Intercepting and Emulating Linux System Calls with Ptrace

June 23, 2018
nullprogram.com/blog/2018/06/23/

The `ptrace(2)` ("process trace") system call is usually associated with debugging. It's the primary mechanism through which native debuggers monitor debuggees on unix-like systems. It's also the usual approach for implementing strace[1] — system call trace. With Ptrace, tracers can pause tracees, inspect and set registers and memory[2], monitor system calls, or even *intercept* system calls.

By intercept, I mean that the tracer can mutate system call arguments, mutate the system call return value, or even block certain system calls. Reading between the lines, this means a tracer can fully service system calls itself. This is particularly interesting because it also means **a tracer can emulate an entire foreign operating system**. This is done without any special help from the kernel beyond Ptrace.

The catch is that a process can only have one tracer attached at a time, so it's not possible emulate a foreign operating system while also debugging that process with, say, GDB. The other issue is that emulated systems calls will have higher overhead.

For this article I'm going to focus on Linux's Ptrace[3] on x86-64, and I'll be taking advantage of a few Linux-specific extensions. For the article I'll also be omitting error checks, but the full source code listings will have them.

You can find runnable code for the examples in this article here:

https://github.com/skeeto/ptrace-examples

strace

Before getting into the really interesting stuff, let's start by reviewing a bare bones implementation of strace. It's no DTrace[4], but strace is still incredibly useful.

Ptrace has never been standardized. Its interface is similar across different operating systems, especially in its core functionality, but it's still subtly different from system to system. The `ptrace(2)` prototype generally looks something like this, though the specific types may be different.

```
long ptrace(int request, pid_t pid, void *addr, void *data);
```

The `pid` is the tracee's process ID. While a tracee can have only one tracer attached at a time, a tracer can be attached to many tracees.

The `request` field selects a specific Ptrace function, just like the `ioctl(2)` interface. For strace, only two are needed:

- `PTRACE_TRACEME`: This process is to be traced by its parent.
- `PTRACE_SYSCALL`: Continue, but stop at the next system call entrance or exit.
- `PTRACE_GETREGS`: Get a copy of the tracee's registers.

The other two fields, `addr` and `data`, serve as generic arguments for the selected Ptrace function. One or both are often ignored, in which case I pass zero.

The strace interface is essentially a prefix to another command.

```
$ strace [strace options] program [arguments]
```

My minimal strace doesn't have any options, so the first thing to do — assuming it has at least one argument — is `fork(2)` and `exec(2)` the tracee process on the tail of `argv`. But before loading the target program, the new process will inform the kernel that it's going to be traced by its parent. The tracee will be paused by this Ptrace system call.

```
pid_t pid = fork();
switch (pid) {
    case -1: /* error */
        FATAL("%s", strerror(errno));
    case 0:  /* child */
        ptrace(PTRACE_TRACEME, 0, 0, 0);
        execvp(argv[1], argv + 1);
        FATAL("%s", strerror(errno));
}
```

The parent waits for the child's `PTRACE_TRACEME` using `wait(2)`. When `wait(2)` returns, the child will be paused.

```
waitpid(pid, 0, 0);
```

Before allowing the child to continue, we tell the operating system that the tracee should be terminated along with its parent. A real strace implementation may want to set other options, such as `PTRACE_O_TRACEFORK`.

```
ptrace(PTRACE_SETOPTIONS, pid, 0, PTRACE_O_EXITKILL);
```

All that's left is a simple, endless loop that catches on system calls one at a time. The body of the loop has four steps:

1. Wait for the process to enter the next system call.
2. Print a representation of the system call.
3. Allow the system call to execute and wait for the return.
4. Print the system call return value.

The `PTRACE_SYSCALL` request is used in both waiting for the next system call to begin, and waiting for that system call to exit. As before, a `wait(2)` is needed to wait for the tracee to enter the desired state.

```
ptrace(PTRACE_SYSCALL, pid, 0, 0);
waitpid(pid, 0, 0);
```

When `wait(2)` returns, the registers for the thread that made the system call are filled with the system call number and its arguments. However, *the operating* *system has not yet serviced this system call*. This detail will be important later.

The next step is to gather the system call information. This is where it gets architecture specific. On x86-64, the system call number is passed in `rax`[5], and the arguments (up to 6) are passed in `rdi`, `rsi`, `rdx`, `r10`, `r8`, and `r9`. Reading the registers is another Ptrace call, though there's no need to `wait(2)` since the tracee isn't changing state.

```
struct user_regs_struct regs;
ptrace(PTRACE_GETREGS, pid, 0, &regs);
long syscall = regs.orig_rax;

fprintf(stderr, "%ld(%ld, %ld, %ld, %ld, %ld, %ld)",
        syscall,
        (long)regs.rdi, (long)regs.rsi, (long)regs.rdx,
        (long)regs.r10, (long)regs.r8,  (long)regs.r9);
```

There's one caveat. For internal kernel purposes[6], the system call number is stored in `orig_rax` rather than `rax`. All the other system call arguments are straightforward.

Next it's another PTRACE_SYSCALL and `wait(2)`, then another PTRACE_GETREGS to fetch the result. The result is stored in `rax`.

```
ptrace(PTRACE_GETREGS, pid, 0, &regs);
fprintf(stderr, " = %ld\n", (long)regs.rax);
```

The output from this simple program is *very* crude. There is no symbolic name for the system call and every argument is printed numerically, even if it's a pointer to a buffer. A more complete strace would know which arguments are pointers and use `process_vm_readv(2)` to read those buffers from the tracee in order to print them appropriately.

However, this does lay the groundwork for system call interception.

System call interception

Suppose we want to use Ptrace to implement something like OpenBSD's `pledge(2)`[7], in which a process *pledges* to use only a restricted set of system calls[8]. The idea is that many programs typically have an initialization phase where they need lots of system access (opening files, binding sockets, etc.). After initialization they enter a main loop in which they processing input and only a small set of system calls are needed.

Before entering this main loop, a process can limit itself to the few operations that it needs. If the program has a flaw[9] allowing it to be exploited by bad input, the pledge significantly limits what the exploit can accomplish.

Using the same strace model, rather than print out all system calls, we could either block certain system calls or simply terminate the tracee when it misbehaves. Termination is easy: just call `exit(2)` in the tracer. Since it's configured to also terminate the tracee. Blocking the system call and allowing the child to continue is a little trickier.

The tricky part is that **there's no way to abort a system call once it's started**. When tracer returns from `wait(2)` on the entrance to the system call, the only way to stop a system call from happening is to terminate the tracee.

However, not only can we mess with the system call arguments, we can change the system call number itself, converting it to a system call that doesn't exist. On return we can report a "friendly" EPERM error in `errno` via the normal in-band signaling[10].

```
for (;;) {
    /* Enter next system call */
    ptrace(PTRACE_SYSCALL, pid, 0, 0);
    waitpid(pid, 0, 0);

    struct user_regs_struct regs;
    ptrace(PTRACE_GETREGS, pid, 0, &regs);

    /* Is this system call permitted? */
    int blocked = 0;
    if (is_syscall_blocked(regs.orig_rax)) {
        blocked = 1;
        regs.orig_rax = -1; // set to invalid syscall
        ptrace(PTRACE_SETREGS, pid, 0, &regs);
    }

    /* Run system call and stop on exit */
    ptrace(PTRACE_SYSCALL, pid, 0, 0);
    waitpid(pid, 0, 0);

    if (blocked) {
        /* errno = EPERM */
        regs.rax = -EPERM; // Operation not permitted
        ptrace(PTRACE_SETREGS, pid, 0, &regs);
    }
}
```

This simple example only checks against a whitelist or blacklist of system calls. And there's no nuance, such as allowing files to be opened (`open(2)`) read-only but not as writable, allowing anonymous memory maps but not non-anonymous mappings, etc. There's also no way to the tracee to dynamically drop privileges.

How *could* the tracee communicate to the tracer? Use an artificial system call!

Creating an artificial system call

For my new pledge-like system call — which I call `xpledge()` to distinguish it from the real thing — I picked system call number 10000, a nice high number that's unlikely to ever be used for a real system call.

```c
#define SYS_xpledge 10000
```

Just for demonstration purposes, I put together a minuscule interface that's not good for much in practice. It has little in common with OpenBSD's `pledge(2)`, which uses a string interface[11]. *Actually* designing robust and secure sets of privileges is really complicated, as the `pledge(2)` manpage shows. Here's the entire interface *and* implementation of the system call for the tracee:

```c
#define _GNU_SOURCE
#include <unistd.h>

#define XPLEDGE_RDWR  (1 << 0)
#define XPLEDGE_OPEN  (1 << 1)

#define xpledge(arg) syscall(SYS_xpledge, arg)
```

If it passes zero for the argument, only a few basic system calls are allowed, including those used to allocate memory (e.g. `brk(2)`). The `PLEDGE_RDWR` bit allows various[12] read and write system calls (`read(2)`, `readv(2)`, `pread(2)`, `preadv(2)`, etc.). The `PLEDGE_OPEN` bit allows `open(2)`.

To prevent privileges from being escalated back, `pledge()` blocks itself — though this also prevents dropping more privileges later down the line.

In the xpledge tracer, I just need to check for this system call:

```c
/* Handle entrance */
switch (regs.orig_rax) {
    case SYS_pledge:
        register_pledge(regs.rdi);
        break;
}
```

The operating system will return `ENOSYS` (Function not implemented) since this isn't a *real* system call. So on the way out I overwrite this with a success (0).

```c
/* Handle exit */
switch (regs.orig_rax) {
    case SYS_pledge:
        ptrace(PTRACE_POKEUSER, pid, RAX * 8, 0);
        break;
}
```

I wrote a little test program that opens `/dev/urandom`, makes a read, tries to pledge, then tries to open `/dev/urandom` a second time, then confirms it can read from the original `/dev/urandom` file descriptor. Running without a pledge tracer, the output looks like this:

```
$ ./example
fread("/dev/urandom")[1] = 0xcd2508c7
XPledging...
XPledge failed: Function not implemented
fread("/dev/urandom")[2] = 0x0be4a986
fread("/dev/urandom")[1] = 0x03147604
```

Making an invalid system call doesn't crash an application. It just fails, which is a rather convenient fallback. When run under the tracer, it looks like this:

```
$ ./xpledge ./example
fread("/dev/urandom")[1] = 0xb2ac39c4
XPledging...
fopen("/dev/urandom")[2]: Operation not permitted
fread("/dev/urandom")[1] = 0x2e1bd1c4
```

The pledge succeeds but the second `fopen(3)` does not since the tracer blocked it with `EPERM`.

This concept could be taken much further, to, say, change file paths or return fake results. A tracer could effectively chroot its tracee, prepending some chroot path to the root of any path passed through a system call. It could even lie to the process about what user it is, claiming that it's running as root. In fact, this is exactly how the Fakeroot NG[13] program works.

Foreign system emulation

Suppose you don't just want to intercept *some* system calls, but *all* system calls. You've got a binary intended to run on another operating system[14], so none of the system calls it makes will ever work.

You could manage all this using only what I've described so far. The tracer would always replace the system call number with a dummy, allow it to fail, then service the system call itself. But that's really inefficient. That's essentially three context switches for each system call: one to stop on the entrance, one to make the always-failing system call, and one to stop on the exit.

The Linux version of PTrace has had a more efficient operation for this technique since 2005: `PTRACE_SYSEMU`. PTrace stops only *once* per a system call, and it's up to the tracer to service that system call before allowing the tracee to continue.

```
for (;;) {
    ptrace(PTRACE_SYSEMU, pid, 0, 0);
    waitpid(pid, 0, 0);

    struct user_regs_struct regs;
    ptrace(PTRACE_GETREGS, pid, 0, &regs);

    switch (regs.orig_rax) {
        case OS_read:
            /* ... */

        case OS_write:
            /* ... */

        case OS_open:
            /* ... */

        case OS_exit:
            /* ... */

        /* ... and so on ... */
    }
}
```

To run binaries for the same architecture from any system with a stable (enough) system call ABI, you just need this `PTRACE_SYSEMU` tracer, a loader (to take the place of `exec(2)`), and whatever system libraries the binary needs (or only run static binaries).

In fact, this sounds like a fun weekend project.

See also

- Implementing a clone of OpenBSD pledge into the Linux kernel

tags [linux x86 c bsd]

1. https://blog.plover.com/Unix/strace-groff.html
2. https://nullprogram.com/blog/2016/09/03/
3. http://man7.org/linux/man-pages/man2/ptrace.2.html
4. https://nullprogram.com/blog/2018/01/17/
5. https://nullprogram.com/blog/2015/05/15/
6. https://stackoverflow.com/a/6469069
7. https://man.openbsd.org/pledge.2
8. http://www.openbsd.org/papers/hackfest2015-pledge/mgp00001.html
9. https://nullprogram.com/blog/2017/07/19/
10. https://nullprogram.com/blog/2016/09/23/
11. https://www.tedunangst.com/flak/post/string-interfaces
12. https://nullprogram.com/blog/2017/03/01/
13. https://fakeroot-ng.lingnu.com/index.php/Home_Page
14. https://nullprogram.com/blog/2017/11/30/
15. https://lists.sr.ht/~skeeto/public-inbox

16. mailto:~skeeto/public-inbox@lists.sr.ht?
Subject=Re%3A%20Intercepting%20and%20Emulating%20Linux%20System%20Calls%20with%20Ptra
ce
17. https://lists.sr.ht/~skeeto/public-inbox?
search=Intercepting+and+Emulating+Linux+System+Calls+with+Ptrace
18. https://nullprogram.com/blog/comments/#2018-06-23