Righteous IT *Join the crusade!*

XFS (Part 1) – The Superblock

Posted on May 21, 2018May 31, 2018 by Hal Pomeranz

The XFS file system was originally developed by Silicon Graphics for their IRIX operating system. The Linux version is increasingly popular– Red Hat has adopted XFS as their default file system as of Red Hat Enterprise Linux v7. Unfortunately, while XFS is becoming more common on Linux systems, we are lacking forensic tools for decoding this file system. This series will provide insights into the XFS file system structures for forensics professionals, and document the current state of the art as far as tools for decoding XFS.

I would like to thank the XFS development community for their work on the file system and their help in preparing these articles. Links to the documentation, source code, and the mailing list are available from XFS.org (http://xfs.org/). I wouldn't have been able to do any of this work without these resources.

A Quick Overview of XFS

XFS is a modern journaled file system which uses extent-based file allocation and B+Tree style directories. XFS supports arbitrary extended file attributes. Inodes are dynamically allocated. The block size is 4K by default, but can be set to other values at file system creation time. All file system metadata is stored in "big endian" format, regardless of processor architecture.

Some of the structures in XFS are recognizable from older Unix file systems. XFS still uses 32-bit signed Unix epoch style timestamps, and has the "Year 2038" rollover problem as a result. XFS v5– the version currently used in Linux– does have a creation date (btime) field in addition to the normal last modified (mtime), access time (atime), and metadata change time (ctime) timestamps. XFS timestamps also have an additional 32-bit nanosecond resolution element. File type and permissions are stored in a packed 16-bit value, just like in older Unix file systems.

Very little data gets overwritten when files are deleted in XFS. Directory entries are simply marked as unused, and the extent data in the inode is still visible after deletion. File recovery should be straightforward.

In addition, standard metadata structures in XFS v5 contain a consistent unique file system UUID value, along with information like the inode value associated with the data structure. Metadata structures also have unique "magic number" values. These features facilitate file system and data recovery, and are very useful when carving or viewing raw file system data. Metadata structures include a CRC32 checksum to help detect corruption.

One interesting feature of XFS is that a single file system is subdivided into multiple *Allocation Groups*— four by default on RHEL systems. Each allocation group (AG) can be treated as a separate file system with its own inode and block lists. The intention was to allow multiple threads to write in parallel to the same file system with minimal interaction. This makes XFS a quite high performing file system on multi-core systems.

It also leads to a unique addressing scheme for blocks and inodes that uses a combination of the AG number and a relative block or inode offset within that AG. These values are packed together into a single address, normally stored as a 64-bit value. However the actual length of the relative portion of the address and the AG value can vary from file system to file system, as we will discuss below. In other words, it's complicated.

The Superblock

As with other Unix file systems, XFS starts with a superblock which helps decode the file system. The superblock occupies the first 512 bytes of each XFS AG. The primary superblock is the one in AG 0 at the front of the file system, with the superblocks in the other AGs used for redundancy.

Only the first 272 bytes of the superblock are currently used. Here is a breakdown of the information from the superblock:

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
0x000	58	46	53	42	00	00	10	00	00	00	00	00	00	94	24	00	XFSB..........$.
0x010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0x020	E5	6C	3B	41	CA	03	4B	41	B1	5C	DD	60	9C	B7	DA	71	ål;AÊ.KA±\Ý`.·Úq
0x030	00	00	00	00	00	80	00	04	00	00	00	00	00	00	00	40	...............@
0x040	00	00	00	00	00	00	00	41	00	00	00	00	00	00	00	42	.......A.......B
0x050	00	00	00	01	00	25	09	00	00	00	00	04	00	00	00	00	.....% ....
0x060	00	00	12	84	B4	B5	02	00	02	00	00	08	00	00	00	00	....´µ..........
0x070	00	00	00	00	00	00	00	00	0C	09	09	03	16	00	00	19	....
0x080	00	00	00	00	00	02	C5	00	00	00	00	00	00	00	03	85	......Å.........
0x090	00	00	00	00	00	84	50	DC	00	00	00	00	00	00	00	00	......PÜ........
0x0A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0x0B0	00	00	00	00	00	00	00	04	00	00	00	00	00	00	00	00	....
0x0C0	00	00	00	00	00	00	00	01	00	00	01	8A	00	00	01	8A	....
0x0D0	00	00	00	00	00	00	00	00	00	00	00	01	00	00	00	00	....
0x0E0	0A	58	32	D0	00	00	00	00	FF	FF	FF	FF	FF	FF	FF	FF	X2Ð....ÿÿÿÿÿÿÿÿ
0x0F0	00	00	00	19	00	00	36	EA	00	00	00	00	00	00	00	00	......6ê........
0x100	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0x110	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0x120	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0x130	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0x140	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

```
0-3      Magic Number                       "XFSB"
4-7      Block Size (in bytes)              0x1000 = 4096
8-15     Total blocks in file system        0x942400 = 9,708,544

16-23    Num blocks in real-time device     zeroed
24-31    Num extents in real-time device    zeroed

32-47    UUID                               e56c3b41-...-dd609cb7da71

48-55    First block of journal             0x800004 = 8388612
56-63    Root directory's inode             0x40 = 64

64-71    Real-time extents bitmap inode     0x41 = 65
72-79    Real-time bitmap summary inode     0x42 = 66

80-83    Real-time extent size (in blocks)  0x01
84-87    AG size (in blocks)                0x250900 = 2,427,136 (c.f. 8-15
88-91    Number of AGs                      0x04
92-95    Num of real-time bitmap blocks     zeroed

96-99    Num of journal blocks              0x1284 = 4740
100-101  File system version and flags      0xB4B5 (low nibble is version)
102-103  Sector size                        0x200 = 512
104-105  Inode size                         0x200 = 512
106-107  Inodes/block                       0x08
108-119  File system name                   not set-- zeroed
120      log2(block size)                   0x0C (2^^12 = 4096)
121      log2(sector size)                  0x09 (2^^9 = 512)
122      log2(inode size)                   0x09
123      log2(inode/block)                  0x03 (2^^3 = 8 inode/block)
124      log2(AG size) rounded up           0x16 (2^^22 = 4M > 2,437,136)
125      log2(real-time extents)            zeroed
126      File system being created flag     zeroed
127      Max inode percentage               0x19 = 25%

128-135  Number of allocated inodes         0x2C500 = 181504
136-143  Number of free inodes              0x385 = 901

144-151  Number of free blocks              0x8450dc = 8,671,452
152-159  Number of free real-time extents   zeroed

160-167  User quota inode                   -1 (NULL in XFS)
168-175  Group quota inode                  -1 (NULL in XFS)

176-177  Quota flags                        zero
178      Misc flags                         zero
179      Reserved                           Must be zero
180-183  Inode alignment (in blocks)        0x04
184-187  RAID unit (in blocks)              zeroed
188-191  RAID stripe (in blocks)            zeroed

192      log2(dir blk allocation granularity)          zero
```

```
193      log2(sector size of externl journal device)  zero
194-195  Sector size of external journal device       zero
196-199  Stripe/unit size of external journal device  0x01
200-203  Additional flags                             0x018A
204-207  Repeat additional flags (for alignment)      0x018A

/* Version 5 only */
208-211  Read-write feature flags (not used)          zero
212-215  Read-only feature flags                      zero
216-219  Read-write incompatibility flags             0x01
220-223  Read-write incompat flags for log (unused)   zero

224-227  CRC32 checksum for superblock                0x0A5832D0
228-231  Sparse inode alignment                       zero
232-239  Project quota inode                          -1

240-247  Log seq number of last superblock update     0x19000036EA
248-263  UUID used if INCOMPAT_META_UUID feature      zeroed
264-271  If INCOMPAT_META_RMAPBT, inode of RM btree   zeroed
```

Rather than discussing all of these fields in detail, I am going to focus in on the fields we need to quickly get into the file system.

First we need basic file system structure size information like the block size (bytes 4-7) and inode size (bytes 104-105). XFS v5 defaults to 4K blocks and 512 byte inodes, which is what we see here.

As we'll discuss below, the number of AGs (bytes 88-91) and the size of each AG in blocks (bytes 84-87) are critical for locating data's physical location on the storage device. This file system has 4 AGs which each contain 2,427,136 blocks (roughly 9.6GB per AG or just under 40GB for the file system).

The superblock contains the inode number of the root directory (bytes 56-63)– this value is normally 64. We also find the starting block of the file system journal (bytes 48-55) and the journal length in blocks (bytes 96-99). We'll cover the journal in a later article in this series.

While looking at file system metadata in a hex editor is always fun, XFS does include a program named xfs_db which allows for more convenient decoding of various file system structures. Here's an example of using xfs_db to decode the superblock of our example file system:

```
[root@localhost XFS]# xfs_db -r /dev/mapper/centos-root
xfs_db> sb 0
xfs_db> print
magicnum = 0x58465342
blocksize = 4096
dblocks = 9708544
rblocks = 0
rextents = 0
uuid = e56c3b41-ca03-4b41-b15c-dd609cb7da71
[...]
```

"xfs_db -r" allows read-only access to mounted file systems. The "sb 0" command selects the superblock from AG 0. "print" has a built-in template to automatically parse and display the superblock information.

Inode and Block Addressing

Typically XFS metadata uses “absolute” addresses, which contain both AG information and a relative offset from the start of that AG. This is what we find here in the superblock and in directory files. Sometimes XFS will use “AG relative” addresses that only include the relative offset from the start of the AG.

While XFS typically allocates 64-bits to hold absolute addresses, the actual size of the address fields varies depending on the size of the file system. For block addresses, the number of bits for the “AG relative” portion of the inode is the log2(AG size) value found in superblock byte 124. In the example superblock, this value is 22. So the lower 22 bits of the block address will be the relative block offset. The upper bits will be used to hold the AG number.

The first block of the file system journal is at address 0x800004. Let’s write that out in binary showing the AG and relative block offset portions:

```
     0x800004   =    1000 0000 0000 0000 0000 0100
AG# in upper 2 bits---/\---22 bits of relative block offset
```

So the journal starts at relative block offset 4 from the beginning of AG 2.

But where is that in terms of a physical block offset? The physical block offset can be calculated as follows:

```
(AG number) * (blocks per AG) + (relative block offset)
     2      *    2427136      +         4   =    4854276
```

We could perform this calculation on the Linux command line and use dd to extract the first block of the journal:

```
[root@localhost XFS]# dd if=/dev/mapper/centos-root bs=4096 \
       skip=$((2*2427136 + 4)) count=1 | xxd
0000000: 0000 0021 0000 0000 6901 0000 071a 4dba  ...!....i.....M.
0000010: 0000 0010 6900 0000 4e41 5254 2800 0000  ....i...NART(...
[...]
```

Inode addressing is similar. However, because we can have multiple inodes per block, the relative portion of the inode address has to be longer. The length of relative inode addresses is the sum of superblock bytes 123 and 124– the log2 value of inodes per block plus the log2 value of blocks per AG. In our example this is 3+22=25.

The inode address of the root directory isn’t a very interesting example– it’s just inode offset 64 from AG 0. For a more interesting example, I’ll use my /etc/passwd file at inode 67761631 (0x409f5df). Let’s take a look at the bits:

```
     0x409f5df   =    0100 0000 1001 1111 0101 1101 1111
  AG# in upper 3 bits---/\---25 bits of relative inode
```

So the /etc/passwd file uses inode 0x9f5df (652767) in AG 2.

Where does this inode physically reside on the storage device? The relative block location of an inode in XFS is simply the integer portion of the inode number divided by the number of inodes per block. In our case this is 652767 div 8 or block 81595. The inode offset in this block is 672767 mod 8, which equals 7.

Now that we know the AG and relative block number for this inode, we can extract it as we did the first block of the journal. We can even use a second dd command to extract the correct inode offset from the block:

```
[root@localhost XFS]# dd if=/dev/mapper/centos-root bs=4096 \
                              skip=$((2*2427136 + 81595)) count=1 |
                      dd bs=512 skip=7 count=1 | xxd
0000000: 494e 81a4 0302 0000 0000 0000 0000 0000  IN..............
0000010: 0000 0001 0000 0000 0000 0000 0000 0000  ................
[...]
```

Note that the xfs_db program can perform address conversions for us. However, in order to use xfs_db it must be able to attach to the file system in order to have the correct length for the AG relative portion of the address. Since this may no always be possible, knowing how to manually convert absolute addresses is definitely a useful skill.

Here is how to get xfs_db to convert the block and inode addresses we used in the examples above:

```
[root@localhost XFS]# xfs_db -r /dev/mapper/centos-root
xfs_db> convert fsblock 0x800004 agno
0x2 (2)
xfs_db> convert fsblock 0x800004 agblock
0x4 (4)
xfs_db> convert inode 67761631 agno
0x2 (2)
xfs_db> convert inode 67761631 agino
0x9f5df (652767)
xfs_db> convert inode 67761631 agblock
0x13ebb (81595)
xfs_db> convert inode 67761631 offset
0x7 (7)
```

The first two commands convert the starting block of the journal (xfs_db refers to absolute block addresses as "fsblock" values) to the AG number (agno) and AG relative block offset (agblock). We can also use the convert command to translate inode addresses. Here we calculate the AG number, AG relative inode (agino), the AG relative block for the inode, and even the offset in that block where the inode resides (offset). The values from xfs_db match the values we calculated manually above. You will note that we can use either hex or decimal numbers as input.

Now that we can locate file system structures on disk, Part 2 (https://righteousit.wordpress.com/2018/05/23/xfs-part-2-inodes/) of this series will focus on the XFS inode format. I hope you will return for the next installment.

Posted in Forensics Tagged File Systems, Linux, XFS

Published by Hal Pomeranz

Independent Computer Forensics and Information Security consultant. Expert Witness. Trainer. *View all posts by Hal Pomeranz*

2 thoughts on “XFS (Part 1) – The Superblock”

1. **XFS (Part 1) – The Superblock – Cyber Forensicator** *May 22, 2018*
   […] is becoming more common on Linux systems, but we are lacking forensic tools for decoding it. The first post will provide you with a quick overview of this file system, and focus on the […]

2. **Recent technical articles. - CertDepot** *May 23, 2018*
   […] XFS (Part 1) – The Superblock, […]

Comments are closed.