SANS Digital Forensics and Incident Response Blog | Understanding EXT4 (Part 3): Extent Trees

Hal Pomeranz, Deer Run Associates

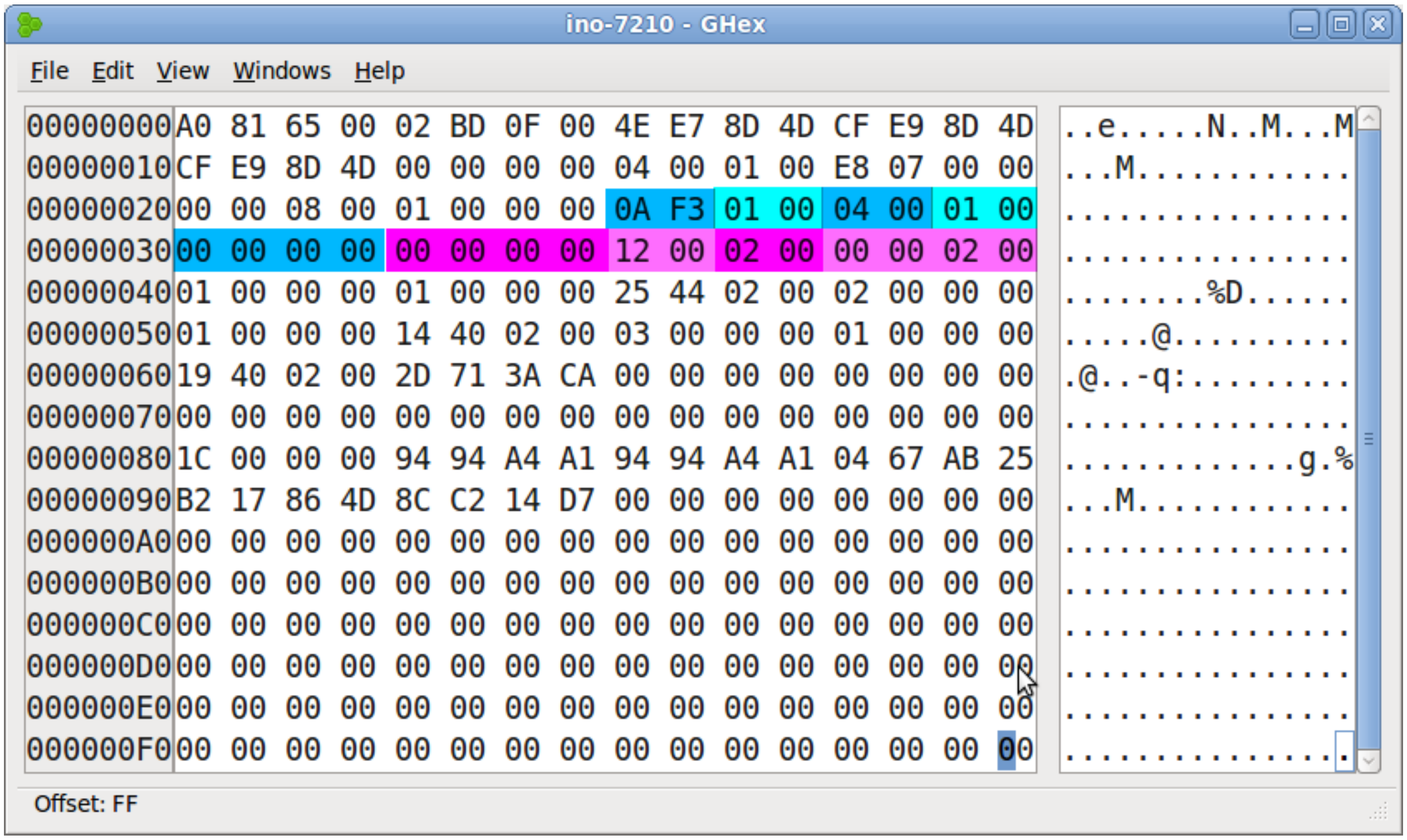
There's one more big concept we need to cover before you can really start decoding EXT4 file systems. As I mentioned in Part 1 of this series, you can only have a maximum of 4 extent structures per inode. Furthermore, there are only 16 bits in each extent structure for representing the number of blocks in the extent, and in fact the upper bit is reserved (it's used to mark the extent as "reserved but initialized", part of EXT4's pre-allocation feature). That means each extent can only contain a maximum of 2^15 blocks- which is 128MB assuming 4K blocks.

Now 128MB is pretty big, but what happens when you have a file that's bigger than half a gigabyte? Such a file would require more than 4 extents to fully index. Or what happens when you have a file that's small but very fragmented? Again, you could need more than 4 extents to represent the collections of blocks that make up the file.

In this installment, I'm going to show you exactly what happens in this case. For our example, I'm going to use the /var/log/messages file on my machine. This file has had data appended to it periodically over the course of a week and has become rather fragmented. Log files often show this behavior if they're in a shared log directory with other files that are being updated in the same manner.

Looking at the Inode

Using the procedure I outlined in Part 1, I dumped the inode associated with the /var/log/messages file and opened it in my trusty hex editor. In the picture below I've shaded the extent header in blue and the first extent structure in purple:



```
00000000 A0 81 65 00 02 BD 0F 00 4E E7 8D 4D CF E9 8D 4D  ..e.....N..M...M
00000010 CF E9 8D 4D 00 00 00 00 04 00 01 00 E8 07 00 00  ...M............
00000020 00 00 08 00 01 00 00 00 0A F3 01 00 04 00 01 00  ................
00000030 00 00 00 00 00 00 00 00 12 00 02 00 00 00 02 00  ................
00000040 01 00 00 00 01 00 00 00 25 44 02 00 02 00 00 00  ........%D......
00000050 01 00 00 00 14 40 02 00 03 00 00 00 01 00 00 00  .....@..........
00000060 19 40 02 00 2D 71 3A CA 00 00 00 00 00 00 00 00  .@..-q:.........
00000070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000080 1C 00 00 00 94 94 A4 A1 94 94 A4 A1 04 67 AB 25  .............g.%
00000090 B2 17 86 4D 8C C2 14 D7 00 00 00 00 00 00 00 00  ...M............
000000A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

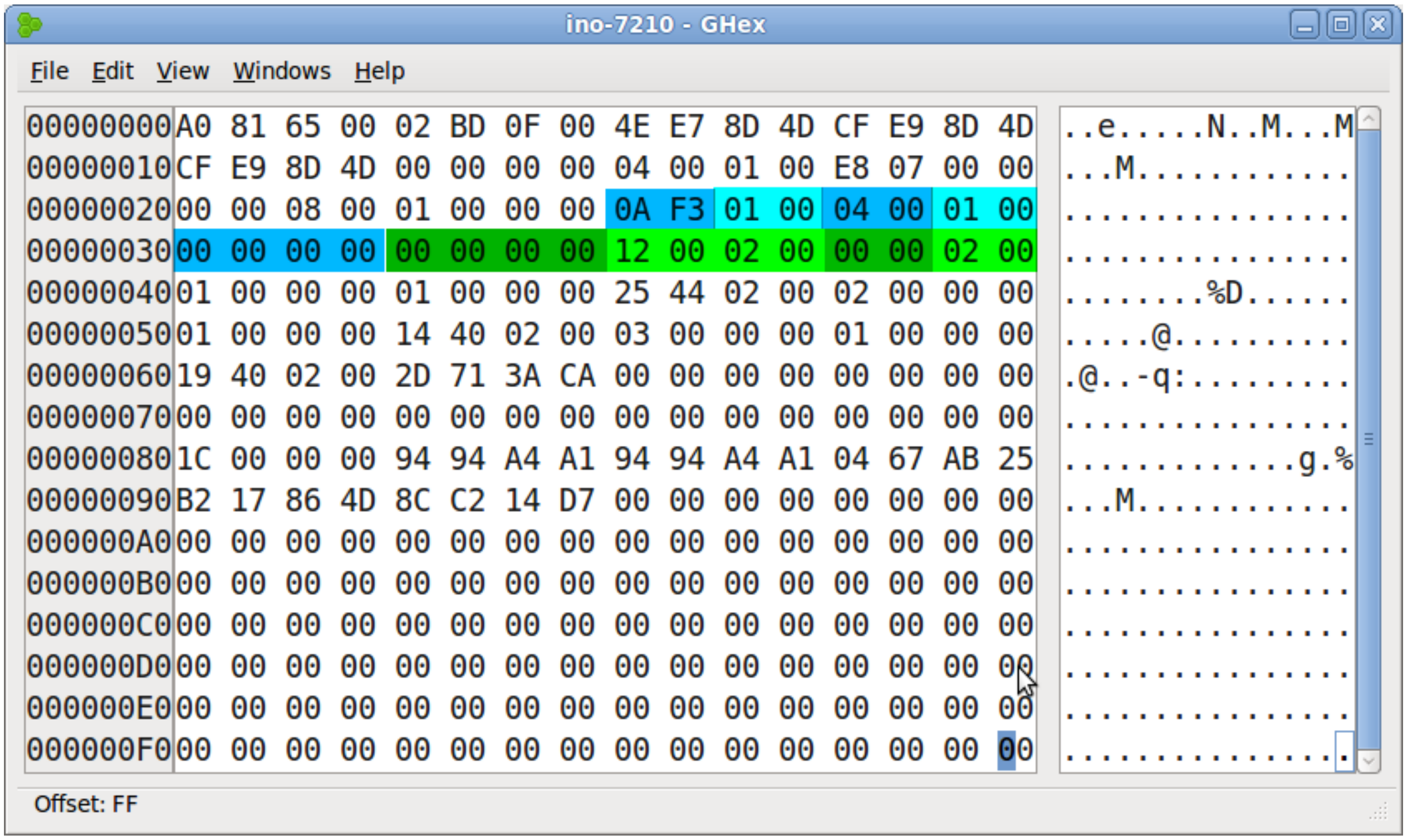
Offset: FF

Looking at the extent header, we first see the standard magic number 0xF30A. The next field tells us there is one extent in this inode. Then we see the expected value 4 for the maximum number of extents in this inode. But then we see something different from our example in Part 1: the "depth of tree" field is one, not zero. And finally we have our generation ID of zero as well.

Let's ignore the depth of tree field for a moment and just look at the first extent structure, which our extent header tells us is being used. Decoding the extent structure, we see that it starts at logical block zero, is 0x0012 = 18 blocks long, and starts at block 0x000200020000 = 8590065664. But there is no way that block number can be right, because that would put us over 32TB into the file system, and my /var partition is only 2GB in size. So what's going on?

What's going on is that the first "extent" in this inode is not a standard extent structure at all, so we're decoding the data incorrectly. When EXT4 needs to use more than four extents, it creates a tree structure on disk for holding the necessary extent fields- that's what the "depth of tree" field in the extent header is trying to help us with. The leaf nodes at the very bottom of the tree are regular extent structures like we saw in Part 1. But the "interior" nodes in the rest of the tree are a different kind of structure called an *extent index*. We know we're dealing with an extent index structure here because "depth of tree" is non-zero, so we're not at a leaf node.

So here's a different picture of our inode with the extent index fields marked out in green:



```
00000000 A0 81 65 00 02 BD 0F 00 4E E7 8D 4D CF E9 8D 4D  ..e.....N..M...M
00000010 CF E9 8D 4D 00 00 00 00 04 00 01 00 E8 07 00 00  ...M............
00000020 00 00 08 00 01 00 00 00 0A F3 01 00 04 00 01 00  ................
00000030 00 00 00 00 00 00 00 00 12 00 02 00 00 00 02 00  ................
00000040 01 00 00 00 01 00 00 00 25 44 02 00 02 00 00 00  ........%D......
00000050 01 00 00 00 14 40 02 00 03 00 00 00 01 00 00 00  .....@..........
00000060 19 40 02 00 2D 71 3A CA 00 00 00 00 00 00 00 00  .@..-q:.........
00000070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000080 1C 00 00 00 94 94 A4 A1 94 94 A4 A1 04 67 AB 25  .............g.%
00000090 B2 17 86 4D 8C C2 14 D7 00 00 00 00 00 00 00 00  ...M............
000000A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

And here are the byte offsets and field descriptions:

```
Bytes 52-55: Logical block number (0x0000)
      56-59: Lower 32 bits of physical block address (0x00020012)
      60-61: Upper 16 bits of physical block address (0x0000)
      62-63: Not used
```

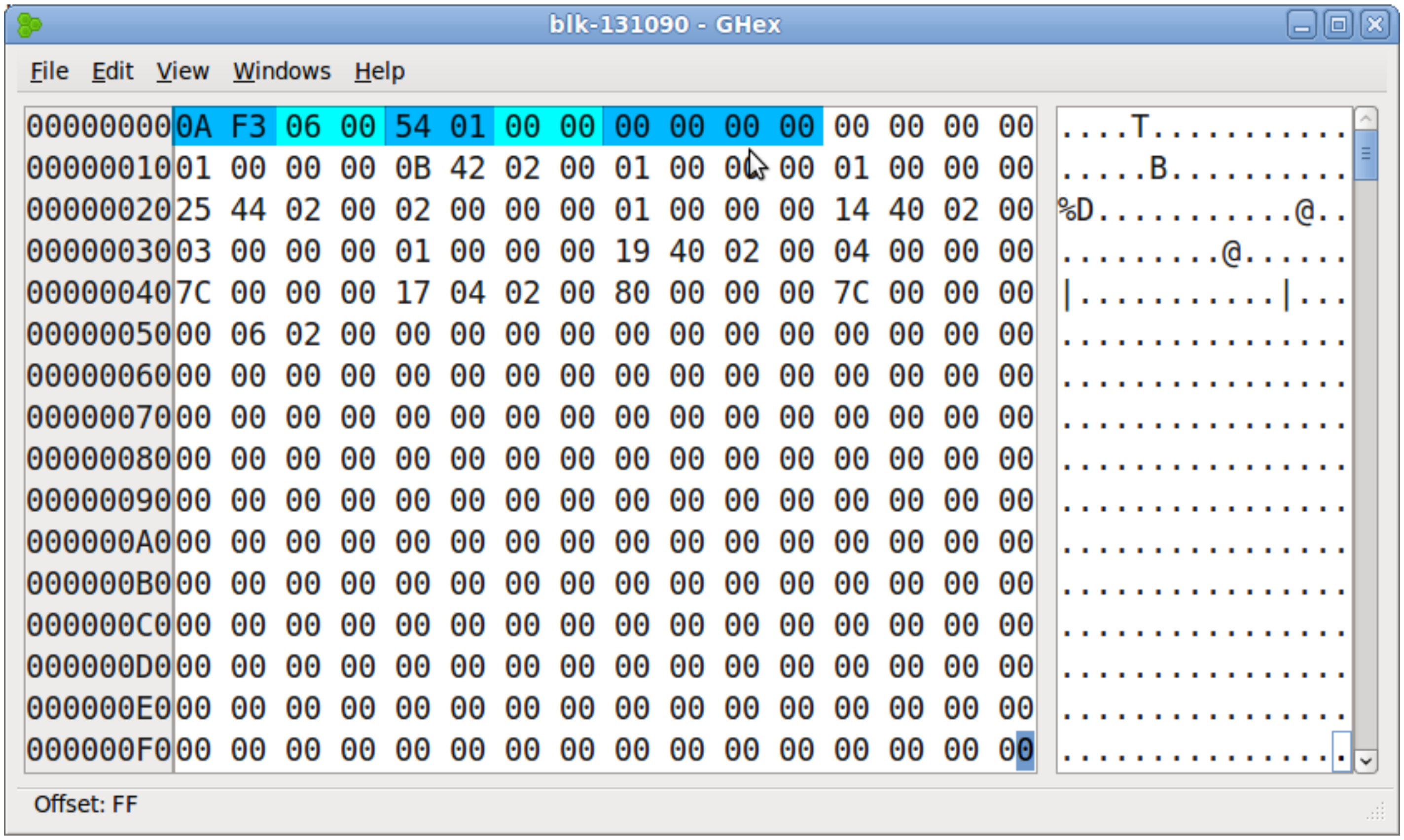
Essentially the extent header contains two values. The first is the logical block where the extents found beneath this node in the tree begin- that's the first four bytes of the extent index structure. In this case the extents in our sub-tree start at logical block zero, meaning the beginning of the file.

The other value in the extent index is the physical block number of a data block that holds the information about the next level in our tree. Like other EXT4 block addresses, this is a 48-bit value broken into two pieces: the 32 low-order bits and the 16 high bits. In our case, this address is 0x000000020012 = 131090, which is a much more sane block offset.

The remaining 16 bits in the extent index are not used. You might expect them to be zero, but in this case we see they're actually set to 0x0002. And in fact, even though the extent header structure in the inode says we're only using the first extent index structure, you can see that the other extent structure fields in bytes 64-99 are likewise non-zero. Why? I'll come back and answer that in a moment, but right now let's check out block 131090.

Decoding the Data Block

Each data block used to store extent tree information begins with its own extent header structure, just like the one we have in the inode. Here's the hex editor view of the first 256 bytes of block 131090 with the extent header fields highlighted:



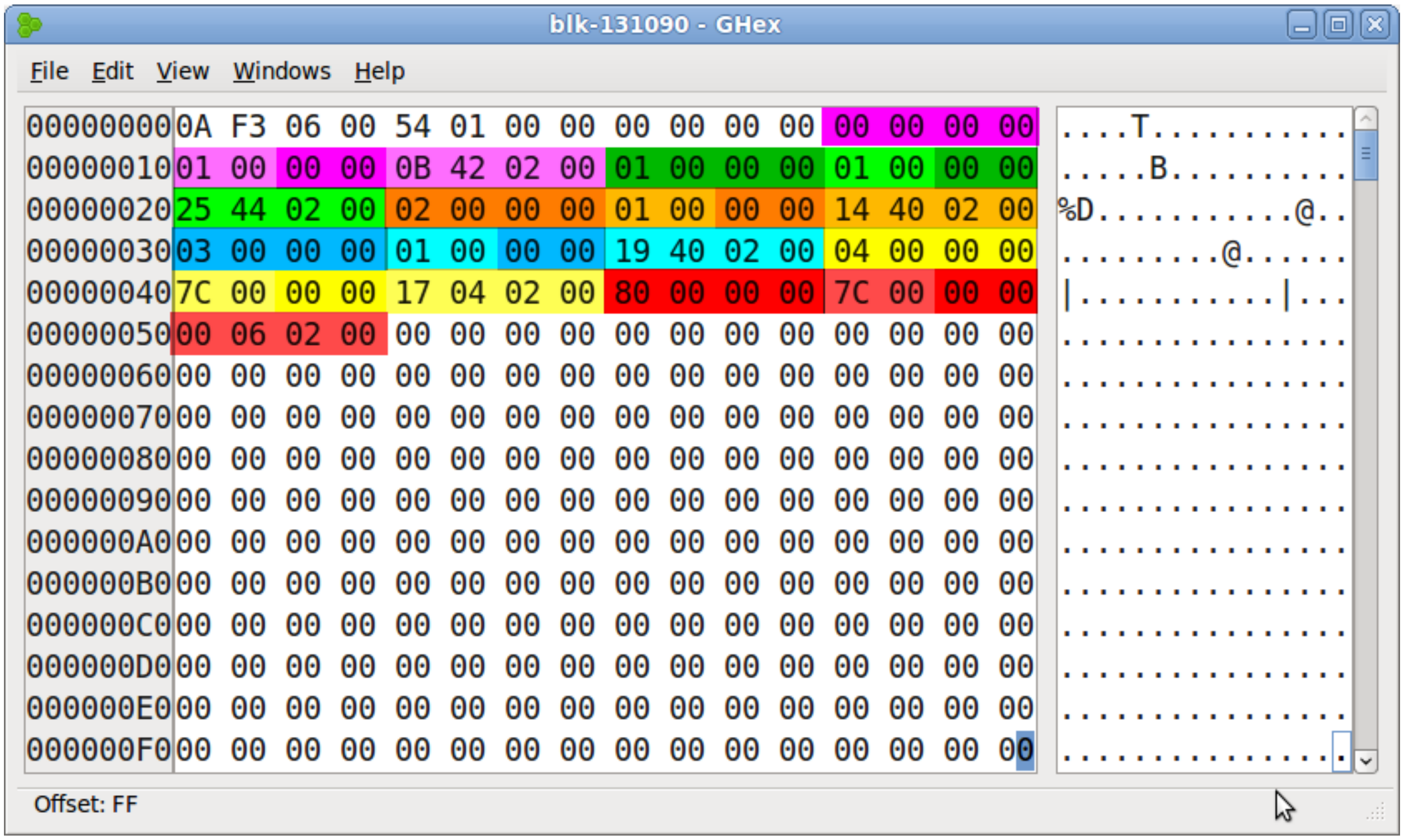
```
0000000000A F3 06 00 54 01 00 00 00 00 00 00 00 00 00 00 ....T...........
0000001001 00 00 00 0B 42 02 00 01 00 00 00 01 00 00 00 .....B..........
0000002025 44 02 00 02 00 00 00 01 00 00 00 14 40 02 00 %D..........@..
0000003003 00 00 00 01 00 00 00 19 40 02 00 04 00 00 00 .........@......
000000407C 00 00 00 17 04 02 00 80 00 00 00 7C 00 00 00 |...........|...
0000005000 06 02 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000006000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000007000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000008000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000009000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000A000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000B000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000C000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000D000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000E000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000F000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
Offset: FF
```

We see the standard "magic number" (0xF30A), and if we look ahead to the "depth of tree field" (bytes 6-7) we can see that this field is zero. So we've moved "down" one level in the tree and any extent structures we find after the header will be regular extents and not extent index structures.

Bytes 2-3 tell us that we actually have 6 extents. But check out the "maximum number of extents" field in bytes 4-5. The value there is 0x0154 = 340! Remember that we've got a full 4K data block to use up here. The extent header at the start of the block consumes 12 bytes, but that still leaves 4084 bytes to hold extent structures. It turns out that 340 12-byte extent structures occupies 4080 bytes of data, so that's the maximum we can pack into the unused space in our block.

It seems very unlikely that under normal operations a file would become so fragmented that it requires more than 340 extents to describe it. And with a maximum size of 128MB per extent, 340 extents lets you address files up to 42.5GB. Larger than that and we'll have to grow our extent tree some more.

But for now, let's decode the extents that we have. I've marked the six extents in the picture below:



```
00000000 0A F3 06 00 54 01 00 00 00 00 00 00 00 00 00 00  ....T...........
00000010 01 00 00 00 0B 42 02 00 01 00 00 00 01 00 00 00  .....B..........
00000020 25 44 02 00 02 00 00 00 01 00 00 00 14 40 02 00  %D...........@..
00000030 03 00 00 00 01 00 00 00 19 40 02 00 04 00 00 00  .........@......
00000040 7C 00 00 00 17 04 02 00 80 00 00 00 7C 00 00 00  |...........|...
00000050 00 06 02 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

See if you can decode the extents without referring to my answers below:

1. Logical Block: 0, Number of Blocks: 1, Starting Block: 147979
2. Logical Block: 1, Number of Blocks: 1, Starting Block: 148517
3. Logical Block: 2, Number of Blocks: 1, Starting Block: 147476
4. Logical Block: 3, Number of Blocks: 1, Starting Block: 147481
5. Logical Block: 4, Number of Blocks: 124, Starting Block: 132119
6. Logical Block: 128, Number of Blocks: 124, Starting Block: 132608

Let's actually see if we decoded things properly:

```
# blkcat /dev/mapper/RD-var 147979 >ext1-blks
# blkcat /dev/mapper/RD-var 148517 >ext2-blks
# blkcat /dev/mapper/RD-var 147476 >ext3-blks
# blkcat /dev/mapper/RD-var 147481 >ext4-blks
# blkcat /dev/mapper/RD-var 132119 124 >ext5-blks
# blkcat /dev/mapper/RD-var 132608 124 >ext6-blks
# cat ext* | tr -d 0 >newmess
# md5sum newmess /var/log/messages
8e8c9445d8ff3e17a22ef5a3034422a9  newmess
8e8c9445d8ff3e17a22ef5a3034422a9  /var/log/messages
```

First I use blkcat to dump out the blocks in each extent per our calculations above. The last extent is going to have some trailing nulls because the file doesn't consume the entire last block, so I use tr to trim the nulls as I'm reassembling the extents into the original file. The md5sum output shows that the "newmess" file I created via dumping the extents manually is exactly identical to the original /var/log/messages file. How cool is that?

One Last Thing

Earlier I pointed out that the unused extent structures in the inode appeared to have data in them. If you look carefully, you'll see that the data in the extent structures in inode bytes 64-99- which would normally hold extents 2-4- exactly matches the data in extents 2-4 in block 131090.

I also pointed out that the upper two bytes of the extent index structure in the inode, which are normally unused, appeared to have some data in them. Again, if you compare, you'll see that the bytes "02 00" in the last two bytes of the extent index match the last two bytes of extent structure #1 in block 131090.

So what happened? Apparently the EXT4 code is a bit lazy. My /var/log/messages file kept growing and fragmenting, so the file system kept adding extents in the inode. The moment it needed the fifth extent, however, it incremented the "depth of tree" value in the extent header and overwrote the first extent with an extent index structure. However, the code didn't bother overwriting the now unused extents in the inodes with zero. In fact, it didn't even bother nulling out the last two unused bytes in the extent index structure! I suppose this is more efficient, but it sure looks messy.

Wrapping Up For Now

There's a bit more research that I want to share with you, but this is more than enough for you to digest right now. But I'll be back again with Part 4 before you know it!

And another gentle reminder for all of you fans of low-level file system analysis: Chad Tilbury and I will be teaching For508 via vLive! starting in June. You can even get a free iPad if you take the course!

Hal Pomeranz is an independent Forensic Consultant, a SANS Institute Faculty Fellow, and a GCFA. He's also too sexy for these inodes by far.