Understanding Indirect Blocks in Unix File Systems

When I'm covering Linux Digital Forensics on the last day of Sec506 (that's my SANS Linux/Unix Security track for those sluggards out there that haven't memorized the SANS course numbering scheme), questions always come up about the function of indirect blocks in standard Unix file systems (meaning ext2/ext3 on Linux, UFS on Solaris, FFS on BSD, and others derived, directly or indirectly, from Kirk McKusick's original Fast File System from 4.2BSD). Generally these questions always arise in one of two contexts:

- What's that extra information at the end of my istat output?
- Why do I always need the -d option when using foremost on Unix file system images?

It turns out that even people who've been using Unix for a long time are a little fuzzy on the subject of indirect blocks, what they are, how they work, and so on. So let's talk about indirect blocks, and why you care about them from the perspective of file system forensics.

A Quick Review of Unix File Systems

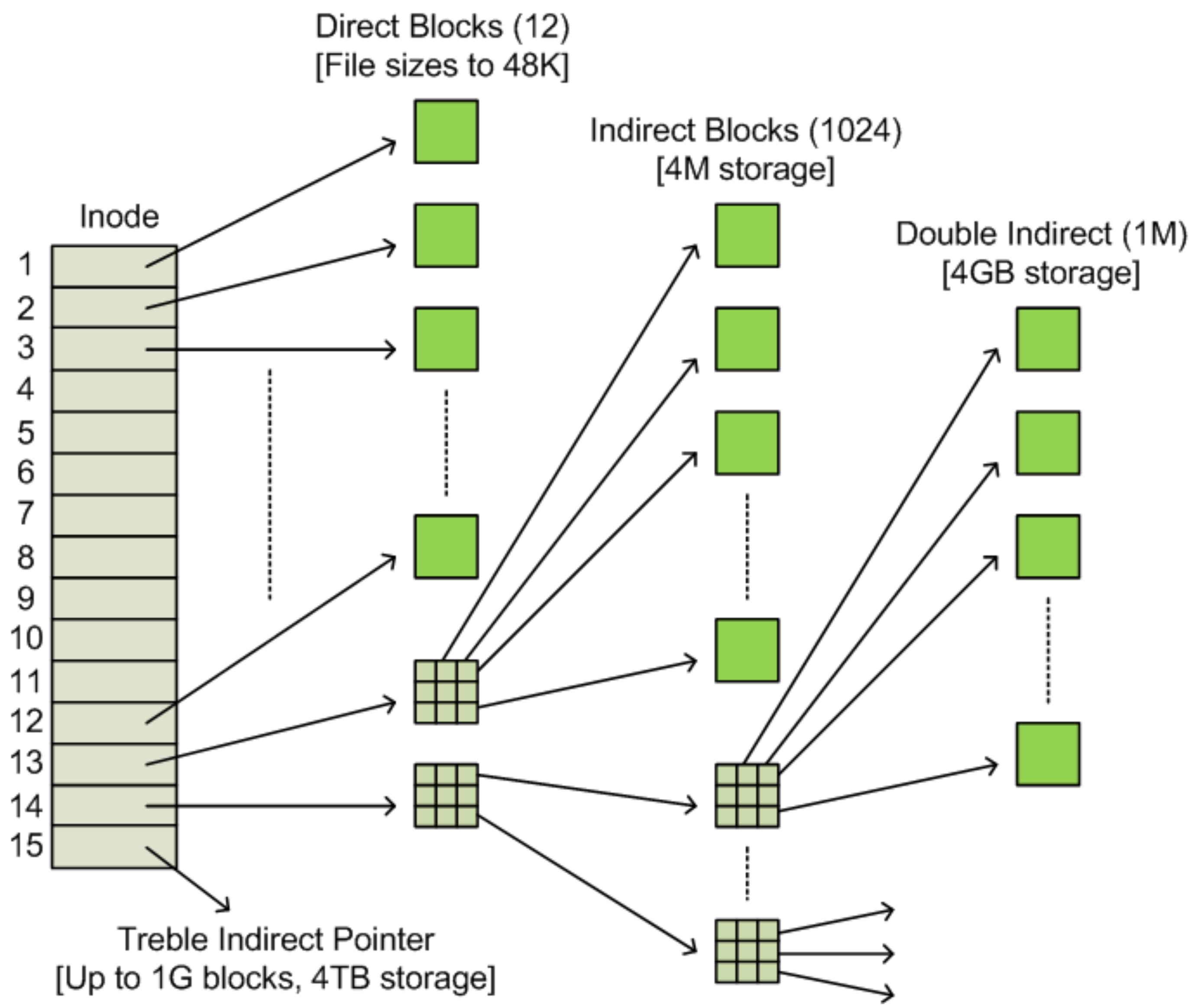
In a standard Unix file system, files are made up of two different types of objects. Every file has an *index node* (*inode* for short) associated with it that contains the metadata about that file: permissions, ownerships, timestamps, etc. The contents of the file are stored in a collection of *data blocks*. At this point in the discussion, a lot of people just wave their hands and say something like, "And there are pointers in the inode that link to the data blocks."

As it turns out, there are only fifteen block pointers in the inode. Assuming standard 4K data blocks, that means that the largest possible file that could be addressed directly would be 60K-- obviously not nearly large enough. In fact, only the first 12 block pointers in the inode are reserved for *direct block pointers*. This means you can address files of up to 48K just using the direct pointers in the inode.

Beyond that, you start getting into indirect blocks:

- The thirteenth pointer is the *indirect block pointer*. Once the file grows beyond 48K, the file system grabs a data block and starts using it to store additional block pointers, setting the thirteenth block pointer in the inode to the address of this block. Block pointers are 4-byte quantities, so the indirect block can store 1024 of them. That means that the total file size that can be addressed via the indirect block is 4MB (plus the 48K of storage addressed by the direct blocks in the inode).
- Once the file size grows beyond 4MB + 48KB, the file system starts using *doubly indirect blocks*. The fourteenth block pointer points to a data block that contains the addresses of other indirect blocks, which in turn contain the addresses of the actual data blocks that make up the file's contents. That means we have up to 1024 indirect blocks that in turn point to up to 1024 data blocks-- in other words up to 1M total 4K blocks, or up to 4GB of storage.
- At this point, you've probably figured out that the fifteenth inode pointer is the *trebly indirect block pointer*. With three levels of indirect blocks, you can address up to 4TB (+4GB from the doubly indirect pointer, +4M from the indirect block pointer, +48K from the direct block pointers) for a single file.

Here's a picture to help you visualize what I'm talking about here:



By the way, while 4TB+ seemed like an impossibly large file back in the 1970s, these days people actually want to create files that are significantly larger than 4TB. Modern file systems deal with this by recognizing that addressing every single block in the file is incredibly wasteful. Instead the file system allocates collections of consecutive blocks, usually referred to as *extents*, and then stores pointers to the first block in the extents. Even with a relatively small extent size, you can index huge files-- 64K extents mean you can create a 64TB file!

Why All This Matters for Forensics

While all of this is fascinating in a deeply nerdy sort of way, why am I discussing this on a digital forensics blog? First, this answers the question of why you always must use the -d option with foremost-- without -d foremost won't follow the indirect block pointers and you'll never recover more than the first 48K of the files that foremost finds. I've often wondered why the -d behavior isn't simply the default when foremost detects a Unix file system, and I speculate it's because the foremost developers are primarily Windows-oriented and are interested in analyzing Windows file systems, which are obviously far more numerous in the general population.

You can also observe indirect blocks using tools from the Sleuthkit (TSK). Take a look at this snippet of istat output:

```
# istat /dev/mapper/elk-home 802839
inode: 802839
Allocated
Group: 98
Generation Id: 2610194750
uid / gid: 0 / 0
mode: -rwxrwx---
size: 2145910784
num of links: 1

[...]



Direct Blocks:
8552962 8552963 8552964 8552965 8552966 8552967 8583175 16566928
16566929 16566930 16566931 16566932 16566934 16566935 16566936 16566937
[...]



Indirect Blocks:
16566933 16578347 16578348 16579373 16580398 16581943 16582968 16583993
[...]
```

As you can see, this is a huge file-- 2GB in size, which means we should be using doubly indirect pointers to index all of the data blocks. Indeed, at the end of the istat output you can see the list of indirect blocks.

What's perhaps more interesting is that if you look at the list of direct blocks, you'll notice that there's a gap in the numbering between the 12th (block number 16566932) and 13th blocks (16566934). You'll find block number 16566933 in the list of indirect blocks. So the file system assigned the first 12 direct blocks to the direct block pointers in the inode, then realized it was going to need an indirect block pointer and just grabbed the next contiguous block to use as the indirect block and then started using the subsequent blocks as data blocks (whose addresses will appear in the indirect block). This illustrates a couple of important aspects of the Unix file system:

1. Consecutive data blocks are allocated whenever possible
2. The file system will just greedily grab the next available block when it needs an indirect block pointer

I'm going to be exploiting these tendencies in just a moment to get some interesting information out of the file system.

But first let's actually use dcat to dump the contents of one of the indirect blocks:

```
# dcat -h /dev/mapper/elk-home 16566933
0       96cafc00 97cafc00 98cafc00 99cafc00     .... .... .... ....
16      9acafc00 9bcafc00 9ccafc00 9dcafc00     .... .... .... ....
[...]
4064    23f7fc00 24f7fc00 25f7fc00 26f7fc00     #... $... %... &...
4080    27f7fc00 28f7fc00 29f7fc00 2af7fc00     '... (... )... *...
```

This is actual output from my Linux AMD_64 laptop, and consequently the block pointers in the indirect block are in little-endian byte order. That means that the actual address of the first block is 0x00fcca96, or 16566934 in decimal (i.e. the thirteenth block listed in istat's summary of direct blocks in the file). You can see the block addresses increasing monatomically from that point on. Again there's a sort of geeky coolness factor here, but is there a practical application for this information? Well, there's no question in my mind about that.

Consider that when a file is deleted on an ext3 file system, the contents of the inode associated with that file are zeroed before the inode is returned to the free list. That means you lose all of the block pointers in the inode, and reconstructing the file is extremely difficult in cases where tools like foremost don't recognize the type of file you're searching for. You can use string searching to find particular strings of interest in the middle of a file, and because the file system tends to allocate blocks contiguously you can often find larger chunks of the file by dumping the blocks before and after the block containing your string of interest. But this process is time consuming and error-prone.

Leveraging Indirect Blocks

However, suppose you could locate one of the indirect blocks for the file-- suddenly you have the exact addresses of up to 4MB of data from the file! But how do we go about tracking down an indirect block and recognizing it once we find it? Consider that whatever string of interest we locate by searching our file system image is probably in the middle of a collection of at most 1024 blocks addressed by an indirect block pointer. If the file was created in a single write operation (the most common situation for pretty much every file in the file system, except possibly log files and other incrementally growing files), the indirect block pointer is sitting right at the beginning of that collection of 1024 blocks. That means you should run into the indirect block pointer before too long if you start searching backwards through the file system from your string of interest. How will you know when you get there? Just look for a block where the first four bytes of the block translate to the address of the next block-- and where the rest of the block is probably full of monotomically increasing 4-byte hex values, which are the addresses of the next 1023 blocks. Can't miss it.

But it gets better:

- Chances are that the block after the last data block address in your indirect block pointer is yet another indirect block pointing at another 4MB of the deleted file's original contents (and very often that block is exactly 1025 blocks away from the indirect block you've found).
- You may be able to find what appears to be two indirect blocks back-to-back. This means you've found the beginning of the double indirect block collection and can recover up to 4GB of data from the file just by following the block pointers appropriately. Extra bonus points for finding the three blocks that start the trebly indirect block collection.

I was interested in playing around with these notions, so I threw together a quick little Perl script called idcat for dumping the contents of an indirect block. The syntax is similar to TSK's dcat utility, but it will dump the contents of all of the data blocks stored in the indirect block you specify on the command line (because I was lazy, the program uses dcat to dump the block contents, so you'll need to have TSK installed). If you use the -a option, idcat will dump the block addresses, rather than dumping the block contents. This is just a proof-of-concept-- the interested reader is invited to create the a more useful tool that actually heuristically analyzes file systems looking for collections of indirect blocks necessary to reconstruct the entire file.

Hal Pomeranz is an independent IT/Security consultant and SANS Institute Faculty Fellow. While not a professional forensic analyst, he has been known to grovel through Unix file system internals for fun. They don't let Hal out very much.--