SANS Digital Forensics and Incident Response Blog | Understanding EXT4 (Part 2): Timestamps

Hal Pomeranz, Deer Run Associates

Well I certainly didn't plan on three months elapsing between my last post on EXT4 and this follow-up, but time marches on. That was supposed to be a clever segue into the topic for this installment-- the new timestamp format in the EXT4 inode. OK, I know what you all are thinking now: "Shut up Hal and get to the geeky stuff." Your wish is my command!

Setting the Stage

Let's set up another test file in an EXT4 file ssytem:

```
# echo Time for knowledge >testfile
# touch -a -t 211101231917.42 testfile
# touch -m -t 204005160308.19 testfile
```

I'm using the touch command as root to explicitly set the atime and mtime values so that we have some interesting timestamp values to look at-- otherwise, all of my timestamps would be set to the instant that I created the new file. One thing I want to point out here is that I'm setting both the atime and mtime to a dates far in the future, which in older Unix file systems would normally cause problems because of the 32-bit date rollover issue (or "year 2038 problem" as it's sometimes called).

Now let's compare the output of the standard Linux stat command, the Sleuthkit's istat output, and debugfs' version of stat:

```
# stat testfile
 File: `testfile'
 Size: 19  Blocks: 8 IO Block: 4096 regular file
Device: fc03h/64515d Inode: 6554914 Links: 1
Access: (0644/-rw-r--r--) Uid: ( 0/ root) Gid: ( 0/ root)
Access: 2111-01-23 19:17:42.000000000 -0800
Modify: 2040-05-16 03:08:19.000000000 -0700
Change: 2011-03-12 07:36:13.872411014 -0800
# istat /dev/mapper/RD-home 6554914
inode: 6554914
Allocated
Group: 800
[...]
Inode Times:
Accessed:       Tue Dec 17 12:49:26 1974
File Modified:  Wed May 16 03:08:19 2040
Inode Modified: Sat Mar 12 07:36:13 2011
[...]
# debugfs -R 'stat <6554914>' /dev/mapper/RD-home
[...]
 ctime: 0x4d7b92ed:cfffbe18 -- Sat Mar 12 07:36:13 2011
 atime: 0x0954b156:00000001 -- Tue Dec 17 12:49:26 1974
 mtime: 0x845e5913:00000000 -- Wed May 16 03:08:19 2040
crtime: 0x4d7b92e4:148af06c -- Sat Mar 12 07:36:04 2011
[...]
```

I've edited the output to make it easier to focus in on the timestamps. As you can see, all three commands basically agree on the mtime and ctime values. But both istat and debugfs stat have problems interpreting the atime value from 2111, though the stat command gets it right.

You may have also noticed some other new features in EXT4:

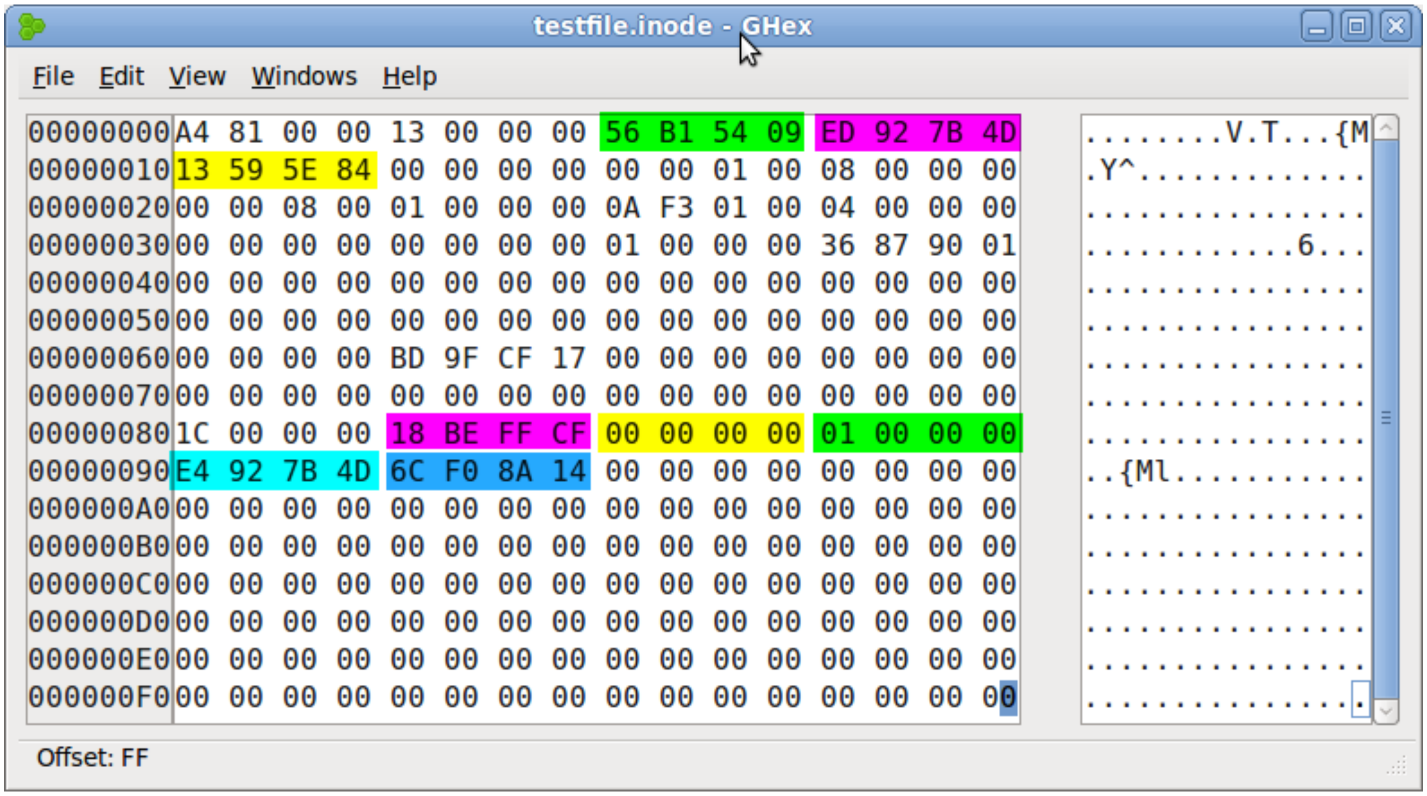
- The stat and debugfs stat commands are reporting fractional timestamp values. EXT4 now supports nanosecond resolution on file timestamps.
- There's a "crtime" (*create time*) timestamp in the debugfs stat output. Yes, EXT4 finally supports a "born on" time (btime) like NTFS, giving the timestamp for when the file was created within the disk volume.
- And for both stat and touch at least, dates in the far future appear to be perfectly acceptable. That implies that EXT4 has also fixed (well, postponed actually) the year 2038 problem.

EXT4 has provided much of this functionality by moving to 64-bit timestamp fields-- this gives you enough resolution to support nanosecond timestamps and still have a couple of bits left over so you can have dates extending into the future.

Show Me the Bits

But as I mentioned in the previous installment, the EXT4 developers tried very hard to maintain backwards compatibility with the EXT2/EXT3 inode layout. 64-bit timestamps and a completely new file creation timestamp obviously complicate this goal. The EXT4 developers solved this problem by putting the extra stuff in the upper 128 bits of the new, larger 256-bit EXT4 inode.

Here's the hex editor view (refer to Part 1 for the process of carving out the correct inode):



```
00000000 A4 81 00 00 13 00 00 00 56 B1 54 09 ED 92 7B 4D  ........V.T...{M
00000010 13 59 5E 84 00 00 00 00 00 00 01 00 08 00 00 00  .Y^.............
00000020 00 00 08 00 01 00 00 00 0A F3 01 00 04 00 00 00  ................
00000030 00 00 00 00 00 00 00 00 01 00 00 00 36 87 90 01  ............6...
00000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000060 00 00 00 00 BD 9F CF 17 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 1C 00 00 00 18 BE FF CF 00 00 00 00 01 00 00 00  ................
00000090 E4 92 7B 4D 6C F0 8A 14 00 00 00 00 00 00 00 00  ..{Ml...........
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
```

Counting from zero at the start of the inode, here are the byte offsets and meanings for the various highlighted fields:

```
Bytes   8 -  11: Access time seconds
       12 -  15: Change time seconds
       16 -  19: Modification time seconds

      132 - 135: Change time "extra"
      136 - 139: Modification time "extra"
      140 - 143: Access time "extra"
      144 - 147: Create time seconds
      148 - 151: Create time "extra"
```

The standard MACtime values near the beginning of the inode are essentially unchanged. They store the number of seconds since Jan 1, 1970 (aka the Unix "epoch"). However, EXT4 treats these values as *unsigned* integers, rather than signed values as the original Unix file system spec called for. Using the extra bit actually turns our "year 2038 problem" into a "year 2106 problem".

You can actually see this in action with our mtime value from the year 2040. Converted from the little-endian format that the inode uses, our hex mtime value is 0x845E5913, or 2220775699 decimal-- a number that would normally cause roll-over in a 32-bit signed value. We can convert this decimal value to a human-readable date with a little shell magic and the GNU date command:

```
# date -d @2220775699
Wed May 16 03:08:19 PDT 2040
```

If you compare with the stat output from earlier, you can see that we've computed the correct value.

But Wait! There's More!

However, in the upper 128 bytes of the inode there are also an "extra" 32 bits for each MAC timestamp-- I've marked them here with corresponding colors, so it's easier to match things up. You can think of the "extra" fields as the nanoseconds portion of each timestamp. But you actually only need 30 bits to represent nanosecond resolution. So the upper 30 bits of the "extra" fields are the nanoseconds, but the lower two bits remaining are actually used to *extend the seconds field* in the earlier portion of the inode.

Confused? Let's do an example with our atime value from 2111. If we just looked at the standard atime bytes 8-11, we'd have a hex value of 0x0954B156, or 156545366 decimal. Let's convert that to a date string:

```
# date -d @156545366
Tue Dec 17 12:49:26 PST 1974
```

You can see now where istat and debugfs stat are going wrong in their output. They're only converting the standard 32-bit atime value from the beginning of the inode.

But look what happens when we add in the low order two bits from the "extra" atime field. The low-order byte of the "extra" atime field is 0x01, or "00000001" in binary. So the low-order two bits are "01". We pre-pend these extra bits on the front of our standard atime value. So the actual atime is 0x010954B156, which is 4451512662 decimal. Now we convert that into a human-readable string:

```
# date -d @4451512662
Fri Jan 23 19:17:42 PST 2111
```

Bingo! There's the correct date that matches our touch command and stat output.

With the extra two bits, the largest value that can be represented is 0x03FFFFFFFF, which is 17179869183 decimal. This yields a GMT date of 2514-05-30 01:53:03, which certainly pushes our date rollover problem quite a bit further out. Still, it's not clear to me that we actually need nanosecond file timestamp resolution. I wish the EXT4 developers had chosen less precision and given me another 2-6 bits to extend the lifetime of our timestamps. On the other hand, by 2514 none of this will probably matter anyway.

Fractional Seconds

The funny packing of the "extra" bits for each timestamp also makes things more complicated when you're interested in the nanosecond component of each timestamp. Let's use the new create time fields as an example. The 64 bits needed for the create time seconds and the create time "extra" field are kept in the upper portion of the new, larger EXT4 inode. They appear right after the "extra" fields for the other timestamps and are highlighted in two shades of blue in the figure above.

The hex value of the create time "extra" is 0x148AF06C, or 344649836. But the low-order two bits are not used for counting nanoseconds. We need to throw those bits away and shift everything right by two bits-- this is equivalent to dividing by 4. So our actual nanosecond value is 344649836 / 4 = 86162459.

We can do the same thing for the change time "extra" field. The hex value 0xCFFFBE18 is 3489644056 decimal. Divide by 4 to get 872411014. If you refer back to the stat output from earlier in the article, you'll see that our computed value matches the fractional seconds that are reported for ctime.

Wrapping Up For Now

At this point we've covered the main EXT4 inode changes that impact most users-- *extents* and the new timestamp format. But the discussion of extents in Part 1 skipped over the issue of what happens when the file becomes so fragmented that you actually need more than the four available extent structures in the standard EXT4 inode. That will be the topic for the next installment of this series.

By the way, if you think this sort of low-level file system analysis is fun and interesting, Chad Tilbury and I will be teaching SANS' For508 class via vLive! starting in June. We spend a lot of time in For508 getting down and dirty with NTFS, FAT, and EXT. So fire up your hex editors and join us on-line from the comfort of your home or office for some wacky file system fun!

Hal Pomeranz is an independent Forensic Consultant, a SANS Institute Faculty Fellow, and a GCFA. He's getting a little squinty from all the time with the hex editors.