EXT3 File Recovery via Indirect Blocks « Threat Research Blog

by Hal Pomeranz

Recovering complete file images from unallocated space on Linux systems can be a tricky problem. The EXT3 metadata structures-*index nodes* or *inodes* for short-are mostly zeroed out when they are deallocated. During this process, all of the inode's block pointers (that would normally be used to access the file data when the file was allocated) are lost. The original file contents will still exist in unallocated data blocks in the file system-at least until those blocks are reused-but there's no "map" to reconstruct those data blocks into the original file.

Historically, file recovery in this scenario has relied on "file carving" tools such as Foremost. These tools leverage the fact that the data blocks that make up a file tend to be allocated consecutively. If you know a byte "signature" that marks the beginning of a particular type of file, you can often recover most or all of the file by collecting subsequent blocks. If the file format also has a "signature" that marks the end of the file, that can be used as a marker for when to stop collecting blocks.

However, there are some known issues with this technique when recovering data from Linux systems:

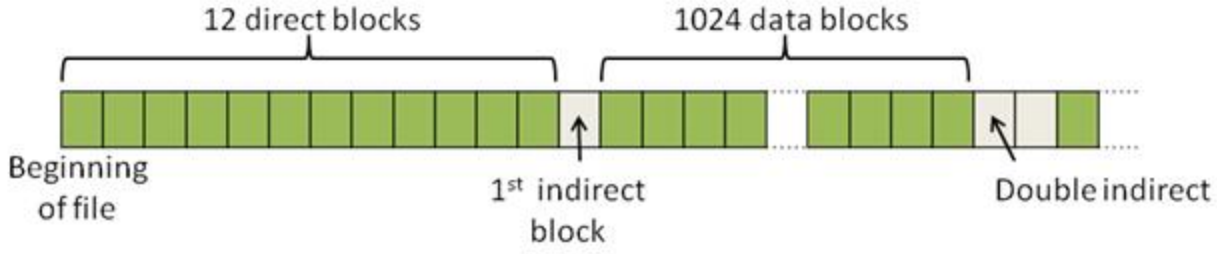
- Being a largely text-oriented operating system, many common Linux file artifacts lack good file "signatures". And even for common binary file types such as GZIP data where a file signature can be developed, lack of a consistent "end of file" marker makes accurate recovery of these files a challenge.
- EXT3 uses data blocks in the middle of a file's data run to store metadata. These so-called *indirect blocks* are used to store pointers to data blocks once a file grows too large to be represented by the relatively small number of block pointers in the inode. These indirect blocks will cause file corruption if they are extracted along with the regular data blocks that contain file content.
- Attempting to recover a file by gathering consecutive blocks breaks down when the file becomes fragmented across multiple areas of the disk.

The problem of indirect blocks in the middle of the file content is addressed by tools like Foremost by simply skipping over the indirect block and ignoring its contents. Actually, Foremost will skip the first indirect block that normally occurs in the 13^{th} data block in the run but fails to remove later indirect blocks (the double and treble indirect block chains) from the recovered image, again leading to file corruption on recovered files larger than 4MB or so.

Simply skipping over or attempting to edit out the indirect block data from the recovered file is probably the wrong thing to do in any event. After all, the block pointer metadata in the indirect blocks provide a map to the location of large chunks of file content from the original file. I have developed a couple of simple command-line tools to find and use the indirect block data to more accurately recover files from unallocated space.

Finding and Using Indirect Blocks

The figure below shows the typical block layout for a file in a Linux file system. From the start of the file, there are 12 data blocks which are normally indexed by the 12 *direct block pointers* in the inode-assuming 4K blocks, this addresses the first 48K of the file. Once the file grows beyond 48K in size, the operating system uses the next block as an indirect block for storing block pointers and then resumes using the subsequent blocks to store data. A single 4K indirect block can store 1024 4-byte block pointers, which handles file sizes of up to 4MB (plus the 48K indexed by the direct block pointers). When the file grows beyond 4MB+48K in size, double and even treble indirection can be used-indirect blocks that address other indirect blocks and so on.



One key fact here is that the indirect blocks are at predictable locations relative to the beginning of the file and to each other:

- Unless the file is heavily fragmented, the first indirect block should be 13 blocks from the beginning of the file. If you have a reliable file signature for the type of file you're searching for, locating the first indirect block should be straightforward.
- Subsequent indirect blocks should occur in the block immediately following the last data block referenced by any given indirect block. Because EXT3 null fills slack space, you will know that you've reached the end of the chain of indirect blocks when you encounter and indirect block with null block pointers.

Another important fact is that the first block pointer in an indirect block will typically reference the very next block in the file system. In other words, if you have an indirect block at block number 1000, then the first data block that it points to will be block 1001. This gives us a signature for the indirect blocks themselves: look for blocks whose first four bytes decode to the address of the next block. While this property doesn't necessarily have to hold true-and there is also the possibility of false-positives-real-world testing has shown that this signature is very, very good at correctly identifying indirect blocks.

Combining all of these ideas has led to the development of two tools. The first is a tool called frib (File Recovery via Indirect Blocks) which when given the block number of the beginning of a file or of an indirect block will attempt to recover the complete file contents from that point. The simplest case is to use frib in conjunction with another tool like sigfind from the Sleuthkit which will let you locate "beginning of file" signatures:

```
# sigfind -b 4096 1F8B0800 ext3-example.img
Block size: 4096 Offset: 0 Signature: 1F8B0800
Block: 251904 (-)
Block: 252096 (+192)
Block: 252293 (+197)
Block: 252599 (+306)
...
# frib ext3-example.img 252599 >recovered.gz
# tar ztf recovered.gz
...
perl-5.10.1/patchlevel.h
perl-5.10.1/Configure
```

Here we're feeding sigfind a common signature for GZIP-ed files, and it locates a number of blocks that start with this signature. Choosing one of the block numbers returned by sigfind, we invoke frib which emits the recovered file data to the standard output (which we redirect into a file). Because this is a test example created for validating the tool, we know that the recovered file is a GZIP-ed tar archive, and we can use the tar command to verify that we've recovered the entire archive without corruption.

But what about cases where you're attempting to recover data formats that have no strong beginning of file signature (e.g. Linux log files)? I've developed a second tool, fib (Find Indirect Blocks), that uses the "block N whose first four bytes decode to N+1" signature discussed above to locate potential indirect blocks. If we run fib against our test image, we get output like this:

```
# fib ext3-example.img
585
611
34828
53
37288
193
37290
10
38924
432
41609
35
43020
2066
61452
43
131598
8181
139799
98
229956
75
251916
179
252108
184
252305
293
252611
3436
256063
485
```

The first column of output is the block numbers of the potential indirect blocks identified by our signature. The second column is the total number of data blocks in the fully unpacked indirect block chain starting at the given block. As you can see, fib was able to detect the double indirect block chains following blocks 43020, 131598, and 252611 and reassemble lists of blocks beyond the 1024 blocks referenced in the initial indirect block (fib will also properly recognize treble indirect block chains and reassemble those as well, though there are none in this example image).

Block 252611 should be the first indirect block of the .tar.gz file starting at block 252599 that we recovered in the previous example. Let's use frib again to recover the same file starting with the indirect block address this time:

```
# frib -I dblks ext3-example.img 252611>indblks
# ls -lh *blks
```

```
-rw-r--r-- 1 hal hal 48K 2011-01-16 08:09 dblks
-rw-r--r-- 1 hal hal 14M 2011-01-16 08:09 indblks
# cat dblks indblks >recovered2.gz
# ls -l recovered*.gz
-rw-r--r-- 1 hal hal 14118912 2011-01... recovered2.gz
-rw-r--r-- 1 hal hal 14118912 2011-01... recovered.gz
# diff recovered*.gz
```

When invoked with the "-I" option, the block address passed in to frib is assumed to be the address of the first indirect block in the file. This means that normally the preceding 12 blocks should be the first 48K of the file. The argument to the "-I" option is the name of a file where the contents of those 12 blocks should be placed. The remainder of the file referenced by the indirect block chain is written to the standard output, which again we're redirecting into a file in order to capture the data. If you believe that the first 12 blocks captured by frib genuinely are the first 48K of the file, then you can concatenate the files we captured in order to recreate the original file image. As you can see from the ls and diff output above, the resulting file is identical to the one we recovered in the first example where we used sigfind to find the beginning of file marker.

In fact, the assumption that the 12 blocks before the indirect block are the start of the file is true so frequently that frib allows you to specify "-I -" in order to emit those 12 blocks on the standard output along with the rest of the data recovered by frib:

```
# frib -I - ext3-example.img 252611 >recovered3.gz
# diff recovered.gz recovered3.gz
```

Both frib and fib are implemented as Perl scripts that use the blkcat utility from the Sleuthkit to actually dump the recovered data blocks (fib also requires TSK's fsstat utility). So you must have the Sleuthkit installed to make use of these tools. However, the advantage here is that frib and fib can work against both live systems and any sort of image file format supported by the Sleuthkit. You may use the standard Sleuthkit "-o" option for specifying a sector offset of the beginning of a file system in a disk image with both frib and fib (indeed this option is simply passed through to the underlying Sleuthkit tools).

NOTE: Both frib and fib assume little-endian byte ordering. If you're using these tools on a file system image from a big-endian machine, specify "-B" to use the correct byte ordering. Both frib and fib support the "-d" flag for dumping debugging output.

Because EXT3 null fills slack space at the end of the trailing block in a file, files recovered by frib-which works on a block-by-block basis-will typically have extra trailing nulls at the end. Usually this is not much of an issue for most Linux file formats. However, frib does support the "-t" flag to "trim" trailing nulls from the end of the file. This is particularly useful when using frib to recover text files such as Linux log files. *NOTE: For various binary formats, blindly stripping off nulls at the end of the file may produce a corrupted file image.*

Some Additional fib Features

String searching is a common forensic technique for zeroing in on potentially interesting data. It is relatively straightforward to compute the address of a block that contains a string of interest, but recovering the entire file from that point is more difficult.

fib supports "-a" and "-A" options for locating the address of a given block inside other potential indirect blocks in the file system. For example, if you had a string of interest in block 123456 of an image file called "myfile.img", then "fib -a 123456 myfile.img" would return the address of the first block in the image that contains the 123456 encoded as a 4-byte hex value. Because there is the

potential for false-positives, the "-A" option will return *all* blocks in the file system that contain the specified address.

fib itself is not terribly fast, so you can specify a range of blocks to search:

```
fib ext3-example.img 196608 229375 # scans only blocks
# 196608-229375 (inclusive)
fib ext3-example.img 229376
# scans from 229376 to
# end of image
```

You may use block ranges either in fib's normal mode or with the -a/-A options.

Searching block ranges can be very useful in EXT3 file systems because data blocks are allocated in "block groups" and EXT3 tries very hard to allocate all files in a given directory to the same block group. So if you locate a string of interest or other file signature in a given block, chances are the rest of the file is contained in blocks in the same block group. You can constrain your fib searches to the much smaller collection of blocks in the local block group-typically only 32K blocks. Block group ranges can be viewed in the output of TSK's fsstat tool.

Conclusion

Mandiant has used these tools with good success on actual customer engagements. We hope you will find them a useful addition to your arsenal. Questions about the tools, bug reports, and suggestions should be directed to Hal.Pomeranz@mandiant.com. Download the tools.