SANS Digital Forensics and Incident Response Blog | Understanding EXT4 (Part 1): Extents

Hal Pomeranz, Deer Run Associates

EXT4 is a next generation file system replacement for the EXT2/EXT3 family of Linux file systems. It was accepted as "stable" in the Linux 2.6.28 kernel in October 2008[1]. As of this writing, it's starting to appear as the default file system in newer versions of several Linux distros. While the developers did try to maintain some degree of backwards compatibility with EXT2/EXT3, there is quite a bit that's new and different with EXT4. Popular forensic tools like the Sleuthkit are not fully compatible with these changes in EXT4, although some of their functionality does still work.

While I had read some of the presentations[2] related to EXT4, I was curious about how the EXT4 structures actually looked on disk and how and why the changes made in the EXT4 file system broke existing forensic tools. So I decided to fire up the old hex editor and see for myself. This is the first in a series of articles that describes my findings.

Block Addressing

EXT4 has moved to 48-bit block addresses. I'll refer you to the paper cited above for the whys and wherefores of this decision and what it means as far as maximum file system size, etc. What's really a departure for EXT4 however, is the use of *extents* rather than the old, inefficient indirect block[3] mechanism used by earlier Unix file systems (e.g. EXT2/EXT3) for tracking file content. Extents are similar to cluster runs in the NTFS file system- essentially they specify an initial block address and the number of blocks that make up the extent. A file that is fragmented will have multiple extents, but EXT4 tries very hard to keep files contiguous.

This new block addressing strategy is one of the things that causes the most problems for existing forensic tools. For example, look what happens when I create a new file in the EXT4 file system on my laptop and then try to use istat from the Sleuthkit against it:

```
# echo Here is a new file >testfile
# ls -li testfile
918817 -rw-r--r-- 1 root root 19 2010-12-05 11:08 testfile
# istat /dev/mapper/elk-home 918817
inode: 918817
Allocated
Group: 112
Generation Id: 3173542730
uid / gid: 0 / 0
mode: rrw-r--r--
Flags:
size: 0
num of links: 1

Inode Times:
Accessed:         Sun Dec  5 11:08:49 2010
File Modified:    Sun Dec  5 11:08:49 2010
Inode Modified:   Sun Dec  5 11:08:49 2010

Direct Blocks:
```

istat is completely unable to decode the new extent structures in the inode, and so no block addresses are displayed. If you look closely at the output above, you'll also see that the file size is reported as zero bytes, which is clearly wrong. On the other hand, many of the other values from the inode metadata appear to be correct- owner, group owner, MAC times, etc.

In fact, the EXT4 developers tried very hard to make the EXT4 inode as backwards compatible as possible with the EXT2/EXT3 inode structure. But changes like extents, new and higher-resolution timestamps, et al have required some incompatible changes.

Unpacking the EXT4 Inode

I really wanted to look at the EXT4 inode with my hex editor, but that meant figuring out exactly where on disk the inode for this file resides. Fortunately the superblock and block group descriptor tables in EXT4 are compatible enough to let fsstat give us the information we need:

```
# fsstat /dev/mapper/elk-home
FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: Ext3
[...]
CONTENT INFORMATION
--------------------------------------------
Block Range: 0 - 113971199
Block Size: 4096
Free Blocks: 13506529

BLOCK GROUP INFORMATION
--------------------------------------------
Number of Block Groups: 3479
Inodes per group: 8192
Blocks per group: 32768
[...]
Group: 112:
 Inode Range: 917505 - 925696
 Block Range: 3670016 - 3702783
 Layout:
 Data bitmap: 3670016 - 3670016
 Inode bitmap: 3670032 - 3670032
 Inode Table: 3670048 - 3670559
 Data Blocks: 3670033 - 3670047, 3670560 - 3702783
 Free Inodes: 3281 (40%)
 Free Blocks: 0 (0%)
 Total Directories: 2
[...]
```

We know from the istat output above that our inode resides in block group 112. You can also confirm this by looking at the fsstat output for group 112 and seeing that our inode addess, 918817, falls within the range of inodes for this group.

One of the important changes in EXT4 is that inodes are now 256 bytes, as opposed to 128 bytes as they were in the EXT2/EXT3 days. That means there are 16 inodes per 4K block in EXT4, so the 8192 inodes per block group should occupy 512 blocks at the beginning of each group. And you can see that the inode table for our block group occupies the 512 blocks from 3670048 - 3670559. So that all checks out as expected.

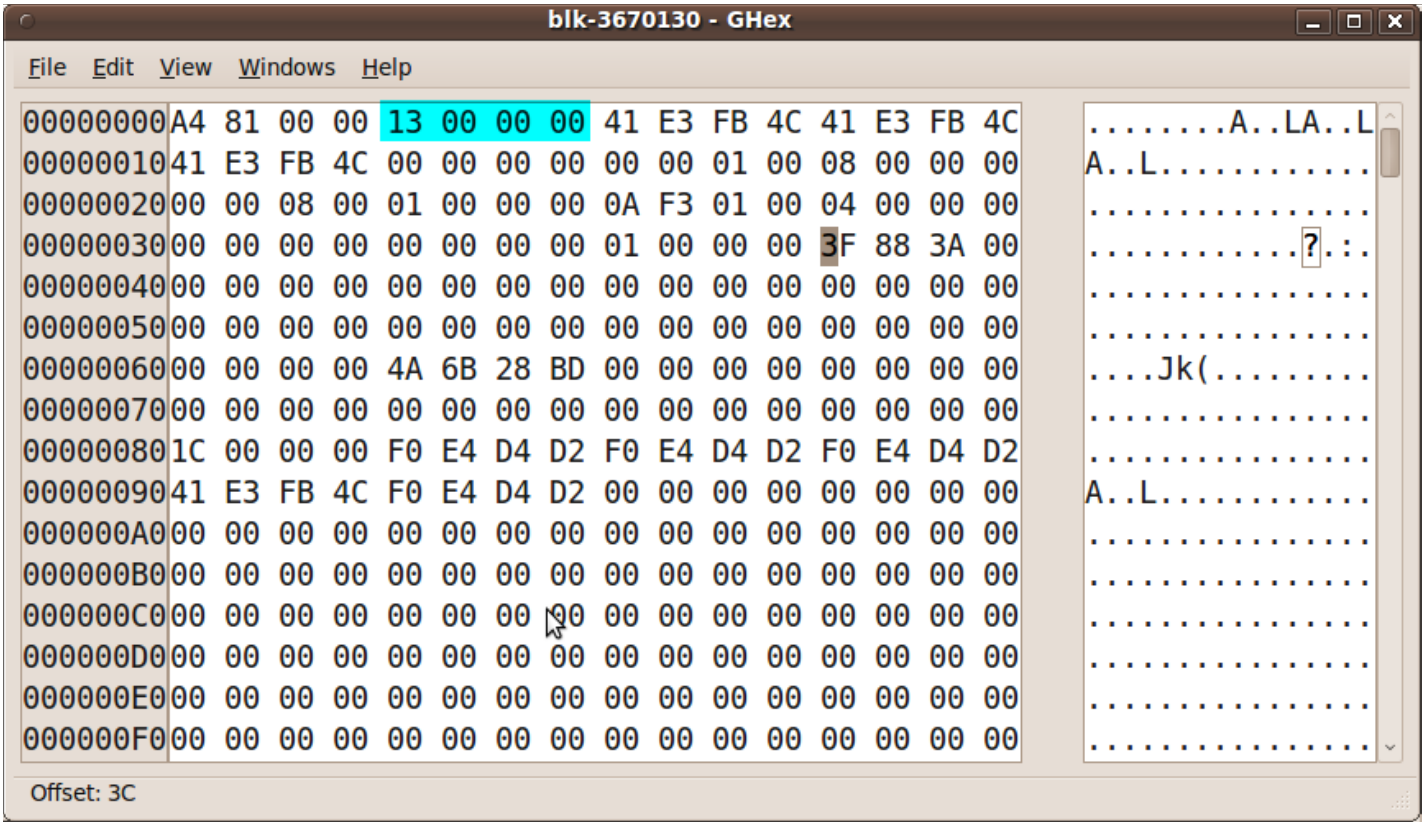
But which block is our inode in? The first inode address in Group 112 is 917505. Subtracting this value from 918817, we find that our inode is 1312 inodes in from the beginning of the inode table. Happily, this puts us right at the beginning of a block- 1312 inodes divided by 16 inodes/block puts us exactly 82 blocks into the inode table. If the first block address of the inode table is 3670048, we should find our inode in the first 256 bytes of block 3670130.

I'll use blkcat to dump out this block so that we can more easily look at it in a hex editor:

```
# blkcat /dev/mapper/elk-home 3670130 >blk-3670130
```

The EXT4 Inode Under the Microscope

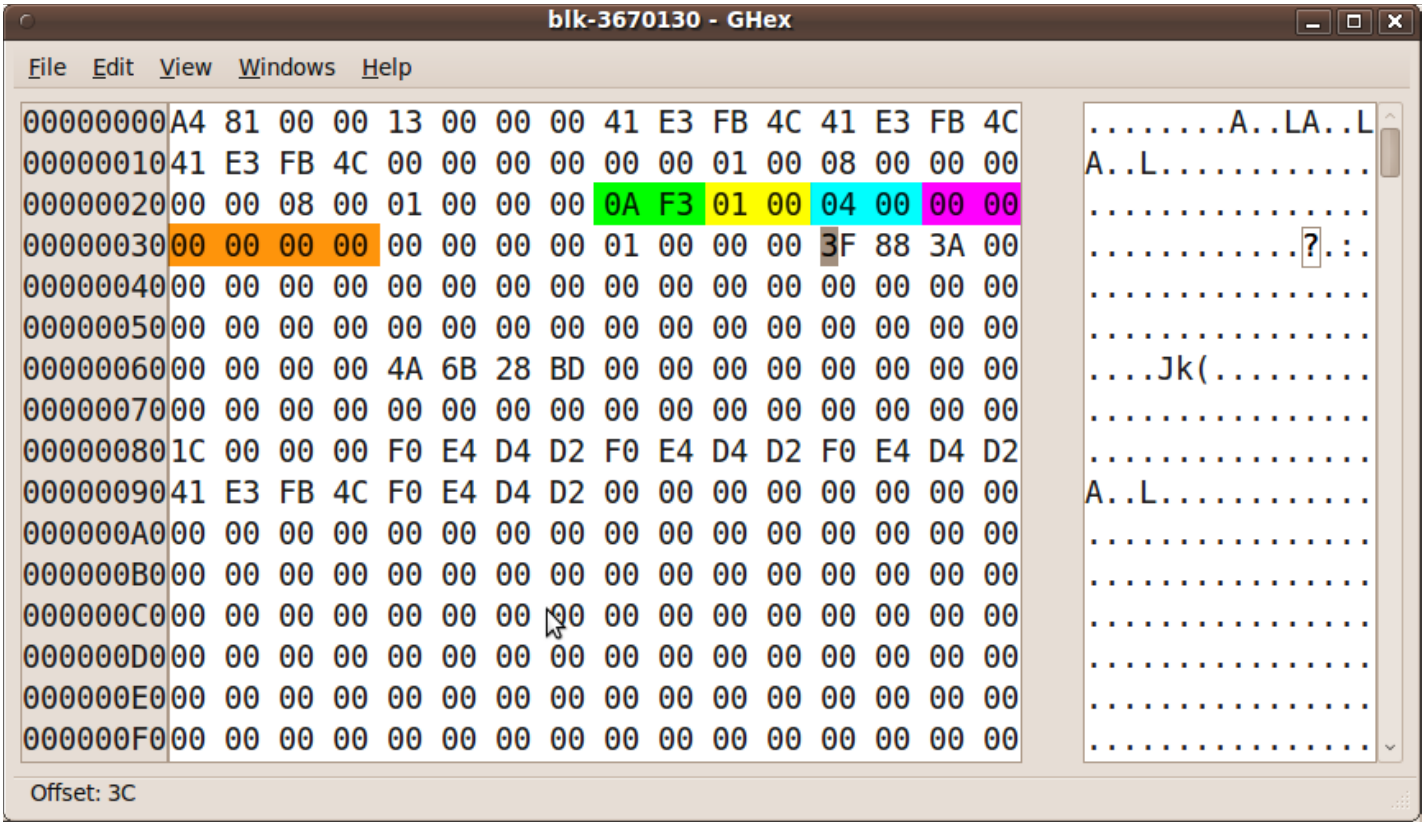
While the new EXT4 inode is double the size of the older EXT3 inode structure, the EXT4 developers tried as much as possible to not alter the way the fields in the first 128 bytes of the inode are used. So for example, you will still find the low-order 32 bits of the file size in bytes 4-7[4]:



In little-endian, we interpret this to mean that our file size is 19 bytes, which is the file size we were expecting. It seems like we've found the correct inode!

However, because EXT4 uses extents rather than block pointers to track the file content, the 60 bytes from 40-99 that used to contain the block pointers are now used to hold extent information. Extent structures are 12 bytes in size, so you would expect there to be a maximum of 5 extents in each inode. However, the first 12 bytes of the extent area (bytes 40-51) are occupied by an *extent header* structure, so the number of extents that may be contained in an inode is actually 4.

The values in the extent header are broken out as follows:



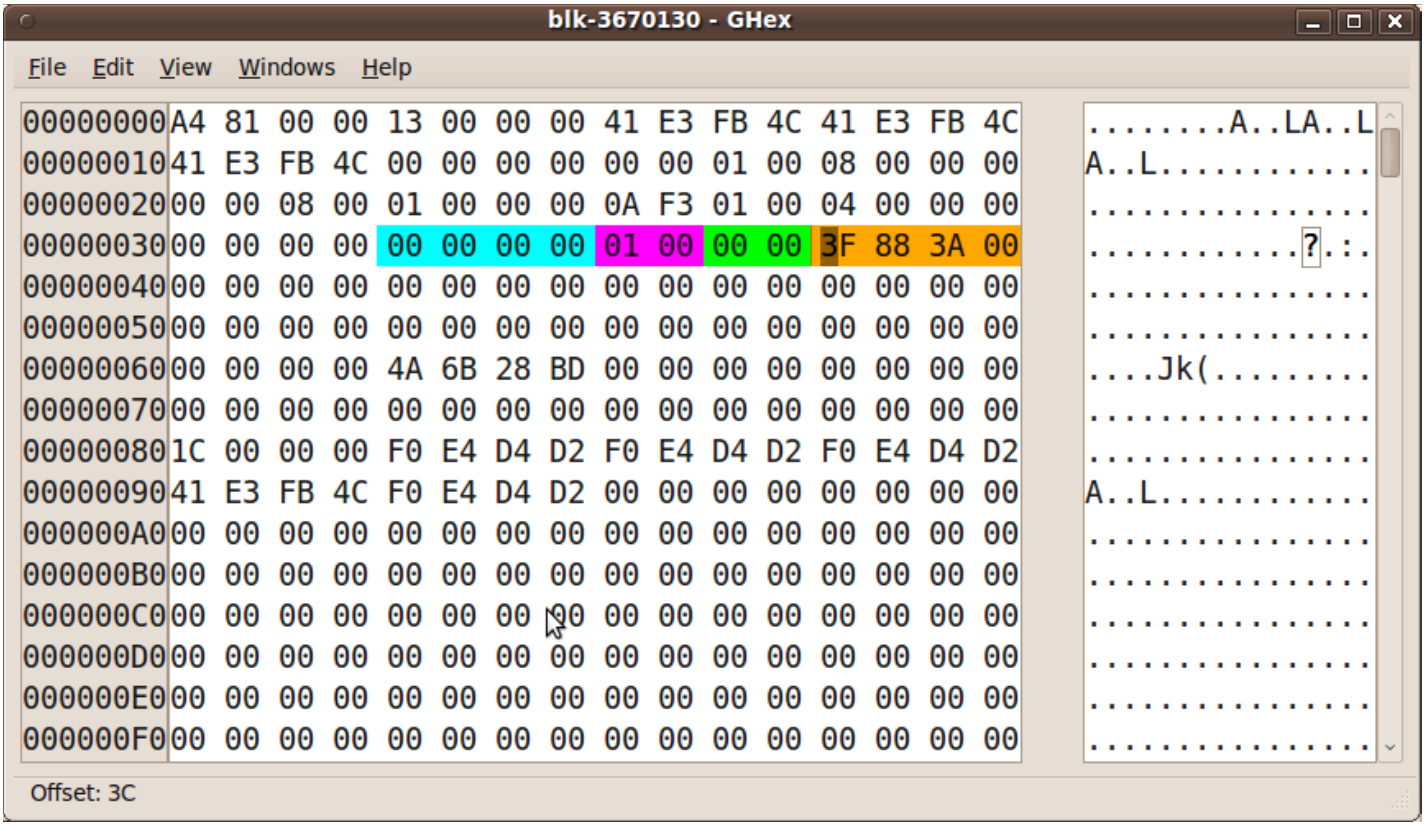
Broken down by byte offset, we have:

```
Bytes 40-41: Magic number (0xF30A = 62218)
      42-43: Number of extents (0x0001 = 1)
      44-45: Max number of extents (0x0004 = 4)
      46-47: Depth of tree (0x0000 = 0)
      48-51: Generation ID (0x00000000 = 0)
```

The magic number is designed to differentiate between different extent implementations. As new features are added, the magic number can change to ensure backwards compatibility with older implementations. We will discuss the "Depth of tree" and "Generation ID" values in future articles in this series.

Per our previous discussion, the maximum number of extents in the inode is 4, and bytes 44-45 document this. In the future, of course, the implementers may elect to store additional extent structures in the inode, so burning two bytes here is a place-holder to allow future functionality. Bytes 42-43 tell us that this file only has a single extent.

The next 12 bytes tell us what we need to know about this extent:



Again, the byte offset breakdown is:

```
Bytes 52-55: Logical block number (0x0000)
      56-57: Number of blocks in extent (0x0001)
      58-59: Upper 16 bits of physical block address (0x0000)
      60-63: Lower 32 bits of physical block address (0x003A883F)
```

The logical block number tells us where this extent begins relative to the start of the file. This becomes a lot more important when you have multiple extents. But since we only have a single extent in this file, it must start from the beginning of the file which is logical block zero.

Next we have two bytes that tell us how many blocks are included in this extent. It's a small file, so we only need one block.

The next six bytes give us the physical block number of the first block in the extent- i.e., where the extent actually begins on disk. Now modern computer systems want their values to align on 16-, 32-, or 64-bit boundaries and 48-bits is something of a problem. So the 48-bit block address is actually represented as two values: first two bytes giving the upper 16 bits of the block address and then four bytes containing the low-order 32 bits of the address. So in our example, we would interpret the block address as 0x0000003A883F, which is block number 3835967.

Let's see if we're right about that:

```
# blkcat /dev/mapper/elk-home 3835967
Here is a new file
```

I love it when a file system comes together!

Since there are no further extents (per the extent header we unpacked earlier), the next 36 bytes of the inode are null. It would be interesting to experiment and see if these unused fields could be used to hide data.

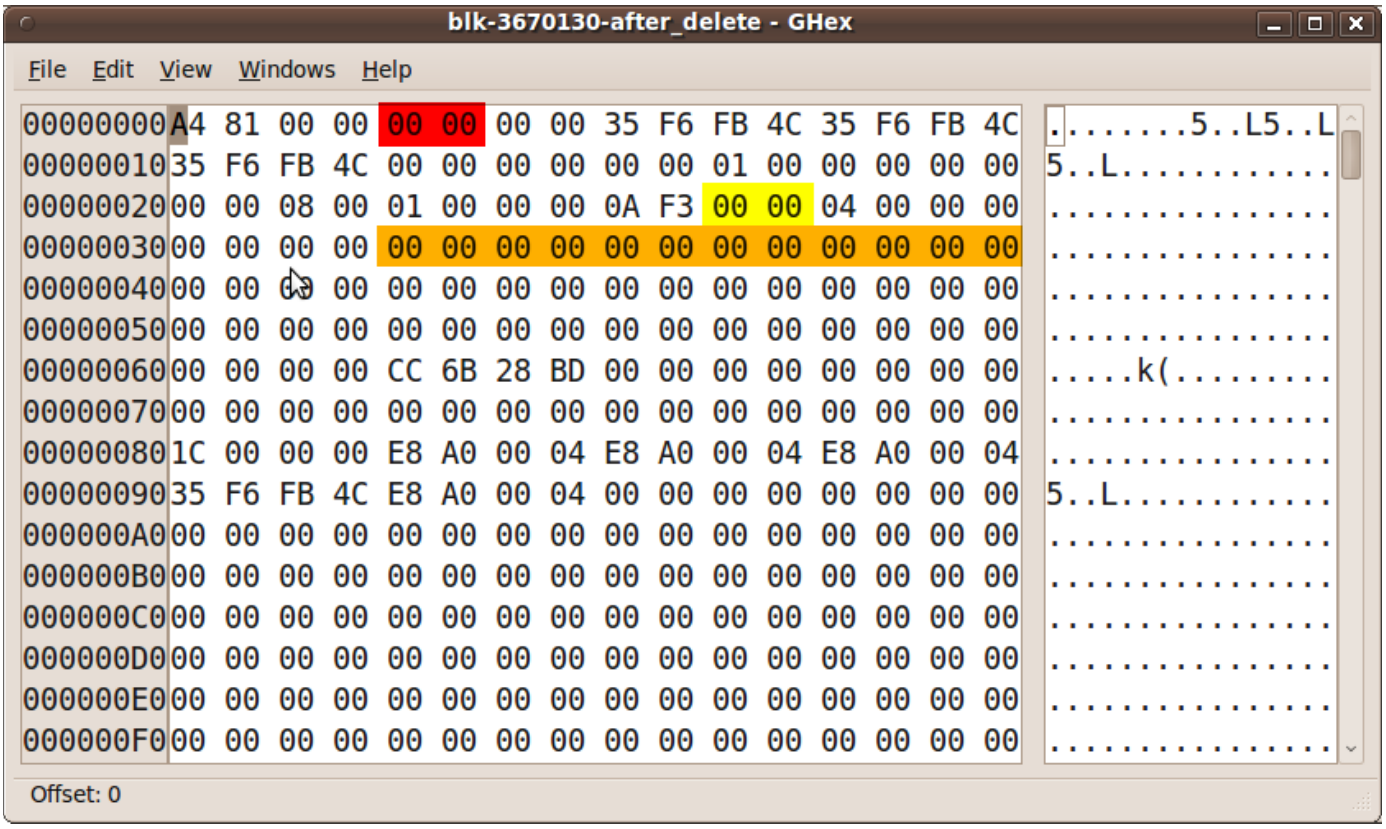
Deleting the File

Let's see what happens in the inode when the file is deleted:

```
# rm testfile
# blkcat /dev/mapper/elk-home 3835967
Here is a new file
# blkcat /dev/mapper/elk-home 3670130 >blk-3670130-after_delete
```

As you can see, the data blocks are not cleared when the file is deleted. This is standard behavior for file systems.

But what happens in the inode when it is deallocated?



There are a number of changes:

- The file size value is set to zero
- The number of extents value in the extent header is likewise zeroed
- The extent itself is also cleared

Clearing the extent means that we lose the physical block address of the first block as well as the length of the extent. In other words, there's no meta-data left in the inode that will help us recover the deleted file. This behavior is analogous to EXT3 clearing the block pointers in the inode when the inode is deallocated. Unfortunately, this means that we're forced to rely on traditional file-carving methods to recover deleted files, which makes life much more difficult.

More to Come

This has been a high-speed introduction to the new and wonderful world of EXT4 extents. But there's more complexity that needs to be discussed. For example, what happens when a file becomes so fragmented that it requires more than the four extent structures that can fit in the inode?

Also, there are some additional values in the EXT4 inode that forced its expansion to 256 bytes. This includes a new timestamp- EXT4 now has a file creation timestamp like NTFS- as well as higher precision timestamps (64-bit values instead of 32-bit). We'll talk about all of this in future episodes!

[1] Wikipedia: "EXT4", http://en.wikipedia.org/wiki/Ext4

[2] "The New EXT4 File System: Current Status and Future Plans", http://www.kernel.org/doc/ols/2007/ols2007v2-pages-21-34.pdf

[3] "Understanding Indirect Blocks in Unix File Systems", http://computer-forensics.sans.org/blog/2008/12/24/understanding-indirect-blocks-in-unix-file-systems/

[4] "File System Forensic Analysis", http://www.amazon.com/System-Forensic-Analysis-Brian-Carrier/dp/0321268172/

Hal Pomeranz is an Independent IT/Security Consultant, a SANS Institute Faculty Fellow, and a GCFA. He has an

unhealthy attachment to hex editors.