [Carrier 2005]. A similar model is often used on SANS Forensic course. This model offers a framework that allows describing and understanding almost any type of file system in existence and reduces their complexity allowing the development of techniques and procedures that can be applied over different file system types. A summary of this five layer model is as follows:

- `Files System:` Describes the structure of the file system, this information includes the size of data units, structure offset and mounting information, group descriptors, etc. In a Linux/Unix system this structure is referred as `"superblock"`. Is in the superblock that we can find the information described in this category.

- `File Name:` This category includes the directory entries where the FS stores the file name and the inode number for that file. You can compare this category with a book's table of content

- `Metadata:` Contains information describing the characteristics and structure of a file. In Unix/Linux this metadata structures are known as "inodes". The following information is stored in an inode: file owner identifier, file type, file access permissions, MACtimes, number of links, file size and pointers to content blocks.

- `Data/content:` This is category ferers to structures where the actual content of a file is stored

- `Application:` In the case of Ext3 file system we are talking about the journal. The structure of the journal records the modifications of the file system. Originally was designed for fast recovery

Traditionally the first four layers are considered essential data structures, while data structures

belonging to the last layer are considered non-essential [Carrier 2005], most techniques and procedures are focused on essential data structures, but this research will show that the last layer contains information crucial to a forensic investigation, especially on Ext3 file systems.

A file in Ext2/3 will be composed as shown the following figure:

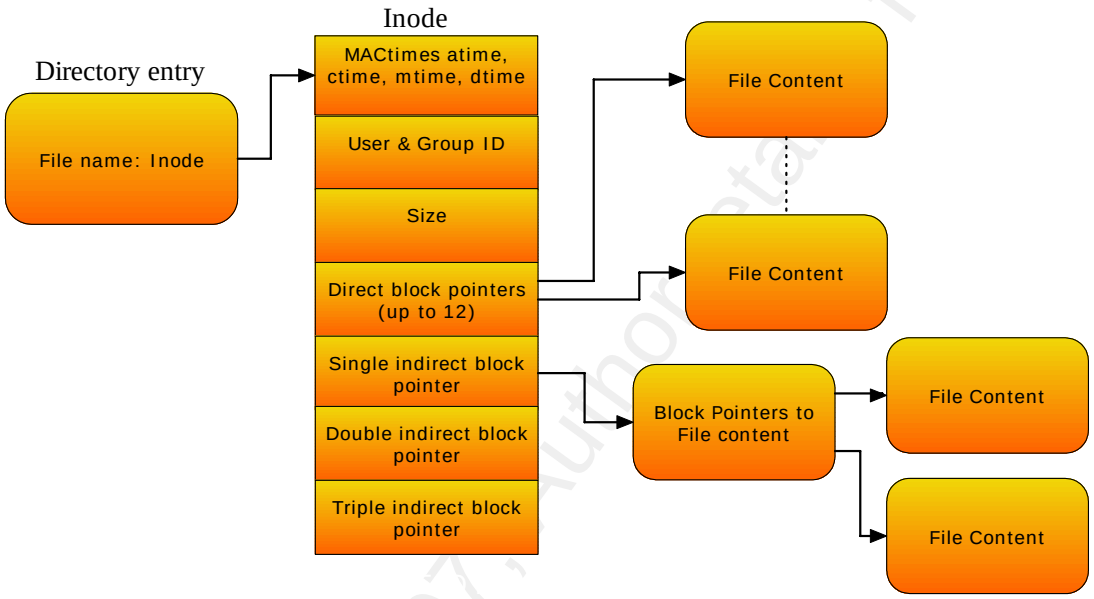


*Fig 2.1 File name, inode and content blocks relationship*

The information on the application layer on Ext3 is what it's called `journal`. This structure stores changes on the file system and it is the main difference with Ext2. Ext3 offers three modes of journaling, the difference is what data the journal stores and how, impacting the performance. Let's take a quick review of the three modes:

- `Journal` - Logs all file system data and metadata changes. This journaling mode minimizes the chance of losing the changes you have made to any file in an Ext3 file system. This approach has a penalty in performance since data is being written twice (once to the journal, a second time to the file system), making it the slowest of the three journaling modes.

- `Ordered` - Only logs changes to file system metadata (inodes), but flushes file data updates to disk before making changes to associated file system metadata, keeping the journal synchronized with data writes. This is the default Ext3 journaling mode.

- `Write back` - Only logs changes to file system metadata but relies on the standard file system write process to write file data changes to disk. This is the fastest Ext3 journaling mode.

From a forensic point of view the first mode (journal) will be preferred because offers the most information regarding file system activity and facilitates deleted file content recovery, the other two options only permits to recover metadata activity on the file system. Unfortunately the default behavior of Ext3 journal is ordered and thus only metadata changes are recorded. Keep this in mind when analyzing an Ext3 file system.

In most cases the journal exists in the same Ext3 file system. It resides in a special area within the file system. The first structure in the journal is called `journal superblock` and keeps information regarding the block size of the journal, total number of blocks that the journal has available for storage, where the journal actually starts, sequence number of the first transaction, where the first journal transaction is located and general structure information about the journal. The mechanism of the journal keeps track of the changes in a file system with the use of transactions sequences. A transaction sequence is made up of the following components

- `Descriptor block:` Every transaction initiates with a block that describes the beginning of the transaction
- `Metadata block:` There can be one or many metadata blocks for each transaction, this blocks are where the changes are recorded
- `Commit block:` Depending on the journal mode, basically this block indicates the end of a successful transaction.
- `Revoke block:` If there is an error during the operation a revoke block is created and holds a list of the file system block that needs to restore during a consistency check.

To better understand how the journal is composed we can observe the following graphic

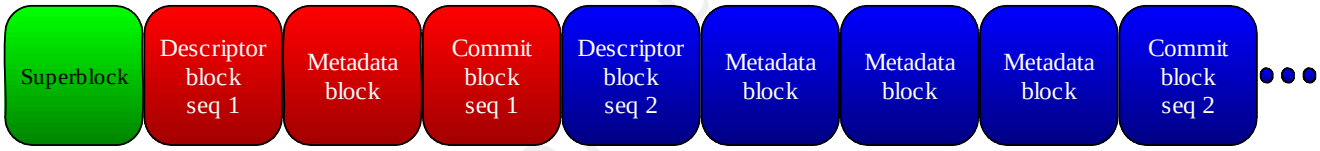


*Fig 2.2: Ext3 Journal general structure*

What is interesting is that the journaling mechanism act at the block level, this part of how journaling works is the core for the concepts discussed in this research, because when a bit of metadata (`inode`) of a file system is modified the whole block where that inode resides is copied to the journal. In other words the neighbor inodes (same block) are copied to the journal. This concept is what Dan Farmer and Witse Venema named the `bystander effect` [Farmer and Venema 2007].

## 2.1 Journal Life Cycle

The life cycle of the journal is also something important to be aware of during an investigation. The journal is restarted every time the file system is unmount and mounted again, or when the journal becomes full it start all over itself reusing the first blocks, like a circular list; this will destroy any evidence within the journal, so its good idea to grab an image of the journal as soon as possible using the TSK tools or just imaging the suspect file system with dd for later analysis

To make this point clear let's take a look to a series of snapshots from a journal during different points of time of an Ext3 file system, to accomplish this we will use the jls tool from TSK. This tool allows the investigator to browse the journal. If we execute the command `jls sda6img.dd` on sda6 image file that holds the Ext3 file system will be mounted with standard options as soon as was initialized by mke2fs :

```
JBlk    Descriptrion
0:      Superblock (seq: 0)
1:      Unallocated FS Block Unknown
2:      Unallocated FS Block Unknown
3:      Unallocated FS Block Unknown
4:      Unallocated FS Block Unknown
5:      Unallocated FS Block Unknown
6:      Unallocated FS Block Unknown
7:      Unallocated FS Block Unknown
8:      Unallocated FS Block Unknown
9:      Unallocated FS Block Unknown
10:     Unallocated FS Block Unknown
[REMOVED]
```

This output shows the journal, right after the file system was created. As we can observe the sequence number in the superblock is 0 an all the blocks are unallocated without any type of transaction or information stored. Now the file system is mounted:

```
JBlk    Descriptrion
0:      Superblock (seq: 0)
1:      Allocated Descriptor Block (seq: 2)
2:      Allocated FS Block 183
3:      Allocated Commit Block (seq: 2)
4:      Unallocated FS Block Unknown
5:      Unallocated FS Block Unknown
6:      Unallocated FS Block Unknown
7:      Unallocated FS Block Unknown
8:      Unallocated FS Block Unknown
9:      Unallocated FS Block Unknown
[REMOVED]
```

When the file system is being mounted, we can see on journal block 1 that sequence number started to increase. Then a series of files were copied to the file system:

```
JBlk    Descriptrion
0:      Superblock (seq: 0)
1:      Allocated Descriptor Block (seq: 2)
2:      Allocated FS Block 183
3:      Allocated Commit Block (seq: 2)
4:      Allocated Descriptor Block (seq: 3)
5:      Allocated FS Block 295094
6:      Allocated FS Block 1
7:      Allocated FS Block 295095
8:      Allocated FS Block 295093
9:      Allocated FS Block 295595
10:     Allocated FS Block 0
[REMOVED]
```

We can appreciate that after copying files, the sequence number has increased sequentially; also it shows the file system blocks that has been updated during the copy process. Then we unmount and then mount the file system again:

```
JBlk    Descriptrion
0:      Superblock (seq: 0)
1:      Allocated Descriptor Block (seq: 16)
2:      Allocated FS Block 327682
3:      Allocated Commit Block (seq: 16)
4:      Unallocated Descriptor Block (seq: 3)
5:      Unallocated FS Block 295094
6:      Unallocated FS Block 1
```

```
7:      Unallocated FS Block 295095
8:      Unallocated FS Block 295093
9:      Unallocated FS Block 295595
10:     Unallocated FS Block 0
[REMOVED]
```

The sequence number keeps increasing but since the file system was unmounted cleanly the journal restarts and the next transaction begins at block 1 of the journal, the process of overwriting evidence takes places (journal blocks 2 and 3). Also we can notice that the remaining transactions from the previous mount are now marked as unallocated. Finally some more operations occur in the device:

```
JBlk    Descriptrion
0:      Superblock (seq: 0)
1:      Allocated Descriptor Block (seq: 16)
2:      Allocated FS Block 327682
3:      Allocated Commit Block (seq: 16)
4:      Allocated Descriptor Block (seq: 17)
5:      Allocated FS Block 182
6:      Allocated FS Block 1
7:      Allocated FS Block 183
8:      Allocated FS Block 295595
9:      Allocated FS Block 683
10:     Allocated FS Block 181
[REMOVED]
```

Here the overwriting of the journal keeps taking away precious evidences as can be observed on journal blocks 4 to 10. One must notice that if the journal operation mode is for metadata only (`ordered / write back modes`) the blocks shown in the journal are blocks containing some sort of metadata, such blocks could be part of the inode table, inode bitmaps or block bitmaps. If it's working in journal mode (metadata + data) also will include copies of blocks with file content.

Just one last observation regarding the preservation of evidence on the journal: when mounting an Ext3 image file (remember always should be mounted as read only), keep in to account that extra precaution must be taken otherwise the integrity of the image will be compromised the moment the journal is being reset and replay against the file system when the file system is mounted (this applies also for forensic images). Just to give you an idea of what we refer with extreme caution, take the hash value of a device image, an then mount it, without doing anything else, unmount, an take a second hash to the device image an compare both values, here is an example:

```
[root@Akulal workbench]# md5sum sda7img.dd > sda7img.md5
[root@Akulal workbench]# mount -t ext3 -o loop sda7img.dd test
[root@Akulal workbench]# umount test
[root@Akulal workbench]# md5sum sda7img.dd > sda7img.md5a
[root@Akulal workbench]# more *md5*
::::::::::::::
sda7img.md5
::::::::::::::
2bd7c92834e839636289926b9451066ac9  sda7img.dd
::::::::::::::
sda7img.md5a
::::::::::::::
831c6ea560ec9d48d28bc5219b5eff58  sda7img.dd
```

Oops, we forgot to put the option "`ro`" (`read only`) when mounting the image, immediately we unmount the image, but it's already to late; the journal was reset/replay and that causes the hash values not to match.

## 2.2 File Deletion Process: Ext2 Vs Ext3

In this section we'll review how Ext2 deletes a file and compare it to Ext3 procedure. On Ext2 the deletion of a file by the OS can be resumed as marking the directory entry, inode and data blocks that make up a file as unallocated, this marking occurs in the block and inode bitmaps of each block group. For our example we have a file called "`reference.pdf`" with a size of `562378 bytes`, its structure looks something similar to the following diagram:

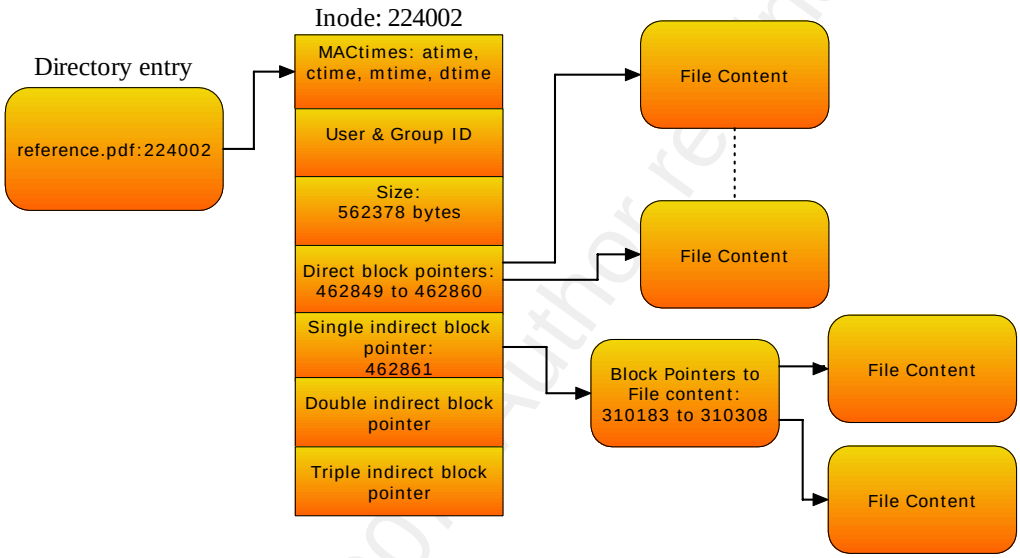


*Fig 2.3: File before deletion in Ext2 FS*

When file "reference.pdf" is deleted on Ext2 the inode and content blocks are marked as unallocated so the OS can reuse them when needed but basically all the information is still there as shown in the next figure:

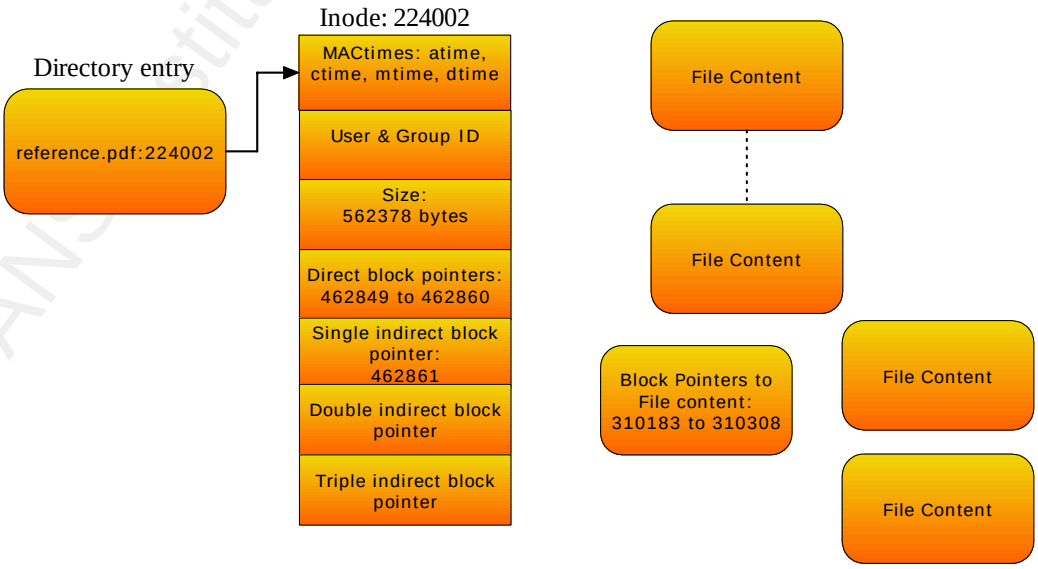


*Fig 2.4: File after deletion in Ext2 FS*

This can be seen on a real life system using two tools from TSK ils and istat. The following is the output of ils tool on an Ext2 file system with the option –r listing all deleted inodes on the image.

```
[root@Akula1 workbench]# ils -r sda5img.dd
class|host|device|start_time
ils|Akula1||1194288876
st_ino|st_alloc|st_uid|st_gid|st_mtime|st_atime|st_ctime|st_mode|st_nlink|st_size|
st_block0|st_block1
[REMOVED]
224002|f|0|0|1182963421|1194249392|1194282669|100644|0|562378|462849|462850
224003|f|0|0|1182963371|1194249453|1194282669|100644|0|69330|462988|462989
[REMOVED]
```

Let’s take the first inode on this list, this inode (224002) shows a probable file with a size of 562378 bytes. Take a look to the stats of inode 224002 using istat.

```
[root@Akula1 workbench]# istat sda5img.dd 224002
inode: 224002
Not Allocated
Group: 14
Generation Id: 231094706
uid / gid: 0 / 0
mode: -rw-r--r--
size: 562378
num of links: 0

Extended Attributes  (Block: 689)
security.selinux=root:object_r:file_t:s0

Inode Times:
Accessed:       Mon Nov  5 01:56:32 2007
File Modified:  Wed Jun 27 11:57:01 2007
Inode Modified: Mon Nov  5 11:11:09 2007
Deleted:        Mon Nov  5 11:11:09 2007

Direct Blocks:
462849 462850 462851 462852 462853 462854 462855 462856
462857 462858 462859 462860 462862 462863 462864 462865
462866 462867 462868 462869 462870 462871 462872 462873
462874 462875 462876 462877 462878 462879 462880 462881
462882 462883 462884 462885 462886 462887 462888 462889
462890 462891 462892 462893 462894 462895 462896 462897
462898 462899 462900 462901 462902 462903 462904 462905
462906 462907 462908 462909 462910 462911 462912 462913
462914 462915 462916 462917 462918 462919 462920 462921
462922 462923 462924 462925 462926 462927 462928 462929
462930 462931 462932 462933 462934 462935 462936 462937
462938 462939 462940 462941 462942 462943 462944 462945
462946 462947 462948 462949 462950 462951 462952 462953
462954 462955 462956 462957 462958 462959 462960 462961
462962 462963 462964 462965 462966 462967 462968 462969
462970 462971 462972 462973 462974 462975 462976 462977
462978 462979 462980 462981 462982 462983 462984 462985
462986 462987

Indirect Blocks:
462861
[root@Akula1 workbench]#
```

As we can see this last output yields a lot of useful information, like MAC times, permission for the file, and more important for the purpose of recovering the file, the size of the file and the direct and indirect pointers to the data blocks. If the data blocks have not been overwritten by other files the recovery becomes quite simple using icat from TSK and just to verify that the file is what we deleted at the beginning we use file to find out its type:

```
[root@Akula] workbench]# icat -r sda5img.dd 224002 > recovered
[root@Akula] workbench]# file recovered
recovered: PDF document, version 1.3
```

To make the comparison between both mechanisms will use the same ”reference.pdf” file as in the previous example, but this time it’s being copied to an Ext3 file system, with the same size, but different inode and block pointer because it’s located in sda6, this file has the following structure:

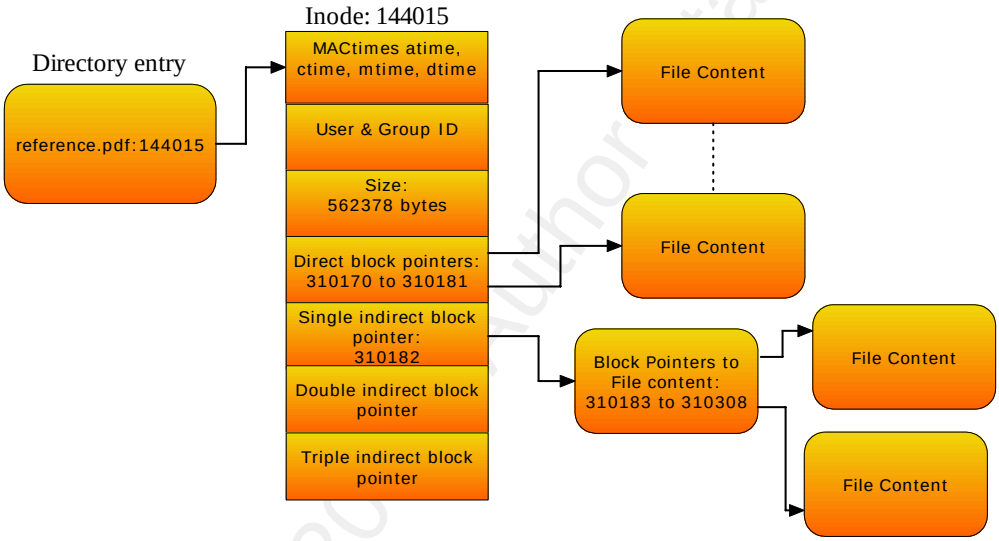


*Fig 2.4: File before deletion in Ext3 FS*

In Ext3 the OS takes some other steps when a file deletion occurs. Inside the inode the size of the file as well the block pointers (direct and indirect) are zeroed leaving us with out a way to trace back what data blocks belongs to particular file as shown next diagram:

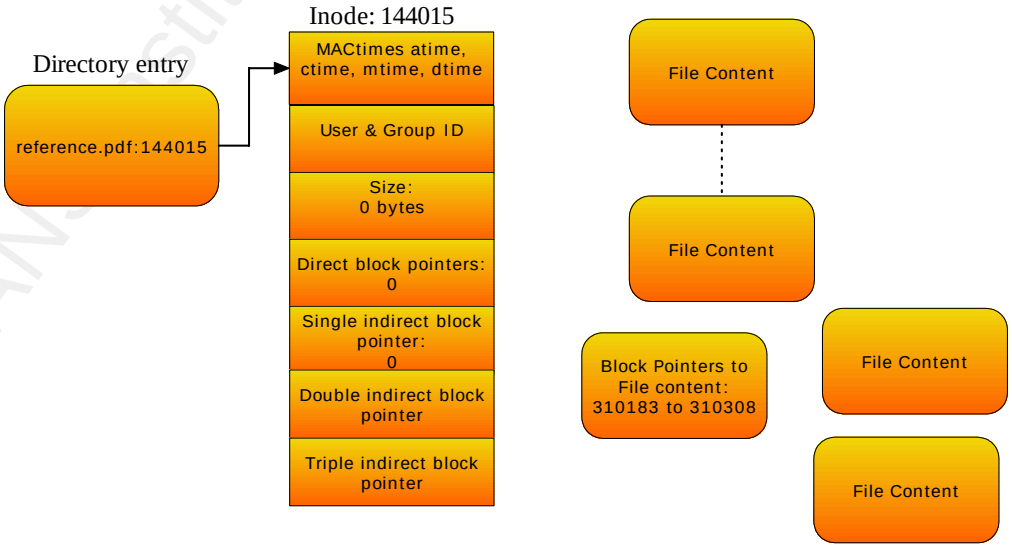


*Fig 2.5: File after deletion in Ext3 FS*

Now let's see what really happens on an Ext3 file system. After deleting some files and imaging the file system on sda6 the process to find and recover a file is repeated:

```
[root@Akula1 workbench]# ils -r sda6img.dd
class|host|device|start_time
ils|Akula1||1194252024
st_ino|st_alloc|st_uid|st_gid|st_mtime|st_atime|st_ctime|st_mode|st_nlink|st_size|
st_block0|st_block1
144014|f|0|0|1194231994|1194231994|1194231994|40755|0|0|0|0
144015|f|0|0|1194231994|1194231982|1194231994|100644|0|0|0|0
[REMOVED]
[root@Akula1 workbench]# istat sda6img.dd 144015
inode: 144015
Not Allocated
Group: 9
Generation Id: 3670456940
uid / gid: 0 / 0
mode: -rw-r--r--
size: 0
num of links: 0

Extended Attributes  (Block: 295595)
security.selinux=root:object_r:file_t:s0

Inode Times:
Accessed:       Sun Nov  4 21:06:22 2007
File Modified:  Sun Nov  4 21:06:34 2007
Inode Modified: Sun Nov  4 21:06:34 2007
Deleted:        Sun Nov  4 21:06:34 2007

Direct Blocks:
[root@Akula1 workbench]#
```

The output confirms what we discussed before, the links to the data blocks and the size of the file has been zeroed out. This is why recovering a deleted file from Ext3 has always being considered almost an impossible task. Now let's discuss an option that a forensic investigator could use in order to recover a file under Ext3.

## 3. File Recovery Using Ext3 FS Journal

The first step in this technique is to have the inode of the deleted file; there are several ways to obtain this, like using debugfs or using fls or ils from TSK. For simplicity of this discussion let's say we already know the file inode and now we verify this information with ils:

```
[root@Akula1 workbench]# ils -r sda6img.dd
class|host|device|start_time
ils|Akula1||1194252024
st_ino|st_alloc|st_uid|st_gid|st_mtime|st_atime|st_ctime|st_mode|st_nlink|st_size|
st_block0|st_block1
144014|f|0|0|1194231994|1194231994|1194231994|40755|0|0|0|0
144015|f|0|0|1194231994|1194231982|1194231994|100644|0|0|0|0
144016|f|0|0|1194231994|1182970801|1194231994|100644|0|0|0|0
144017|f|0|0|1194231994|1182970801|1194231994|100644|0|0|0|0
144018|f|0|0|1194231994|1182970801|1194231994|100644|0|0|0|0
144019|f|0|0|1194231994|1182970801|1194231994|100644|0|0|0|0
144020|f|0|0|1194231994|1194231982|1194231994|100644|0|0|0|0
```

Let's check stats for first two inodes:

```
[root@Akula1 workbench]# istat sda6img.dd 144014
inode: 144014
Not Allocated
Group: 9
Generation Id: 3670456939
uid / gid: 0 / 0
mode: drwxr-xr-x
size: 0
num of links: 0

Extended Attributes  (Block: 295595)
security.selinux=root:object_r:file_t:s0

Inode Times:
Accessed:       Sun Nov  4 21:06:34 2007
File Modified:  Sun Nov  4 21:06:34 2007
Inode Modified: Sun Nov  4 21:06:34 2007
Deleted:        Sun Nov  4 21:06:34 2007

Direct Blocks:
[root@Akula1 workbench]# istat sda6img.dd 144015
inode: 144015
Not Allocated
Group: 9
Generation Id: 3670456940
uid / gid: 0 / 0
mode: -rw-r--r--
size: 0
num of links: 0

Extended Attributes  (Block: 295595)
security.selinux=root:object_r:file_t:s0

Inode Times:
Accessed:       Sun Nov  4 21:06:22 2007
File Modified:  Sun Nov  4 21:06:34 2007
Inode Modified: Sun Nov  4 21:06:34 2007
Deleted:        Sun Nov  4 21:06:34 2007
```

```
Direct Blocks:
[root@Akula1 workbench]#
```

We can learn that inode 144014 was linked to a directory and inode 144015 contained a file but their stats and block pointers are lost, also we learned that inode 144015 belongs to block group 9 and now is marked as unallocated as result of the deletion process. Now we can look the stats of block group 9:

```
[root@Akula1 workbench]# fsstat sda1img.dd | grep -i "group: 9"
Group: 9:
  Inode Range: 144001 - 160000
  Block Range: 294912 - 327679
  Layout:
    Super Block: 294912 - 294912
    Group Descriptor Table: 294913 - 294913
    Data bitmap: 295093 - 295093
    Inode bitmap: 295094 - 295094
    Inode Table: 295095 - 295594
    Data Blocks: 295595 - 327679
```

If we take a look closely to the information of group 9 we can see that for this group there are 16000 inodes (Inode range: 144001-160000), and the inode table has a size of 500 blocks (Inode Table: 295095-295594). Each block of the inode table has 32 inodes (16000 divided by 500), thus inode 144015 it's the 15th entry in the table and its content is located in the first block of the inode table. Remember the journal works at the block level so the block we must look for in the journal is 295095 (in this case the first block of the inode table). Checking the output from jls, we find out that there are several references to 295095, lets use the first one, but keep in mind that in the case of multiple instance of a particular block we might have to analyze each one. One way to decide the chronological order in case there are multiple references to the block we are interested is to take a look at the sequence number of the transaction where the block is being referenced. The lower the number the sequence number is, the older the inode copy will be.

```
[root@Akula1 workbench]# jls  sda1img.dd
JBlk    Descriptrion
0:      Superblock (seq: 0)
1:      Allocated Descriptor Block (seq: 2)
2:      Allocated FS Block 183
3:      Allocated Commit Block (seq: 2)
4:      Allocated Descriptor Block (seq: 3)
5:      Allocated FS Block 295094
6:      Allocated FS Block 1
7:      Allocated FS Block 295095
8:      Allocated FS Block 295093
9:      Allocated FS Block 295595
[REMOVED]
```

The output shows that block 7 of the journal contains information regarding an operation on the inode table of group 9, and since the journal at least records copies of the metadata (default journal mode is ordered) that has been modified; we can look for a copy of inode 144015 within the journal. There are cases that checking each instance of a particular block in the journal must be analyzed, but in this case we just want to recover the earliest one.

As we discussed before the entry we are looking inside the inode table is the 15th within the inode range for block group 9, in order to extract the copy of inode 144015 from the journal we will use jcat in combination with dd and xxd; but before we need to find out the inode size. Usually inode size is 128 bytes on Ext2/3 file systems but we can obtain that information running fsstat or dump2fs on the image

of the file system to verify, here we'll use dumpe2fs:

```
[root@Akula1 workbench]# dumpe2fs sda6img.dd|grep -i "inode size"
dumpe2fs 1.38 (30-Jun-2005)
Inode size:               128
[root@Akula1 workbench]#
```

By the way dumpe2fs is also an excellent source for file system info. Ok, know execute the following:

```
jcat sda6img.dd 8 7 | dd bs=128 skip=14 count=1 | xxd
```

A little explanation on the previous command is in order. The TSK tool jcat takes three parameters, the first one is the image file, the second is the inode where the journal begins and the third is the entry within the journal that we are interested (in this case is entry 7), dd is being used to carve one inode (144015) from journal block 7. As we mentioned before we are interested on the 15th inode from the inode range that indicates that we need to skip the 14 blocks before getting the one of interest, hence skip=14, and the size of the inode is 128 bytes. The last part will give hex dump format by sending the output from the previous two commands (jcat and dd) to xxd. Now let's get the result:

```
[root@Akula1 workbench]# jcat sda6img.dd 8 7 | dd bs=128 skip=14 count=1 | xxd
1+0 records in
1+0 records out
128 bytes (128 B) copied, 0.00402034 seconds, 31.8 kB/s
0000000: a481 0000 ca94 0800 b1b3 8246 bc13 2e47  ...........F1..G
0000010: dd96 8246 0000 0000 0000 0100 b004 0000  ...F........`...
0000020: 0000 0000 0000 0000 9abb 0400 9bbb 0400  ................
0000030: 9cbb 0400 9dbb 0400 9ebb 0400 9fbb 0400  ................
0000040: a0bb 0400 a1bb 0400 a2bb 0400 a3bb 0400  ................
0000050: a4bb 0400 a5bb 0400 a6bb 0400 0000 0000  ................
0000060: 0000 0000 bcba c6da ab82 0400 0000 0000  ....l...........
0000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
[root@Akula1 workbench]#
```

What we have is a copy of what used to be inode 144015 at that time, but we need to interpret this. The structure of an inode is quite extensive and out of the scope of this research but there are several sources where a more detailed description can be obtained [Carrier 2005]. Here we'll focus mainly on the search for block pointers, it's important to notice that metadata in our examples is stored by the OS in little endian notation (in this case a x86 platform) and as such should be read from the hex dumps

**Table 3.1: Partial inode structure**

| Byte Range | Description |
|---|---|
| 4 to 7 | Lower 32 bits of file size in bytes |
| 40 to 87 | List of twelve direct block pointers |
| 88 to 91 | Single indirect block pointer |
| 92 to 95 | Double indirect block pointer |
| 95 to 99 | Triple indirect block pointers |

Now applying this knowledge we have the following:

The size of the file that was linked to inode 144015, at that time was 562378 bytes (0x0894ca):

```
0000000: a481 0000 ca94 0800 b1b3 8246 bc13 2e47  ..........F1..G
```

There are 12 direct block pointers: 310170 to 310181 (0x4bb9a TO 0x4bba5):

```
0000020: 0000 0000 0000 0000 9abb 0400 9bbb 0400  ................
0000030: 9cbb 0400 9dbb 0400 9ebb 0400 9fbb 0400  ................
0000040: a0bb 0400 a1bb 0400 a2bb 0400 a3bb 0400  ................
0000050: a4bb 0400 a5bb 0400 a6bb 0400 0000 0000  ................
```

There is a single indirect block pointer: 310182 (0x4bba6):

```
0000050: a4bb 0400 a5bb 0400 a6bb 0400 0000 0000  ................
```

There are no double or triple indirect block pointers:

```
0000050: a4bb 0400 a5bb 0400 a6bb 0400 0000 0000  ................
0000060: 0000 0000 bcba c6da ab82 0400 0000 0000  ....1...........
```

But we are not done yet! We only have the first 12 data blocks. That's just 49152 bytes of the file, and according to the information recovered from the copy of inode 144015 the file size is 562378 bytes. Don't lose your hope, we still have the single indirect block pointer that indicates block 310182 (0x4bba6) may contain the pointers to the rest of the file.

So, let's take a look at the content of block 310182, for that we'll use dcat:

```
[root@Akula1 workbench]# dcat -h sda6img.dd 310182
0       a7bb0400 a8bb0400 a9bb0400 aabb0400     .... .... .... ....
16      abbb0400 acbb0400 adbb0400 aebb0400     .... .... .... ....
32      afbb0400 b0bb0400 b1bb0400 b2bb0400     .... .... .... ....
48      b3bb0400 b4bb0400 b5bb0400 b6bb0400     .... .... .... ....
64      b7bb0400 b8bb0400 b9bb0400 babb0400     .... .... .... ....
80      bbbb0400 bcbb0400 bdbb0400 bebb0400     .... .... .... ....
96      bfbb0400 c0bb0400 c1bb0400 c2bb0400     .... .... .... ....
112     c3bb0400 c4bb0400 c5bb0400 c6bb0400     .... .... .... ....
128     c7bb0400 c8bb0400 c9bb0400 cabb0400     .... .... .... ....
144     cbbb0400 ccbb0400 cdbb0400 cebb0400     .... .... .... ....
160     cfbb0400 d0bb0400 d1bb0400 d2bb0400     .... .... .... ....
176     d3bb0400 d4bb0400 d5bb0400 d6bb0400     .... .... .... ....
192     d7bb0400 d8bb0400 d9bb0400 dabb0400     .... .... .... ....
208     dbbb0400 dcbb0400 ddbb0400 debb0400     .... .... .... ....
224     dfbb0400 e0bb0400 e1bb0400 e2bb0400     .... .... .... ....
240     e3bb0400 e4bb0400 e5bb0400 e6bb0400     .... .... .... ....
256     e7bb0400 e8bb0400 e9bb0400 eabb0400     .... .... .... ....
272     ebbb0400 ecbb0400 edbb0400 eebb0400     .... .... .... ....
288     efbb0400 f0bb0400 f1bb0400 f2bb0400     .... .... .... ....
304     f3bb0400 f4bb0400 f5bb0400 f6bb0400     .... .... .... ....
320     f7bb0400 f8bb0400 f9bb0400 fabb0400     .... .... .... ....
336     fbbb0400 fcbb0400 fdbb0400 febb0400     .... .... .... ....
352     ffbb0400 00bc0400 01bc0400 02bc0400     .... .... .... ....
368     03bc0400 04bc0400 05bc0400 06bc0400     .... .... .... ....
384     07bc0400 08bc0400 09bc0400 0abc0400     .... .... .... ....
400     0bbc0400 0cbc0400 0dbc0400 0ebc0400     .... .... .... ....
416     0fbc0400 10bc0400 11bc0400 12bc0400     .... .... .... ....
432     13bc0400 14bc0400 15bc0400 16bc0400     .... .... .... ....
448     17bc0400 18bc0400 19bc0400 1abc0400     .... .... .... ....
464     1bbc0400 1cbc0400 1dbc0400 1ebc0400     .... .... .... ....
480     1fbc0400 20bc0400 21bc0400 22bc0400     ....  ... !... "...
496     23bc0400 24bc0400 00000000 00000000     #... $... .... ....
512     00000000 00000000 00000000 00000000     .... .... .... ....
```

```
528     00000000 00000000 00000000 00000000     .... .... .... ....
544     00000000 00000000 00000000 00000000     .... .... .... ....
[REMOVED]
```

The content of block 310182 is a list of data blocks, we presume that they are part of the file, but keep in mind that these blocks could have been overwritten. The block range is from 310183 to 310308 (0x4bbaf to 0x4bc24). Finally to recover what we seems be the file we can carve the data with dd using the block pointers from the inode copy residing in the journal and either manually or with foremost recover the file.

In the case that the files to recover were fragmented an option would be to carve out the blocks manually using dd or a similar tool into a single file and then run foremost to see if it figure out what type of file is and recover it. In this example, luck was on our part and the file was not fragmented, as can be seen in the block pointers recovered:

```
[root@Akula1 workbench]# dd bs=4096 skip=310168 count=141 if=sda6img.dd
of=recover.dd
141+0 records in
141+0 records out
577536 bytes (578 kB) copied, 0.00310458 seconds, 186 MB/s

[root@Akula1 workbench]# foremost -b 4096 -o recovery -t pdf  recover.dd
Processing: recover.dd
|*|
[root@Akula1 workbench]# cd recovery
[root@Akula1 recovery]# more *.txt
Foremost version 1.5.1 by Jesse Kornblum, Kris Kendall, and Nick Mikus
Audit File

Foremost started at Mon Nov  5 19:59:13 2007
Invocation: foremost -b 4096 -o recovery -t pdf recover.dd
Output directory: /media/workbench/ext3default/recovery
Configuration file: /usr/local/etc/foremost.conf
------------------------------------------------------------------
File: recover.dd
Start: Mon Nov  5 19:59:13 2007
Length: 564 KB (577536 bytes)

Num      Name (bs=4096)        Size      File Offset     Comment

0:      00000002.pdf          553 KB            8192
Finish: Mon Nov  5 19:59:13 2007

1 FILES EXTRACTED

pdf:= 1
------------------------------------------------------------------

Foremost finished at Mon Nov  5 19:59:13 2007
```

In this case we carved out some blocks before and some blocks after from where the information says the file could be, foremost will take care of the rest. In figure 3.1 we can see a graphical resume of this technique:

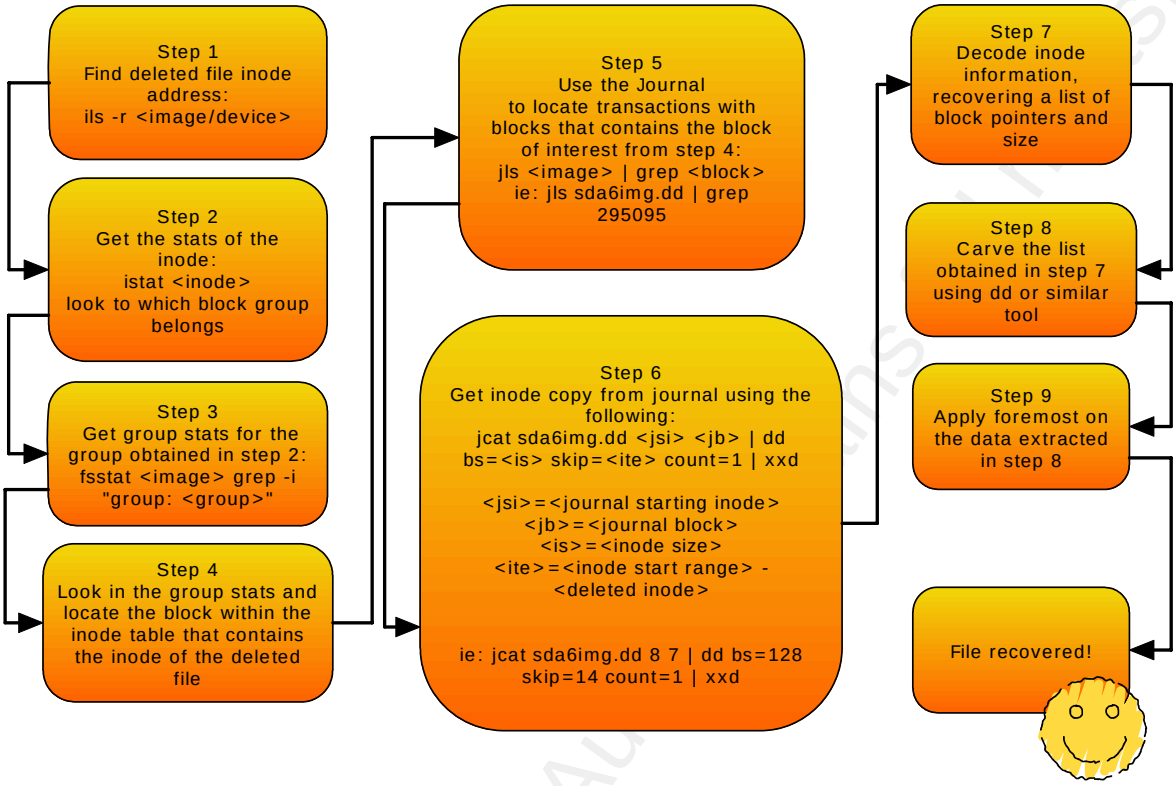


*Fig 3.1: Deleted file recovery using Ext3 journal*

## 4. Special Case: When Ext3 Journal is an External Device

Most of the time you will find the journal was created within the file system but the administrator could decide where the journal will be, in other words it's possible that the journal resides in an external device or file system. Under this scenario tools such as jcat, jls will not work, they will send a message that says "`Cannot determine file system type`".

In this case the administrator has two devices one is sda8 that will be the Ext3 file system; the other sda9, will be the journal for sda8. To create this configuration a two step process is required; the administrator will first create the journal with something like this:

```
[root@Akula1 workbench]# mke2fs -O journal_dev -L journal_dev /dev/sda9
mke2fs 1.38 (30-Jun-2005)
Filesystem label=journal_dev
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
0 inodes, 722917 blocks
0 blocks (0.00%) reserved for the super user
First data block=0
0 block group
32768 blocks per group, 32768 fragments per group
0 inodes per group
Superblock backups stored on blocks:

Zeroing journal device: done
[root@Akula1 workbench]#
```

The option `-O journal_dev` and the option `-L` set up the label for this device (sda9). As we can see in this output there are no inodes or block groups, the block size for the journal is 4096 bytes and the journal is 722917 blocks long, giving us in theory a size of 2.76 GB (722917 by 4096).

The second step creates the actual ext file system on sda8, indicating that its journal will be on sda9.

```
[root@Akula1 workbench]# mke2fs -J device=/dev/sda9  -L ext3external /dev/sda8
mke2fs 1.38 (30-Jun-2005)
Filesystem label=ext3external
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
368000 inodes, 734965 blocks
36748 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=754974720
23 block groups
32768 blocks per group, 32768 fragments per group
16000 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912

Writing inode tables: done
Adding journal to device /dev/sda9: done
Writing superblocks and filesystem accounting information: done


This filesystem will be automatically checked every 29 mounts or
```

```
180 days, whichever comes first.  Use tune2fs -c or -i to override.
[root@Akula1 workbench]#
```

If you want to recreate this scenario just remember that both devices must be created with the same block size value, this fact is mentioned in the man page for mke2fs [Ts'o 2002].

Now a forensic investigator might not know the exact configuration of the system being analyzed, so he or she will have to find out where the journal resides. There are several options at hand. One is to use fsstat from TSK on the device or image, another option is use dumpe2fs also over the device/image. Here's the output from dumpe2fs:

```
[root@Akula1 workbench]# dumpe2fs sda9img.dd
dumpe2fs 1.38 (30-Jun-2005)
Filesystem volume name:   journal_dev
Last mounted on:          <not available>
Filesystem UUID:          52559687-e4a5-4c79-b8a0-fbcc006af0c4
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
Filesystem features:      journal_dev
Default mount options:    (none)
[REMOVED]
Journal block size:       4096
Journal length:           722917
Journal first block:      2
Journal sequence:         0x00000031
Journal start:            0
Journal number of users:  1
Journal users:            fd7de6be-490b-4a6a-a0a6-9d093a09170d
[root@Akula1 workbench]# dumpe2fs sda8img.dd
dumpe2fs 1.38 (30-Jun-2005)
Filesystem volume name:   ext3external
Last mounted on:          <not available>
Filesystem UUID:          fd7de6be-490b-4a6a-a0a6-9d093a09170d
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
Filesystem features:      has_journal ext_attr resize_inode filetype sparse_super
large_file
[REMOVED]
Journal UUID:             52559687-e4a5-4c79-b8a0-fbcc006af0c4
Journal device:           0x0809
Default directory hash:   tea
Directory Hash Seed:      c2c2ce20-d8a4-47c9-962c-3e5f45b7789b
[REMOVED]
```

This output shows us that sda9 is the journal of sda8. We know learn this by comparing the Journal UUID from sda8 with Filesystem UUID from sda9. They match!

## 4.1 Journal Structure

Well, don't despair if you encounter an scenario like the one described above, Brian Carrier author of File System Forensic Analysis [Carrier 2005] has done a superb job of detailing the structures and inner workings of most common file systems in use, including Ext2/Ext3. From this reference it's possible to obtain a detailed description of the journal structure. But there is a caveat: these structures are focused on a journal inside of an Ext3 file system. If we recall the output from the creation of the external journal there are no inodes to speak of, so how this type of journal is structured? To start answering that question lets execute the following command:

```
# blkid
/dev/sda1:       LABEL="workbench"       UUID="5c7fe035-0eb3-4bb0-9337-2987db6661cf"
TYPE="ext2"
/dev/sda5:       LABEL="baseline"        UUID="ba082c1b-93e5-459f-8db5-67ec1caae3c1"
TYPE="ext2"
/dev/sda6:      LABEL="ext3default"      UUID="62a95b00-514b-4918-84f6-7731d1a41d2a"
SEC_TYPE="ext2" TYPE="ext3"
/dev/sda7:         LABEL="test"          UUID="d251c669-91d6-442f-bacf-3f8a23306f8"
SEC_TYPE="ext2" TYPE="ext3"
/dev/sda8:     LABEL="ext3external"      UUID="fd7de6be-490b-4a6a-a0a6-9d093a09170d"
SEC_TYPE="ext2" TYPE="ext3"
/dev/sda9:      LABEL="journal_dev"      UUID="52559687-e4a5-4c79-b8a0-fbcc006af0c4"
TYPE="jbd"
```

As we can see sda9 is being reported as "jbd". JBD stands for Journal Block Device. Ext3 actually does not interact directly with the OS, instead there is a mechanism called JBD. Sovani makes a clear definition on this: "A journaling filesystem first records all the operations it has performed in the journal. Once the set of operations that is part of one single atomic operation has completed and been recorded in the journal, only then is it written to the actual block device." [Sovani 2006]. The JBD structure described in Sovani's article is at the core the same as an Ext3 file system. The difference between a standard Ext3 with the journal as part of the file system and a journal in an external device is basically that the external device contains only the journal and no other structures but the journal structures in both are the same.

In order to decode this external journal a series of scripts were developed, the source is included for revision, and comments. The analysis the first script does, is to automate the process to decode each block from the journal. The script is written using awk and takes as input a hex dump from the device that acts as journal.

There are five types of blocks that a journal could have, the first four are administrative and are known as: `Superblock, Descriptor, Commit` and `Revoke` blocks. The fifth type is the one that stores the metadata or data that is recorded in the journal depending on the journal operation mode. Each administrative block type holds information related to its type, but all four administrative blocks share the same format on the first 12 bytes. This common structure receives the name of "header". The fifth type can be either metadata blocks that holds copies of the inodes being modified in the file system, if the journal is using ordered/write back modes and content blocks if the journal is in journaled mode. The internal structures for the different types of block are resumed on the following tables and are based from Carter's work on Ext3 forensics [Carter 2005]. For simplicity only the fields that were of interest in the development of the scripts are shown:

**Table 4.1: Journal administrative block standard header**

| Byte Range | Description | Values |
|---|---|---|
| 0-3 | Signature | 0xc03b3998 |
| 4-7 | Block type | 1 Descriptor<br>2 Commit<br>3 Superblock Version1<br>4 Superblock Version 2<br>5 Revoke |
| 8-11 | Sequence number | Any |

**Table 4.2: Journal Superblock**

| Byte Range | Description |
|---|---|
| 0-11 | Standard Header |
| 12-15 | Journal Block Size |
| 16-19 | Number of Journal blocks |
| 20-23 | Journal block where the journal actually start |
| 24-27 | Sequence number of first transaction |
| 28-31 | Journal block of first transaction |

**Table 4.3: Journal descriptor block**

| Byte Range | Description | Values |
|---|---|---|
| 0-11 | Standard Header | See table 1 |
| 12-15 | File system block | Any |
| 16-19 | Entry flags | 0x01 Journal block has escaped (Note 1)<br>0x02 Entry has the same UUID as the previous (SAME_UUID)<br>0x04 Block was deleted by current transaction (currently not in use)<br>0x08 Last entry in descriptor block |
| 20-23 | UUID (Does not exist if SAME_UUID flag is set) | |

Note 1: The value 0x01 it is to indicate that the block has the same value as the signature

**Table 4.4: Journal commit block**

| Byte Range | Description |
|---|---|
| 0-11 | Standard Header |

**Table 4.5: Journal revoke block**

| Byte Range | Description |
|---|---|
| 0-11 | Standard Header |
| 12-15 | Size in bytes of revoke data |
| 16-SIZE | List of 4-byte file system block addresses being revoked |

Before we start with decoding it's necessary to mention that the notation used by the journal for storing the information of all administrative blocks is big endian and it's not platform dependent, but remember that if you are looking to one of the metadata/content blocks the notation will depend on the system that originates it. Here are some examples of how to apply the information on the tables to manually decode different types of journal blocks.

In the first example we are reading the first block that is part of the journal skipping block 0 from the file system:

```
[root@Akula1 ext3external]# dd bs=4096 skip=1 count 1 if=sda9img.dd | xxd | less
0000000: c03b 3998 0000 0004 0000 0000 0000 1000  .;9.............
0000010: 000b 07e5 0000 0002 0000 0031 0000 0000  ...........1....
0000020: 0000 0000 0000 0000 0000 0001 0000 0000  ................
0000030: 5255 9687 e4a5 4c79 b8a0 fbcc 006a f0c4  RU....Ly.....j..
0000040: 0000 0001 0000 0000 0000 0000 0000 0000  ................
[REMOVED]
00000a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000100: fd7d e6be 490b 4a6a a0a6 9d09 3a09 170d  .}..I.Jj....:...
[REMOVED]
```

As the table at the beginning of this section indicates the first 4 bytes (0-3) will always contain a specific signature (0xc03b3998), this indicates that this is an administrative block in the journal. In bytes 4 and 7 we can find out what type of administrative block we are looking at. Here the value is 0x04, and according to the tables it's the journal superblock

Next is a descriptor block, again we can observe the signature in bytes 0 to 3 and the type in bytes 4 to 7. The value of 0x01 in these bytes gives us the type of block; in this case we are dealing with a descriptor block. The following 4 bytes gives us the sequence number in this case is 47 (0x2f).

```
[root@Akula1 ext3external]# dd bs=4096 skip=2 count 1 if=sda9img.dd | xxd | less
0000000: c03b 3998 0000 0001 0000 002f 0000 00b7  .;9......./....
0000010: 0000 0008 0000 0000 0000 0000 0000 0000  ................
0000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
[REMOVED]
```

Also we can observe that the FS block being modified and hence copied to the journal is 183 (0xb7) and the following 4 bytes indicates that this is the last entry for this transaction (0x08).

A commit block is very simple because only uses the standard header (bytes 0 to 11). Here it shows the signature (c03b3998), type (2) and sequence number (0x2f).

```
[root@Akula1 ext3external]# dd bs=4096 skip=4 count 1 if=sda9img.dd | xxd | less
0000000: c03b 3998 0000 0002 0000 002f 0000 0000  .;9......./....
0000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
[REMOVED]
```

The last example shows a revoke block. Here we can see the signature, type and transaction sequence

number. What is interesting is that shows on bytes 12 to 15 the size in bytes of data being revoked, in this case 248 bytes (0xf8), the rest is a list of 4 byte block addresses that will be revoked.

```
[root@Akula1 ext3external]# dd bs=4096 skip=6 count 1 if=sda9img.dd | xxd | less
0000000: c03b 3998 0000 0005 0000 002c 0000 00f8  .;9........,....
0000010: 000b 1556 000a ef55 000b 1957 000a eb54  ...V...U...W...T
0000020: 000b 0d54 000b 2694 000a f757 000b 1155  ...T..&....W...U
0000030: 000a f356 000b 0552 000b 2292 000a e351  ...V...R..".....Q
0000040: 000b 0953 000b 2693 000a df50 000a c5d0  ...S..&....P....
0000050: 000a e753 000a e352 0000 07cb 0000 0bcc  ...S...R........
0000060: 000a ff59 0000 0bcd 000a fb58 000b 1d58  ...Y.......X...X
0000070: 000b 2159 0000 13d4 0000 17d5 000a b800  ..!Y............
0000080: 0000 17d6 000a b80d 000a c64d 000a bc0f  ...........M....
0000090: 000a ca4f 000a c5cf 000a bc0e 000a ca4e  ...O...........N
00000a0: 000a c1ce 000a 81f6 0000 1823 0000 1c24  ...........#...$
00000b0: 0000 1c25 0000 0e68 0000 03e8 0000 1269  ...%...h.......i
00000c0: 0000 126a 000a d9a5 000a d5a4 000b 3424  ...j..........4$
00000d0: 000a dda6 000b 3022 000a cda1 000b 3423  ......0".....4#
00000e0: 000a d1a3 000a d1a2 000b 2dec 000b 29ea  ..........-...).
00000f0: 000b 2deb 000a bea8 0000 0000 0000 0000  ..-.............
0000100: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000110: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000120: 0000 0000 0000 0000 0000 0000 0000 0000  ................
[REMOVED]
```

## 4.2 How to Automate Journal Decoding

Now applying the information on how to read the administrative blocks of a journal manually, a script named journal.awk was developed and here it is:

```
#Ver 1.00
#November 6 2007
#Author: Gregorio Narvaez Jr


#Initialize variables
# <sig> contains the signature that every administrative block has in the first 4
bytes of the header
# <rec>: is the counter of the lines in the output
# <descriptor>: its a flag that indicates that indicates two things:
#1. That a descriptor block has been found
#2. What analysis case the script will be considered
#If <block> is set to 1 will print the corresponding header and will set the
variable
#<ind> which will help to make the math for printing the second column as the
number of bytes
# or as blocks. For the examples if "block=1", <ind> is being set to 4096 (block
size of our
# examples)

BEGIN {
      sig="c03b3998"
      flags=0
      rec=0
      descriptor=0
#Prints the value of option block  (0/1) that is passed when the script is invoked
with -v block
      print "Signature " sig
      printf "Options: block=%s\n", strtonum(block)
      if (block==1) {
```

```
                print "Rec: JBDblkoff:  Description:"
                ind=4096
                factor=1
                }
        else if (block==0) {
                print "Rec: JBDbyteoff:  Description:"; ind=1; factor=4096
                }
        else  {
              print "Not a Valid parameter use -v block=0|1";
              exit
              }
        }

#For deployment the script should be changed so ind is passed as a parameter or to
#obtain the block size by other means like calling dumpe2fs

#Here is where the decoding process takes place for each line of input
#Remeber the input should be a dd stream of blocks formated as a hex dump or
#a file that contains such output
# Example 1:
# dd if=/dev/sda9 | xxd > test.xxd
# awk -v block=0 -f journal.awk test.xxd | less
# Example 2:
# dd if=/dev/sda9 | xxd | awk -v block=1 -f journal.awk | less
# In a future a small shell script will be created to hide the complexity of
# the use of this script and act more like a single command

#------------------------- Analysis ------------------------------------------
#  <JBD> contains the offset in bytes where the line begins, this is field
position 1 ($1)
#<test> holds the values of the fields 2 and 3 to check if we are looking at the
signature
#of an administrative block (0xc03b3998)
#If comparison is true then we have found the beginning of an Administrative block
#  <type> contains the value of field 5 on the hex dump, this field allows to
decide what type
# of administrative block we have found

{

test=$2$3

if (test==sig) {
            JBD="0x"$1
            JBDo=strtonum(JBD)/strtonum(ind)
            type=$5
            seqn="0x"$6$7
# Type 1 is a Descriptor block
            if (type=="0001")
                {
                printf "%d      %s                Descriptor block (seq %s)\n",
strtonum(rec), JBDo, strtonum(seqn)
                rec++
                fsblk="0x"$8$9
                jblkc=1
                printf "%d      %s               FS Block %s\n", strtonum(rec),
JBDo+(jblkc*factor), strtonum(fsblk)
#Set descriptor flag indicating the first case of analysis for a descriptor block
                descriptor=1
                jblkc++
                }
# Type 2 is a Commit block
            else if (type=="0002")
                {
```

```
                printf "%d       %s                 Commit block (seq %s)\n",
strtonum(rec), JBDo, strtonum(seqn)
                descriptor=0
                }
#Type 3 and 4 indicates a Journal Superblock
           else if (type=="0003") printf "%d    %s          Superblock Ver 1 (seq
%s)\n", JBDo, strtonum(seqn)
           else if (type=="0004") printf "%d    %s          Superblock Ver 2 (seq
%s)\n", strtonum(rec), JBDo, strtonum(seqn)
#Type 5 Indicates a revoke block
           else if (type=="0005")
                {
                printf "%d       %s                 Revoke block (seq %s)\n",
strtonum(rec), JBDo, strtonum(seqn)
                descriptor=0
                }
#Failsafe in case we find something unexpected
           else printf "%d:  0PSSS Not recognized: $s\n", strtonum(rec), JBDo
           rec++
           }
# Try to find  the flags that indicates we already read the last block entry in a
descriptor block
else if (descriptor==1) {
                flags=$3
                if   (flags=="0008"   ||   flags=="0009"   ||   flags=="000a"   ||
flags=="000b") descriptor=0
#Set descriptor flag indicating the second case of analysis for a descriptor block
                else descriptor=2
                }
else if (descriptor==2) {
                fsblk="0x"$4$5
                flags=$7
                printf "%d     %s              FS Block %s\n", strtonum(rec),
JBDo+(jblkc*factor), strtonum(fsblk)
                rec++
                jblkc++
                if   (flags=="0008"   ||   flags=="0009"   ||   flags=="000a"   ||
flags=="000b") descriptor=0
#Set descriptor flag indicating the third case of analysis for a descriptor block
                else {
                      fsblk="0x"$8$9
                      printf "%d     %s            FS Block %s\n", strtonum(rec),
JBDo+(jblkc*factor), strtonum(fsblk)
                      rec++
                      jblkc++
                      descriptor=3
                      }
                }
else if (descriptor==3) {
                flags=$3
                if   (flags=="0008"   ||   flags=="0009"   ||   flags=="000a"   ||
flags=="000b") descriptor=0
                else {
                      fsblk="0x"$4$5
                      printf "%d     %s            FS Block %s\n", strtonum(rec),
JBDo+(jblkc*factor), strtonum(fsblk)
                      rec++
                      jblkc++
                      flags=$7
                      if  (flags=="0008"  ||  flags=="0009"  ||  flags=="000a"  ||
flags=="000b") descriptor=0
                      else  {
                            fsblk="0x"$8$9
                            printf  "%d       %s                 FS  Block  %s\n",
```

```
strtonum(rec), JBDo+(jblkc*factor), strtonum(fsblk)
                                rec++
                                jblkc++
                                }
                        }
                }
}
#Let's clarify the analysis of the Descriptor block:
#By analyzing a lot of blocks from the external journal it was noticed two things:

#The Descriptor block contains a list of the FS block that were being updated by
that transaction
#The Descriptor block present three analysis cases:
#Case 1: For the second 16 (16-31) bytes of the block (line 2 in the hex dump) we
look if the FS block read in
#bytes 12-15 is the last entry in the Descriptor being analyzed using field $3
#Case 2: Is to analyze bytes (32-47) the third line of hex dump. We look for flags
that indicates
#  the last entry on descriptor has been read. We use field $7 for comparison.
Block address are located at
# fields $4-$5 and $8-$9
#Case 3: to analyze from byte 48 and onwards until a flag that indicates the last
entry on the Descriptor. From
#the fourth line until the end of the block in the hex dump a pattern is repeated.
This pattern is
# field $3:flags, fields $4$5 FS block, field $7 flags, fields FS block

#The reason to discard field $2 and field $6 for the flags is that maximum value
under for the flags combining
# all of them is 0x0f, hence the first 2 significant bytes (field $2 or filed $3)
will be 0000 making them
# redundant.

#The possible combination for the flags that indicates a last entry condition are:
# 0008 Last entry in Descriptor
# 0009 Last entry (0x08) + Journal block  has been escaped (0x01)
# 000a Last entry (0x08) + Entry has the same UUID (0x02)
# 000b Last entry (0x08) + Entry has the same UUID (0x02) + Journal block has been
escaped (0x01)

#Any  combination  considering  the  value  0x04  (Block  was  deleted  by  this
transaction) was not considered
# to check is last entry condition occurred because is not currently in use
```

To invoke the scripts it is necessary to pass a parameter `-v block=0|1`. It is a flag to activate a format option for the output if block is set to 1 the script will show the offset from the beginning of the file system in blocks in the second column, if set to 0 will display the offset in bytes in the second column.

The input source can be the result of a device being read with dd and formatted with xxd or a file that received that treatment previously. Here is the output of the script against sda9 image (the external journal)

```
[root@Akula1 workbench]# dd bs=4096 count=10 if=sda9img.dd | xxd | awk -v block=1
-f journal.awk
10+0 records in
10+0 records out
40960 bytes (41 kB) copied, 0.000163428 seconds, 251 MB/s
Signature c03b3998
Options: block=1
Rec: JBDblkoff:  Description:
```

```
0    1          Superblock Ver 2 (seq 0)
1    2          Descriptor block (seq 47)
2    3          FS Block 183
3    4          Commit block (seq 47)
4    5          Commit block (seq 43)
5    6          Revoke block (seq 44)
6    7          Descriptor block (seq 44)
7    8          FS Block 688130
8    9          FS Block 0
9    10          FS Block 688128
10    11          FS Block 1
11    12          FS Block 688129
12    13          FS Block 720896
13    14          FS Block 181
14    15          FS Block 683
15    16          FS Block 183
[root@Akula1 workbench]#
```

And finally we have a way to read an external journal! Now the same steps to recover a deleted file can be applied on a case that involves an external journal.

## 4.3 Verification against jls on Ext3 Internal Journal (sda6)

Other way to check the validity of this script was applying it to an Ext3 file system with internal log; we use sda6 to test it. Here are the two outputs side by side; on the left the journal obtained via jls, on the right what our script yield.

**Table 4.6: jls vs journal.awk on sda6**

| **jls /dev/sda6 |less** | **dd bs=4096 count=1000 if=/dev/sda6 | xxd | awk -v block=0 -f journal.awk | less** |
|---|---|
| JBlk Descriptrion<br>0: Superblock (seq: 0)<br>1: Unallocated Descriptor Block (seq: 16)<br>2: Unallocated FS Block 327682<br>3: Unallocated Commit Block (seq: 16)<br>4: Unallocated Descriptor Block (seq: 17)<br>5: Unallocated FS Block 182<br>6: Unallocated FS Block 1<br>7: Unallocated FS Block 183<br>8: Unallocated FS Block 295595<br>9: Unallocated FS Block 683<br>10: Unallocated FS Block 181<br>11: Unallocated FS Block 17126<br>12: Unallocated FS Block 17185<br>13: Unallocated Commit Block (seq: 17)<br>14: Unallocated Commit Block (seq: 18)<br>15: Unallocated Commit Block (seq: 19)<br>16: Unallocated FS Block Unknown<br>17: Unallocated FS Block Unknown<br>18: Unallocated FS Block Unknown<br>19: Unallocated FS Block Unknown<br>20: Unallocated FS Block Unknown<br>21: Unallocated FS Block Unknown<br>31: Unallocated FS Block Unknown<br>32: Unallocated FS Block Unknown<br>33: Unallocated FS Block Unknown<br>34: Unallocated Commit Block (seq: 3)<br>35: Unallocated Descriptor Block (seq: 4)<br>36: Unallocated FS Block 295095<br>37: Unallocated FS Block 183 | Signature c03b3998<br>Options: block=0<br>Rec: JBDbyteoff: Description:<br>0 2822144 Superblock Ver 2 (seq 0)<br>1 2826240 Descriptor block (seq 16)<br>2 2830336 FS Block 327682<br>3 2834432 Commit block (seq 16)<br>4 2838528 Descriptor block (seq 17)<br>5 2842624 FS Block 182<br>6 2846720 FS Block 1<br>7 2850816 FS Block 183<br>8 2854912 FS Block 295595<br>9 2859008 FS Block 683<br>10 2863104 FS Block 181<br>11 2867200 FS Block 17126<br>12 2871296 FS Block 17185<br>13 2879488 Commit block (seq 17)<br>14 2883584 Commit block (seq 18)<br>15 2887680 Commit block (seq 19)<br>16 2965504 Commit block (seq 3)<br>17 2969600 Descriptor block (seq 4)<br>18 2973696 FS Block 295095<br>19 2977792 FS Block 183 |

The script gave us the same information in terms of type of block and transaction sequence number, where it diverge is that ils gives the status of the journal block being displayed (allocated/unallocated) and the script at the time of this writing does not, but the script gives the offset in bytes/blocks of the of the administrative blocks or the copy of data/metadata blocks where they can be found (2nd column); that information is very helpful to extract any of the blocks and get more information from them.

For example, if we require extracting the copy of the inode table as in the example of section 3 we can use the following shell script:

```
# ejcat.sh ver 1.0
# Script to extrac an specific inode from an external journal transaction
# Author: Gregorio Narvaez Jr
# November 6 2007
# <skiplblk> offset in blocks within the journal where the block was copied (2nd
col value from awk script)
# <blksize>, <isize> block and inode sizes in bytes
# <skipi> offset in blocks for the inode table
# <device> the source to analyze

skipblk=$1
blksize=$2
isize=$3
skipi=$(($4-1))
device=$5

echo "This script need the following information: Transaction starting and ending
blocks,"
echo "journal block size, inodesize, inode entry, and device/dd image to read.
echo "example:
echo "./ejcat 49 4096 128 15 /dev/sda9"
echo "will result in the following command:"
echo "dd bs=$blksize skip=$skipblk count=1 if=$device | dd bs=$isize skip=$skipi
count=1 | xxd"
dd bs=$blksize skip=$skipblk count=1 if=$device | dd bs=$isize skip=$skipi count=1
| xxd
```

The script is named ejcat.sh and stands for external jcat, and the idea is to extract a specific inode from the inode table copy in the journal. This script requires the offset in blocks where there's an entry in the journal with a block that is part of the inode table and holds the inode of interest; then the inode and file system block sizes, and finally the inode entry and device/image to read.

To make it clear and validate its use let's resolve the same problem described in section 3 (page14-15), where we were recovering a file whose inode was 144015, as we discussed before the block in the inode table where inode 144015 is located was block 295095 which is the beginning of the inode table and the inode 144015 it's the 15th entry in the table and we need to retrieve that inode.

```
[root@Akula1 workbench1]# dd bs=4096 count=10000 if=sda6img.dd | xxd | awk -v
block=1 -f journal.awk
Signature c03b3998
Options: block=1
Rec: JBDblkoff:  Description:
0    689          Superblock Ver 2 (seq 0)
1    690          Descriptor block (seq 2)
2    691          FS Block 183
3    692          Commit block (seq 2)
4    693          Descriptor block (seq 3)
```

```
5    694          FS Block 295094
6    695          FS Block 1
7    696          FS Block 295095
8    697          FS Block 295093
9    698          FS Block 295595
10    699          FS Block 0
11    700          FS Block 303104
12    701          FS Block 183
[REMOVED]
[root@Akula1 workbench]# sh ejcat.sh 696 4096 128 15 sda6img.dd
[REMOVED]
0000000: a481 0000 ca94 0800 b1b3 8246 6c13 2e47  ...........Fl..G
0000010: dd96 8246 0000 0000 0000 0100 6004 0000  ...F........`...
0000020: 0000 0000 0000 0000 9abb 0400 9bbb 0400  ................
0000030: 9cbb 0400 9dbb 0400 9ebb 0400 9fbb 0400  ................
0000040: a0bb 0400 a1bb 0400 a2bb 0400 a3bb 0400  ................
0000050: a4bb 0400 a5bb 0400 a6bb 0400 0000 0000  ................
0000060: 0000 0000 6cba c6da ab82 0400 0000 0000  ....l...........
0000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

If we compare this result with the output on page 15 it is the same result, and now the inode can be decoded using the same way technique as we did in section 3.

# 5. Time Machine Reloaded: File Historical Activity with Ext3 Journal

As we are going to discuss in this section there is an advantage when we are talking about MACtime. (`mtime, atime, ctime, dtime`) in an Ext3 file system. Just to refresh the memory the MACtimes are the time values on every file on Ext2/Ext3. Here is a small description of each one:

- `Mtime` (Modify time): This value is updated when the content of a file/directory is changed
- `Atime` (Access time): This value is updated every time the content of a file/directory is read, or copied or moved to a new volume.
- `Ctime` (Change time): This value represents the last time the metadata (inode) of a file has changed. Events that can trigger an update of this value are the creation of a file and when the permissions or ownership of a file are changed.
- `Dtime:` (Delete time): This value is set only when the file is deleted, and cleared when the inode is allocated.

By analyzing a file's MACtimes it's possible to reconstruct the activity of such file within the file system. For example when a file is created the mtime, atime and ctime are updated with the time of creation and dtime is set to zero, and the parent directory's mtime and ctime are updated. Another good example is when a file is deleted; the mtime, atime and ctime are the same as the dtime which is set with the time of deletion.

On Ext2 this values represents a good piece of evidence that could give the forensic analyst a timeline of the events that occurred during an incident, unfortunately in Ext2, MACtimes only shows the most recent activity ( the last) and any prior indication is lost. But in an Ext3 file system the journal becomes an historical file activity archive,

As we have learned the journal at least records the metadata changes, in other words inodes. And as we all know inodes stores among other things the MACtimes of a file. Hence another opportunity arises to take advantage of the journal. If we could recover all the inode copies for a particular file, then we could see the historical activity of that file, not only the last one [Farmer & Venema. 2007].

One tool that is quite helpful is debugfs, traditionally this tool has been used for data recovery in a corrupted file system; but let's see how this tool can help us with this particular task. For this part we use an Ext3 file system on sda6 and we take a look on the file "reference.pdf". To maintain evidence integrity it is advisable to run debugfs with the `-c` option, which makes to act with the file system or image in a read only mode.

If we execute `debugfs -c -R 'logdump -i <144015>' sda6img.dd | grep atime` the debugfs will show us al the metadata related to a file named reference.pdf whose inode was 144015 (example from section 3) on /dev/sda6 image, then this output is filtered using grep to look for the atime, in the same way we obtained ctime, mtime and dtime.

```
[root@Akula1 workbench]# debugfs -c -R 'logdump -i <144015>' sda6img.dd|grep atime
debugfs 1.38 (30-Jun-2005)
sda6img.dd: catastrophic mode - not reading inode or group bitmaps
    atime: 0x4682b3b1 -- Wed Jun 27 14:00:01 2007
```

```
    atime: 0x4682b3b1 -- Wed Jun 27 14:00:01 2007
    atime: 0x472e1ad0 -- Sun Nov  4 13:17:36 2007
    atime: 0x472e1ad0 -- Sun Nov  4 13:17:36 2007
    atime: 0x472e1ad0 -- Sun Nov  4 13:17:36 2007
    atime: 0x472e1ad0 -- Sun Nov  4 13:17:36 2007
    atime: 0x472e88ae -- Sun Nov  4 21:06:22 2007
    atime: 0x472e88ae -- Sun Nov  4 21:06:22 2007
[root@Akula1 workbench]# debugfs -c -R 'logdump -i <144015>' sda6img.dd|grep mtime
debugfs 1.38 (30-Jun-2005)
sda6img.dd: catastrophic mode - not reading inode or group bitmaps
    mtime: 0x46829bdd -- Wed Jun 27 11:57:01 2007
    mtime: 0x46829bdd -- Wed Jun 27 11:57:01 2007
    mtime: 0x46829bdd -- Wed Jun 27 11:57:01 2007
    mtime: 0x46829bdd -- Wed Jun 27 11:57:01 2007
    mtime: 0x46829bdd -- Wed Jun 27 11:57:01 2007
    mtime: 0x46829bdd -- Wed Jun 27 11:57:01 2007
    mtime: 0x46829bdd -- Wed Jun 27 11:57:01 2007
    mtime: 0x472e88ba -- Sun Nov  4 21:06:34 2007
[root@Akula1 workbench]# debugfs -c -R 'logdump -i <144015>' sda6img.dd|grep ctime
debugfs 1.38 (30-Jun-2005)
sda6img.dd: catastrophic mode - not reading inode or group bitmaps
    ctime: 0x472e136c -- Sun Nov  4 12:46:04 2007
    ctime: 0x472e136c -- Sun Nov  4 12:46:04 2007
    ctime: 0x472e136c -- Sun Nov  4 12:46:04 2007
    ctime: 0x472e136c -- Sun Nov  4 12:46:04 2007
    ctime: 0x472e136c -- Sun Nov  4 12:46:04 2007
    ctime: 0x472e136c -- Sun Nov  4 12:46:04 2007
    ctime: 0x472e136c -- Sun Nov  4 12:46:04 2007
    ctime: 0x472e88ba -- Sun Nov  4 21:06:34 2007
[root@Akula1 workbench]# debugfs -c -R 'logdump -i <144015>' sda6img.dd|grep dtime
debugfs 1.38 (30-Jun-2005)
sda6img.dd: catastrophic mode - not reading inode or group bitmaps
    dtime: 0x472e88ba -- Sun Nov  4 21:06:34 2007
```

Now we can see when the activity on the file "reference.pdf", this output shows that the file was accessed several times during Sunday noon of November 4 (atime), and finally deleted the same day at 21:06 (`dtime`). Note how dtime only appears once and mtime, atime and ctime are the same as dtime when the deletion occurred. All this information was recovered from all the copies that are inside the journal. If the file system involved was an Ext2 instead of Ext3 we only could have observed the last set of MACtimes, in this case when the file was deleted, and miss the prior access events of the file.

Unfortunately there are some limitations with the use of debugfs that we must have present:

- The standard version of debugfs only permits to check one file at the time.
- Sometimes debugfs does not recognize the end of the journal and starts throwing garbage.
- Debugfs does not recognize an external journal.

For the first problem, a patch exists for debugfs [Farmer & Venema. 2004] that allows display all the MACtimes for multiple files, for the third one even though it's not possible to use it against an external journal, it's feasible to develop a program or script to get the MACtimes, as previously demonstrated with the journal.awk script used to read the transactions from an external journal.

# 6. Conclusions

One benefit of knowing the internals of a file system is that it allows analysis without specialized tools at hand or we can develop our own if we find a scenario where the actual tools are not working. An example is the case of an external journal.

As the research demonstrates Ext3 file systems offers several possibilities that are not available with Ext2 in terms of forensic evidence:

- File recovery based on the metadata copies from the journal.
- Historical file activity; now it is possible to see repeated activity of files across time as demonstrated with the use of debugfs.
- It is possible to analyze an external journal, if the analyst has access to the internal structures that make up the journal.

But as all things in life everything comes with a compromise. The journal due its cyclic nature tends to overwrite itself either because the device is being remounted or because it runs out of space and starts using the beginning of the journal to keep recording changes. This makes the life expectancy of the content of the journal very short, especially if we consider that the size of the journal in most cases it's fixed to 128 MB, keep in mind that in theory the maximum size can be 102400 file system blocks or 400MB if block size is 4096 bytes. That makes the use of the journal useful for cases where the time of the incident is recent. A possible exception to this is when the file system under examination has its journal on an external device. In this scenario that external device in theory could be of any size (The lab external journal sda9 has a size of 2.76GB) giving a little bit of margin to maneuver to a forensic analyst.

The script journal.awk demonstrated that the internal structure of the journal it's mostly the same as in the internal version, but the script is in early stages of development and needs a more through testing, also there are improvements to be made as showing the list of file system block being revoked. In the area of performance, because it's a script and not a program written on c when dealing with large amounts of data in the magnitude of Gigabytes, it becomes extremely slow. This was tested against a 2.76GB external journal.

And finally at least in one scenario (external journal) tools like TSK will not work at all, leaving the forensic investigator to look for other venues. This is resumed in the following table:

**Table 6.1: Functionality of tools against different sources**

| *Tool* | *Ext3 FS internal journal (device)* | *Ext3 FS external journal (device)* | *External journal (device)* | *Ext3 FS internal journal (image)* | *Ext3 FS external journal (image)* | *External journal (image)* |
|---|---|---|---|---|---|---|
| fsstat (TSK) | Y | Y | N (note 1) | Y | Y | N (note 1) |
| ils (TSK) | Y | Y | N (note 1) | Y | Y | N (note 1) |
| img_cat (TSK) | Y | Y | Y | Y | Y | Y |
| jls (TSK) | Y | N (note 2) | N (note 1) | Y | N (note 1) | N (note 3) |
| jcat (TSK) | Y | N (note 3) | N (note 1) | Y | N (note 3) | N (note 1) |
| dcat(TSK) | Y | Y | N (note 1) | Y | Y | N (note 1) |
| dstat(TSK) | Y | Y | N (note 1) | Y | Y | N (note 1) |
| dumpe2fs | Y | Y | Y | Y | Y | Y |
| tune2fs | Y | Y | N (note 4) | Y | Y | N (note 4) |
| debugfs | Y | Y | Y | Y | Y | Y |

Note 1:Cannot determine file system type
Note 2: Inode value is too small for image (1)
Note 3: Invalid walk range (ext2fs_jblk_walk: end is too large )
Note 4: tune2fs: Filesystem has unsupported feature(s) while trying to open /dev/sda9
Couldn't find valid filesystem superblock.
Note 5: TSK ver 2.09

There is still a lot of thing to do in terms of forensic research on Ext3, among those we could mention three of them:

- Analysis and forensic impact of the other two Ext3 modes: journaled and write back.
- Data hiding on Ext3 journals. Even doe this type of data will have a very limited life span due the cyclic nature of the journal, its possible to store some information in the last bytes of the administrative blocks, specially the commit for a transaction and the journal super block.
- Development of a tool that helps to automate the collection of historical activity contained in the inode copies in the journal (MACtimes)

In the hope that this research becomes useful, I wish a happy hunting to all forensic practitioners.

# 7. References

Carter, B. (2005). File system forensic analysis. *Ext2 and Ext3 Concepts and Analysis*, 437-441.

Carter, B. (2005). Why Recovering a Deleted Ext3 File Is Difficult . . .and why you should back up important files. Retrieved June 3, 2007 from http://linux.sys-con.com/read/117909.htm

Farmer, D., & Wietse, V (2004). Forensic Discovery. *Journaling File Systems and MACtimes*, 31-34

Farmer, D., & Wietse, V (2007). Forensic Discovery. *19th Annual FIRST Conference*, Retrieved August 26, 2007 from http://www.first.org/conference/2007/program/presentations.html

Linux Ext3 FAQ (2004) Retrieved March 25, 2007 from Linux Ext3 FAQ (2004) Retrieved March 25, 2007 from http://batleth.sapienti-sat.org/projects/FAQs/ext3-faq.html-

Sovani, K. (2006). Linux: The Journaling Block Device. Retrieved July 15, 2007 from http://kerneltrap.org/node/6741

Ts'o, T. (2002). mke2fs man page. Retrieved July 20, 2007 from http://www.netadmintools.com/html/8mke2fs.man.html