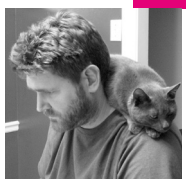
DAVID JOSEPHSEN

iVoyeur: Admin, root thyself.



David Josephsen is the author of *Building a Monitoring Infrastructure with Nagios* (Prentice Hall PTR, 2007) and Senior Systems Engineer at DBG, Inc., where he maintains a gaggle of geographically dispersed server farms. He won LISA '04's Best Paper award for his co-authored work on spam mitigation, and he donates his spare time to the SourceMage GNU Linux Project.

dave-usenix@skeptech.org

DID YOU HAPPEN TO SEE THE LATEST Die Hard movie? The one where the interwebs are broken? Well if you did, it probably annoyed you quite a bit. It's a pretty typical Hollywood blockbuster take on computers and the nerds who love them, and they pretty typically get it all horribly wrong. I'm kind of atypical when it comes to these sorts of films: I actually rather like them. I think I find something endearing in Hollywood's belief in magic [1]. But when I see them, I try to do so alone or in the company of nerds; otherwise someone I'm with will invariably ask, à la Homer Simpson [2], "Computers can do that?!" (or some variation thereof).

I imagine it's the same pain periodically felt by paleontologists who've been dragged to *Jurassic Park*, or um . . . pagans at Harry Potter? Anyway, I don't like to disappoint folks, and since the answer to the Homer question is almost always, "Well, not really," it's better to just avoid the situation when I can. But much as I hate to harsh on their newfound interest in computer security, I can't help but chuckle to myself at how disappointed they'd be if they knew the truth about the security capabilities of today's computers.

I'm not talking about Windows being vulnerable to the sploit of the week, or even theoretical design issues such as mandatory access control. I'm talking about simple functionality that everyone outside of our community probably assumes is there and would be surprised to find out is not. For example, if you asked a random movie producer whether he or she thought a computer kept a record of all the changes made to any given file on the file system for the past week, I think you'd find that most of them would give an emphatic yes.

How many times was /etc/foobar changed, by whom, and when? This is a problem I think most people would assume has been solved by now. But in reality, this type of auditing information is surprisingly difficult to come by. Indeed, very good books [3] have been written on the subject of teasing this type of stuff from a file system offline and after an attack. To pull it off in real time you need to audit changes to every file in the file system. The audit records need to include who, what, and when, and they need to be captured and written in a way that is difficult to bypass or modify after the

fact. Add to that UNIX's rather murky definition of a "file," and this isn't a solved problem. There are several solutions, and they're all far from perfect.

So since that example ties in so well with the filesystem theme of this issue, I'd like to take a look at some ways to monitor changes to the file system, including a method you may not have considered, namely using kernel instrumentation such as DTrace or SystemTap to audit kernel vfs read/write calls. I've become rather fond of the method lately for several reasons, and I hope it will prove useful to you.

By far the most popular way to do this sort of thing normally is with a filesystem integrity checker such as Tripwire [4] or Samhain [5]. These programs are polling engines; they usually run as a daemon and periodically wake up to recursively check the file system against a database of hashes. In practice this works fairly well. They have a reasonable overhead once the hash database is created, they capture changes to file metadata such as permissions, ownership, and modification dates, and they are pretty good at staying out of the way.

I've used Samhain in my production environments for a few years now, and I don't hate it. It has some rudimentary rootkit detection capabilities on Linux and Open/Free BSD via the /dev/kmem file, can hide itself from script kiddies, and does a good job of finding and notifying you of changes to the file system. Although it wants badly for you to use its client/server model, it will play nicely with your existing tools such as syslog, Splunk, databases, and SEC/Logsurfer, if you ask it politely.

The biggest thing I don't like about the filesystem integrity checkers has got to be that they can't tell you who changed the file. Ideally I'd like to know the pid and uid of the thingy that changed a given file. Since integrity checkers simply wake up once every so often and compare files against MD5 hashes in a database, they only know that a file has changed and not who changed it. The question of "who" is what you might call a fundamental piece of information.

A less important nit is that the change notifications are delayed by the length of the polling interval. Depending on your situation, it could take some time before you know what files have changed, which can be frustrating when you're dealing with an intrusion in real time. The integrity checkers can obviously only notify you of changes to "normal" files; changes to special files such as sockets and block devices cannot be detected this way. Finally, the integrity checkers are somewhat high in the stack, so it's possible that they could be bypassed for certain types of events. For example, they won't be able to notify you of read events if you have "noatime" set in fstab (because they won't be able to see a difference in access times in the file metadata).

One way to solve some of these problems, including the polling interval delay, is to use the kernel's inotify subsystem. The inotify subsystem provides user-space programs with notifications of file change events. It's used primarily by content-indexing tools such as Beagle [6], but there's no reason it couldn't be used to log changes to files systemwide. There are several user-space implementations, including some shell tools called "inotifytools" [7]. These include a program called "inotifywait" that basically blocks on inotify events for a given directory or set of directories and provides event details to STDOUT. I haven't used inotifytools to recursively monitor /, so I don't know how much overhead it might incur, but from my limited experience it seems pretty scalable. It's also a bit closer to the kernel, so it's more difficult to fool. Unfortunately inotify doesn't solve the "who" problem. The pid/uid of the

changing process is not one of the pieces of information passed by the kernel to user space. Bummer.

A somewhat more indirect approach might be to use tty snooping. Solutions of this type simply listen in on input from the ttys of the machine, thereby logging the actions of users. There are all sorts of implementations here; most of them are shell replacements or patches to existing shells such as bofh-bash and ttysnoop [8], but some are more elegant, kernel-space tools such as Sebek [9]. These tty sniffers work very nicely when a user can't simply launch another shell to bypass them. They solve the "who" problem, giving you granular detail of what changed and sometimes even the content of the change, depending on how the file was edited. These can induce some overhead, however, and, since they tend to be user-centric, they might be bypassed by non-interactive programs or system processes.

Finally, just about every system has a kernel-space auditing subsystem: SELinux and the kernel audit subsystem for Linux, BSM auditing for Solaris et al. These are used to great effect by folks who know them well, and they are probably the closest thing to the "right" answer, but they aren't necessarily focused on file accesses and can generate metric tons of auditing information. They can also be difficult to use and maintain and rarely play nicely with centralized tools such as OSSIM or Syslog.

So let's take a look at the kernel probes approach I've been playing with lately. I should disclaim that the DTrace folks have explicitly warned against the use of DTrace for security auditing [10] because DTrace might drop events if the system becomes overloaded. This is pretty much a deal breaker for DTrace in this context at the moment, but I have a feeling DTrace will eventually be a useful solution here. So for now I'll focus on the SystemTap script in Figure 1.

If you aren't familiar with SystemTap [11], it is comparable to DTrace but only runs on Linux. There are already healthy religions built up around both tools, and I'll probably get flamed for that last sentence, so I'll leave it at that and let you work out the differences for yourself. SystemTap scripts are written in an awk-like language, parsed into C by an interpreter, compiled into a kernel module, and finally loaded into a running kernel. Once loaded, the module can trace system calls and broker information between kernel and user space. It's a fascinating and useful tool, but it requires some understanding of the kernel internals, or at least a good handle on C and a willingness to dig around at the kernel headers to use.

SystemTap requires that CONFIG_DEBUG_INFO, CONFIG_KPROBES, and optionally CONFIG_RELAY and CONFIG_DEBUG_FS be enabled in the kernel. It also assumes some Red Hat–style symlinks to the running kernel source, so check the README if you're installing it on a box that's not Red Hat. One of the more interesting features of SystemTap is the ability to inject blocks of C directly into the system tap script. In the script in Figure 1, I have a function that is written in raw C, but it is called from within the SystemTap scripting language.

The purpose of the script is to probe kernel vfs_read and vfs_write calls and return information about them to STDOUT. This approach has several advantages. First, it takes advantage of the fact that everything in UNIX is a file, so reads and writes to sockets, fifos, block files, etc., will all be captured. Second, since we are writing the instrumentation, we can ask for whatever pieces of information we want, including the pid/uid of the entity making the file access. Next, the probe is fairly limited in scope, so we get what we want and nothing we don't, and with a very small overhead. Finally, it plays nicely with any of your other tools that take STDIN. The thing

I might like the most about it is that it's actually kind of fun. It isn't every day I get to rummage about in /usr/src/linux/include, and I learned a lot about Linux in the process.

```
#from an error message I got when I misspelled a struct name,
#the structs avail to stap in the vfs_(read|write) context are:
#f_u f_dentry f_vfsmnt f_op f_count f_flags f_mode f_pos f_owner
#f_uid f_gid f_ra f_version f_security private_data f_ep_links f_ep_lock f_mapping

function get_path:string (da:long, va:long) %{
      char *page = (char *)__get_free_page(GFP_ATOMIC);
      struct dentry *dentry = (struct dentry *)((long)THIS->da);
      struct vfsmount *vfsmnt = (struct vfsmount *)((long)THIS->va);
      snprintf(THIS->__retvalue, MAXSTRINGLEN, "%s", d_path(dentry, vfsmnt,\
         page, PAGE_SIZE));
      free_page((unsigned long)page);
%}

probe   kernel.function ("vfs_write"),
        kernel.function ("vfs_read")
{
      dev_nr = $file->f_dentry->d_inode->i_sb->s_dev
      path=get_path($file->f_dentry, $file->f_vfsmnt)

      subPath=substr(path,0,4)
      if((subPath != "/dev") && (dev_nr == (8 << 20 | 3)))
      printf ("%s(%d,%d) %s\n", execname(), pid(), uid(), path )
}
```

FIGURE 1: SAMPLE SYSTEMTAP SCRIPT

So let's step through this script starting with the function declaration in line 1:

```
function get_path:string (da:long, va:long) %{
```

As you can see, the function declaration is *not* C. The function is declared in the SystemTap language; embedded C blocks are denoted by %{ and %}. The second thing you might notice is that both variables are declared long even though, when the function is called below, it is passed pointers to structs. This is because all pointers are cast to longs by the interpreter, so they need to be declared as such in the function declaration and `typedef`'d back into struct pointers later. The entire purpose of this function is to call the `d_path()` function to return the full path of the file in question. So the next three lines set up the required arguments for `dpath()`:

```
char *page = (char *)__get_free_page(GFP_ATOMIC);
struct dentry *dentry = (struct dentry *)((long)THIS->da);
struct vfsmount *vfsmnt = (struct vfsmount *)((long)THIS->va);
```

There may have been a way to directly refer to the file's path with SystemTap built-ins, but if there is, I couldn't find it. The next two lines call `dpath()` and free the page we allocated:

```
snprintf(THIS->__retvalue, MAXSTRINGLEN, "%s", d_path(dentry, vfsmnt, \
   page, PAGE_SIZE));
free_page((unsigned long)page);
```

Below this function is the SystemTap script proper. The first two lines tell SystemTap that we are going to probe for `vfs_(write|read)` calls:

```
probe   kernel.function ("vfs_write"),
        kernel.function ("vfs_read")
```

Any number of comma-separated probes may be declared in a probe statement. The block immediately following the probe statement includes the instructions we want to carry out for each call we capture. In this script the first thing we do is dereference the device number of the current file:

```
dev_nr = $file->f_dentry->d_inode->i_sb->s_dev
```

The dentry struct is defined in /usr/src/linux/include/linux/dcache.h. You can directly reference anything in a struct from SystemTap without needing to resort to embedded C. It's generally preferable to avoid C when you can, because SystemTap uses kprobes' considerable safety and sanity checks as long as you stay within the bounds of its interpreted language. I should also save you some time and note that when you are dereferencing string data from kernel space in the SystemTap language, you need to use the `kernel_string()` conversion function or you'll end up with a `typedef'd long` once again. For example, we could directly dereference the name of the file from within the SystemTap language like so:

```
f_name=kernel_string($file->f_dentry->d_name->name)
```

Next we call our `get_path` function to derive the full path of the file:

```
path=get_path($file->f_dentry, $file->f_vfsmnt)
```

I placed some filters in here to give you a rough feel for the syntax you can use to filter probe data. Generally, the SystemTap language has all the iterative loops and conditionals you'd expect. In the line:

```
subPath=substr(path,0,4) if((subPath != "/dev") && (dev_nr == (8 << 20 | 3)))
```

the first check filters out changes to files in the /dev directory (`grep -v '^/dev'`). The second check filters out everything except files that reside on the third SCSI volume (/dev/sda3) as defined by the device number (major 8, minor 3). You can derive the device number for files on a given partition by `cd`'ing to that partition and performing a `stat -c '%D' *`. Without any filters you get *every* read and write happening on the system. If someone moved a mouse, for example, you'd see writes to /dev/psaux.

Finally, the `printf` built in prints our data to STDOUT:

```
printf ("%s(%d,%d) %s\n", execname(), pid(), uid(), path )
```

The first three arguments are also built-in functions: `execname` returns the name of the program making the change (xterm, vi, etc.), and `pid()` and `uid()` are self-explanatory. I placed the reads and writes in the same probe statement to show you it's possible, but if we wanted to differentiate reads from writes, our script could declare the probes separately, giving each its own instruction block. The read instruction block, for example, could have a `printf` that said `read: %s(%d,%d) %s\n`. We execute the script like so:

```
sudo stap -g figure1.stp
```

or, even better:

```
sudo stap -g figure1.stp | logger -t vfsprobe -p kern.info &
```

The `-g` is for "guru" mode, which allows the execution of embedded C. As I alluded to earlier, guru mode gives you the rope to hang yourself with by turning off quite a bit of sanity checking. This sort of thing should probably not be done lightly. If you are new to SystemTap and are considering running your code on production systems, I'd recommend running it by the gang on the SystemTap mailing list (as I did with this script).

I think kernel probe tools show a lot of potential to solve some of our nagging auditing needs. I've begun running a script like this one under dae-

montools [12] on a few of the boxes in our staging environment, with favorable results. I'm hoping it will eventually replace a few auditing tools we're using now, and I'd really like to expand it to include some other auditing gaps I have. It's not Blockbuster material, probably, but it is close enough to magic for the folks I hang out with.

Take it easy.

REFERENCES

[1] The "magic" entry in the jargon file: http://catb.org/~esr/jargon/html/M/magic.html.

[2] Homer Simpson's classic quotation: http://www.eventsounds.com/wav/cmputers.wav.

[3] *Forensic Discovery* by Dan Farmer and Wietse Venema: http://www.porcupine.org/forensics/forensic-discovery/.

[4] Tripwire: http://www.tripwire.com/.

[5] Samhain: http://la-samhna.de/samhain.

[6] Beagle: http://beagle-project.org/Main_Page/.

[7] See libinotifytools for a scriptable inotify implementation:

http://inotify-tools.sourceforge.net/api/index.html.

[8] ttysnoop: http://freshmeat.net/projects/ttysnoop/.

[9] Sebek: http://www.honeynet.org/tools/sebek/.

[10] DTrace: http://www.solarisinternals.com/wiki/index.php/DTrace_Topics_Limitations.

[11] SystemTap: http://sourceware.org/systemtap/.

[12] daemontools: http://cr.yp.to/daemontools.html.