Linux Fu: Tracing System Calls

Originalartikel

Backup

<html> <p>One of the nice things about Linux and similar operating systems is that you can investigate something to any level you wish. If a program has a problem you can decompile it, debug it, trace it, and &#8212; if necessary &#8212; even dig into the source code for the kernel and most of the libraries the program is probably using. However, the tools to do this aren&#8217;t ones you use every day. One very interesting tool is

```
strace
```

. Using it you can see what system calls any program makes and that can sometimes give you important clues about how the program works or, probably more often, why it doesn&#8217;t work.</p> <p>Let&#8217;s consider the least complex use of the command. Suppose you want to make symlink from

```
testxmit.grc
```

to the

```
/tmp
```

directory. That command is simple:</p> <pre>ln -sf testxmit.grc /tmp</pre> <p>But if you tell

```
strace
```

to run it, the command becomes:</p> <pre>strace ln -sf testxmit.grc /tmp</pre> <p>You might want to redirect the output to a file using the shell or the

```
-o
```

option, though. Some commands generate a lot and often the first page or two of output isn&#8217;t really what you care about anyway.</p> <h2><strong>Let&#8217;s Look</strong></h2> <p>Let&#8217;s look at the output:</p> <pre>execve(„/bin/ln“, [„ln“, „-sf“, „testxmit.grc“, „/tmp“], 0x7fff51ddf6f8 /* 91 vars */) = 0 brk(NULL) = 0x562301ce6000 … openat(AT_FDCWD, „/usr/lib/locale/locale-archive“, O_RDONLY|O_CLOEXEC) = 3 fstat(3, {st_mode=S_IFREG|0644, st_size=3004096, …}) = 0 mmap(NULL, 3004096, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f7c45298000 close(3) = 0 <strong>stat(„/tmp“, {st_mode=S_IFDIR|S_ISVTX|0777, st_size=1360, …}) = 0</strong> lstat(„/tmp/testxmit.grc“, 0x7fff7ae555d0) = -1 ENOENT (No such file or directory) symlinkat(„testxmit.grc“, AT_FDCWD, „/tmp/testxmit.grc“) = 0 lseek(0, 0, SEEK_CUR) = -1 ESPIPE (Illegal seek) close(0) = 0 close(1) = 0 close(2) = 0 exit_group(0) = ? +++ exited with 0 +++</pre> <p>At the top, shortened here, are 25 lines dealing with loading shared libraries into the memory space and four lines dealing with locales. I put the first &#8220;real&#8221; line in bold, where the program calls stat on

```
/tmp
```

and then makes sure the file doesn&#8217;t already exist. Finally, you get to the real system call (

```
symlinkat
```

) followed by a few things to close the program out. A lot of work to get to one system call.</p> <p>You can probably figure out that the part on the left of the equal sign is the return value of the call. Usually, zero is success and other numbers mean different things. However, the

```
openat
```

call, for example, returns a file descriptor (3) and you can see it sent to

```
fstat
```

in the next line.</p> <h2><strong>In Practice</strong></h2> <p>Of course, the ln command works, but humor me and say we were wanting to understand what arguments passed to

```
symlinkat
```

. You could use the

```
-e
```

option to cut the output down to size:</p> <pre>strace -e symlinkat ln -sf testxmit.grc /tmp</pre> <p>You&#8217;ll notice something strange if you did the examples in order. The second time you run the command you get two calls to

```
symlinkat
```

. The first one fails because the file already exists. The second one is to some random file name. Taking off the

```
-e
```

lets you see everything (I&#8217;m only showing the interesting part):</p> <pre>symlinkat(„testxmit.grc“, AT_FDCWD, „/tmp/testxmit.grc“) = -1 EEXIST (File exists) openat(AT_FDCWD, „/dev/urandom“, O_RDONLY) = 3 read(3, „\337\336\10\324\254\233“, 6) = 6 close(3) = 0 getpid() = 29340 getppid() = 29338 getuid() = 1000 getgid() = 1000 symlinkat(„testxmit.grc“, AT_FDCWD, „/tmp/CuzoNWnv“) = 0 renameat(AT_FDCWD, „/tmp/CuzoNWnv“, AT_FDCWD, „/tmp/testxmit.grc“) = 0</pre> <p>Notice that the random part comes from reading some data from /dev/urandom. If you don&#8217;t want all that output, try:</p> <pre>strace -e synlinkat,renameat ln -sf testxmit.grc /tmp</pre> <h2><strong>Other Options</strong></h2> <p>The

```
-p
```

option lets you supply a PID of a running program. Sending the output to a file and then monitoring the file with a

```
tail -f
```

is a good trick. By default, you only see 32 bytes of the call data and that might not be enough. You can adjust that size with the

```
-s
```

option.</p> <p>So far we&#8217;ve only looked at simple programs. But if you want to trace multiple threads, check out the

```
-f
```

and

```
-ff
```

options.</p> <p>If you want a survey of what the program is calling, the

```
-c
```

option will give you a summary.</p> <pre>% time seconds usecs/call calls errors syscall —— ———- ———- ——— ——— ————-

```
0.00    0.000000           0         2           read
0.00    0.000000           0         7           close
0.00    0.000000           0         2           stat
0.00    0.000000           0         3           fstat
0.00    0.000000           0         1           lstat
0.00    0.000000           0         1         1 lseek
0.00    0.000000           0         6           mmap
...
0.00    0.000000           0         4           openat
0.00    0.000000           0         1           renameat
0.00    0.000000           0         2         1 symlinkat
```

—— ———- ———- ——— ——— ————- 100.00 0.000000 46 5 total</pre> <p><strong>Of Course, There&#8217;s More&#8230;</strong></p> <p>This is just one of many tools you can use to examine Linux programs. Debuggers are often useful, especially if you have the source. There are other tools to examine symbol tables and dump executables. But those are topics for another time.</p> <p>What&#8217;s your favorite Linux reverse engineering tool? Let us know in the comments.</p> </html>