opensnoop For Linux

25 Jul 2014

Here's another tool I didn't think would be possible: what files are being opened on my Linux system?

```
# ./opensnoop
Tracing open()s. Ctrl-C to end.
COMM             PID      FD FILE
[...]
sshd             6503     -1 /usr/share/ssh/blacklist.RSA-2048
sshd             6503     -1 /etc/ssh/blacklist.RSA-2048
sshd             6503    0x4 /root/.ssh/authorized_keys
sshd             6503     -1 /var/run/nologin
sshd             6503     -1 /etc/nologin
sshd             6503    0x4 /etc/passwd
sshd             6503    0x4 /etc/shadow
sshd             6503    0x4 /etc/localtime
sshd             6503    0x4 /etc/security/capability.conf
sshd             6503    0x4 /etc/login.defs
[...]
```

This is tracing all processes system-wide, and using a more efficient tracer than strace.

How about just files containing "conf", to see what config files are actively opened?

```
# ./opensnoop conf
Tracing open()s for filenames containing "conf". Ctrl-C to end.
COMM             PID      FD FILE
run              19107   0x3 /etc/nsswitch.conf
stat             19111   0x3 /etc/nsswitch.conf
webapp           19119   0x4 /home/webapp/3.7/rel/conf/logging.conf
webapp           19119   0x7 /etc/nsswitch.conf
webapp           19119   0x7 /etc/host.conf
webapp           19119   0x7 /etc/resolv.conf
webapp           19119   0x8 /home/webapp/3.7/rel/conf/application.properties
catalina.sh      19107   0x3 /etc/nsswitch.conf
run              19122   0x3 /etc/nsswitch.conf
[...]
```

Nice! So that's where my webapp config files are...

How about files ending in "log"?

```
# ./opensnoop 'log$'
Tracing open()s for filenames containing "log$". Ctrl-C to end.
COMM             PID      FD FILE
webapp           21144   0x4 /var/tmp/webapp/3.7/stash/webapp.log
webapp           21159   0x4 /var/tmp/webapp/3.7/stash/webapp.log
webapp           21174   0x4 /var/tmp/webapp/3.7/stash/webapp.log
bash             21301   0x4 /var/log/lastlog
[...]
```

Oh, no wonder I couldn't find them, I wasn't looking in /var/tmp!...

And open()s that returned an error (eg, file not found)?

```
# ./opensnoop -x
Tracing open()s. Ctrl-C to end.
COMM             PID      FD FILE
smtp             2690     -1 /usr/lib/tls/x86_64/libnss_db.so.2
smtp             2690     -1 /usr/lib/tls/libnss_db.so.2
smtp             2690     -1 /usr/lib/x86_64/libnss_db.so.2
smtp             2690     -1 /usr/lib/libnss_db.so.2
java             4680     -1 /home/tomcat/conf/Standalone/localhost
java             4680     -1 /home/tomcat/conf/Standalone/localhost
java             4680     -1 /home/tomcat/conf/Standalone/localhost
[...]
```

Catching a process hunt around its library path is normal, like we see here for smtp. But I wonder if that java app was supposed to be finding that file...

Options

opensnoop options are summarized by the USAGE message (there's also a man page and examples file):

```
# ./opensnoop -h
USAGE: opensnoop [-htx] [-d secs] [-p PID] [-n name] [filename]
                 -d seconds      # trace duration, and use buffers
                 -n name         # process name to match on I/O issue
                 -p PID          # PID to match on I/O issue
                 -t              # include time (seconds)
                 -x              # only show failed opens
                 -h              # this usage message
                 filename        # match filename (partials, REs, ok)
  eg,
       opensnoop                 # watch open()s live (unbuffered)
       opensnoop -d 1            # trace 1 sec (buffered)
       opensnoop -p 181          # trace I/O issued by PID 181 only
       opensnoop conf            # trace filenames containing "conf"
       opensnoop 'log$'          # filenames ending in "log"
```

The -p option employs an in-kernel filter for efficiency.

opensnoop traces events as they happen, which for very frequent open()s can begin to cost measurable overhead. The "-d" mode buffers and prints the buffer at the end, reducing overheads if needed.

What, Why, and How

This is another ftrace-based hack for my perf-tools collection.

There are some great ways to implement opensnoop on Linux using tracers such as perf_events (the "perf" command) with kernel debuginfo, or using SystemTap or ktap. But my aim is to use this now on a fleet of AWS EC2 cloud instances running Linux 3.2, and *without* kernel debuginfo (which is impractical to install everywhere in this dynamic environment, since it can be 100s of Mbytes).

The without-debuginfo requirement makes this especially tricky. One problem was that I didn't think ftrace (or perf_events) were able to dereference strings, having only seen examples with hex addresses. It turns out they do have this capability, and in fact, opensnoop can be a few one-liners:

```
# perf probe --add 'do_sys_open filename:string'
[...]
# perf record --no-buffering -e probe:do_sys_open -o - -a | PAGER=cat perf script -i -
 multilog 19075 [001] 5586.323974: probe:do_sys_open: (ffffffff811af8b0) filename_string="."
 multilog 19075 [001] 5586.324013: probe:do_sys_open: (ffffffff811af8b0) filename_string="./main"
 multilog 19075 [001] 5586.324028: probe:do_sys_open: (ffffffff811af8b0) filename_string="lock"
    snmpd  1255 [000] 5586.576142: probe:do_sys_open: (ffffffff811af8b0) filename_string="/proc/n
    snmpd  1255 [000] 5586.576278: probe:do_sys_open: (ffffffff811af8b0) filename_string="/proc/n
[...]
# perf probe --del do_sys_open
```

That's an example of dynamic tracing of the kernel function do_sys_open(), and examining an argument (filename) as a string. Except this needs kernel debuginfo to recognize the "filename" symbol.

Note that on older systems (including Linux 3.2), instead of --no-buffering it is -D.

Without kernel debuginfo, I can fashion a similar one-liner using CPU registers instead of symbols. I was struggling to get strings to work with this, and started thinking it wouldn't be possible, then asked for help on the perf-users mailing list. Fortunately, it is possible!

Instead of using registers on the entry of do_sys_open(), I ended up tracing the return value of getname() (which do_sys_open() calls), and processing that as either a char * or struct filename * depending on the kernel version. It's a brittle hack, but I thought this was a bit better than guessing the register for do_sys_open() on different platforms.

Conclusion

Tracing open()s can tell you a lot about running applications: the locations of their config, log, and data files, and file open errors. This is a Linux port of my popular opensnoop tool, which I've used for many years to help with performance analysis and troubleshooting.

This is also another proof of concept for older Linux kernels, like iosnoop, using the existing ftrace and kprobes tracing frameworks. If you happen to have kernel debuginfo, you can also just use the one-liners I included above. Warnings apply: opensnoop and those one-liners use dynamic tracing on Linux, which has had kernel panic bugs in the past, so know what you are doing, test first, and use at your own risk.

You can comment here, but I can't guarantee your comment will remain here forever: I might switch comment systems at some point (eg, if disqus add advertisements).