Made of Bugs

Write yourself an strace in 70 lines of code

Aug 29, 2010

Basically anyone who's used Linux for any amount of time eventually comes to know and love the `strace` command. `strace` is the system-call tracer, which traces the calls that a program makes into the kernel in order to interact with the outside world. If you're not already familiar with this incredibly versatile tool, I suggest you go check out my friend and coworker Greg Price's excellent blog post on the subject, and then come back here.

We all love strace, but have you ever wondered how it works? How does it interject itself between the kernel and the userspace program? This post will walk through a minimal implementation of `strace` in about 70 lines of C. It won't be nearly as functional as the real thing, but in the process you'll learn most of what you need to know about the core interfaces it uses.

On Linux (and probably some other UNIXes) `strace` uses a somewhat arcane interface known as `ptrace`, the process-tracing interface. `ptrace` allows one proccess to monitor the status of another process, and to inspect (or even manipulate) its internal state.

`ptrace` is a complex system call, taking a magic "request" first parameter, and doing completely different things depending on its value. Its general prototype looks like:

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

However, because different values of `request` use anywhere from zero to three of the remaining parameters, `glibc` prototypes it as a varargs function, allowing a a developer to only list as many parameters as a given call needs.

In order for one process to trace another, it attaches to that process, and temporarily becomes that process's parent. When a process is `ptrace`d, the tracer can ask for the child to stop whenever various events happen, such as the child making a system call. When this happens, the kernel will stop the child with `SIGTRAP`. Since the tracer is now the child's parent, it can thus watch for this using the standard UNIX `waitpid` system call.

Our miniature `strace` will only support the `strace COMMAND` form of `strace` (as opposed to `strace -p`), and we'll only print syscall numbers and return values – no decoding of names or arguments or anything. So a sample run might look like:

```
$ ./ministrace ls
…
syscall(6) = 0
syscall(54) = 0
syscall(54) = 0
syscall(5) = 3
syscall(221) = 1
syscall(220) = 272
syscall(220) = 0
syscall(6) = 0
syscall(197) = 0
syscall(192) = -1219706880
…
```

Not the most useful thing in the world, but it shows off the core tracing tools. So, let's see the code:

```
#include <sys/ptrace.h>
#include <sys/reg.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
```

We start with the necessary headers. `sys/ptrace.h` defines `ptrace` and the `__ptrace_request` constants, and we'll need `sys/reg.h` to help decode system calls. More about that later. Everything else you should probably recognize.

```
int do_child(int argc, char **argv);
int do_trace(pid_t child);

int main(int argc, char **argv) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s prog args\n", argv[0]);
        exit(1);
    }

    pid_t child = fork();
    if (child == 0) {
        return do_child(argc-1, argv+1);
    } else {
        return do_trace(child);
    }
}
```

We'll start with the entry point. We check that we were passed a command, and then we `fork()` to create two processes – one to execute the program to be traced, and the other to trace it.

```
int do_child(int argc, char **argv) {
    char *args [argc+1];
    memcpy(args, argv, argc * sizeof(char*));
    args[argc] = NULL;
```

The child starts with some trivial marshalling of arguments, since `execvp` wants a `NULL`-terminated argument array.

```
    ptrace(PTRACE_TRACEME);
    kill(getpid(), SIGSTOP);
    return execvp(args[0], args);
}
```

Next, we just execute the provided argument list, but first, we need to start the tracing process, so that the parent can start tracing the newly-executed program from the very start.

If a child knows that it wants to be traced, it can make the `PTRACE_TRACEME` `ptrace` request, which starts tracing. In addition, it means that the next signal sent to this process wil stop it and notify the parent (via `wait`), so that the parent knows to start tracing. So, after doing a `TRACEME`, we `SIGSTOP` ourselves, so that the parent can continue our execution with the `exec` call.

(You might have noticed that `strace COMMAND` output always starts with an `execve` call. Now you should understand why – we're actually going to start tracing immediately after the `kill` returns, so we see the `execve` call that starts the new program).

```
int wait_for_syscall(pid_t child);

int do_trace(pid_t child) {
    int status, syscall, retval;
    waitpid(child, &status, 0);
```

In the parent, meanwhile, we prototype a function we'll need later, and start tracing. We immediately `waitpid` on the child, which will return once the child has sent itself the `SIGSTOP` above, and is ready to be traced.

```
    ptrace(PTRACE_SETOPTIONS, child, 0, PTRACE_O_TRACESYSGOOD);
```

I mentioned earlier that `ptrace` turns basically all events into a `SIGTRAP` on the child. This is inconvenient because it means that when you see the child has stopped due to `SIGTRAP`, there's no good way to know which of several possible reasons it stopped for.

`PTRACE_SETOPTIONS` lets us set a number of options for how we want to trace the child. We use it here to set `PTRACE_O_TRACESYSGOOD`, which means that when the child stops for a syscall-related reason, we'll actually see it stopped with signal number `SIGTRAP | 0x80`, so we can easily distinguish syscall stops from other stops. Since (for the purposes of this demo), we only care about syscalls, this is very convenient.

```
    while(1) {
        if (wait_for_syscall(child) != 0) break;
```

Now we enter the tracing loop. `wait_for_syscall`, defined below, will run the child until either entry to or exit from a system call. If it returns non-zero, the child has exited and we end the loop.

```
        syscall = ptrace(PTRACE_PEEKUSER, child, sizeof(long)*ORIG_EAX);
        fprintf(stderr, "syscall(%d) = ", syscall);
```

Otherwise, though, we know that the child is entering a system call, and so we need to decode the system call number (and potentially arguments, if this were a less toy example). The `PTRACE_PEEKUSER` `ptrace` request reads a word of data from the child's "user area", which is a logical area that holds all of its registers and other internal non-memory state. On i386, the syscall number lives in `%eax`. For various technical reasons, however, the kernel has already clobbered the child's `%eax` at this point, but it saves the original value at a different offset, `ORIG_EAX`, which comes from `sys/regs.h'.

```
        if (wait_for_syscall(child) != 0) break;
```

Once we have the syscall number, we `wait_for_syscall` again, which should leave us stopped at the syscall return.

```
        retval = ptrace(PTRACE_PEEKUSER, child, sizeof(long)*EAX);
        fprintf(stderr, "%d\n", retval);
```

Return values on i386 are also passed in `%eax`, so this time we can read it directly and print the return value, and then return to the top of the loop to wait for the next syscall.

```
     }
    return 0;
}
```

And once the child exits, we just return.

```
int wait_for_syscall(pid_t child) {
    int status;
    while (1) {
        ptrace(PTRACE_SYSCALL, child, 0, 0);
```

`wait_for_syscall` is a simple helper function. We continue the child using `PTRACE_SYSCALL`, which allows a stopped child to continue executing until the next entry to or exit from a system call.

```
        waitpid(child, &status, 0);
```

We then `waitpid` to wait for something interesting to happen in the child.

```
        if (WIFSTOPPED(status) && WSTOPSIG(status) & 0x80)
            return 0;
```

Because of the `PTRACE_O_SYSGOOD` we set above, we can detect a syscall stop by checking if the child was stopped by a signal with the high bit set. If so, we return.

```
        if (WIFEXITED(status))
            return 1;
    }
}
```

If the child exited, we're done here; Otherwise, it stopped for some reason we don't care about (like an `execve`, for instance), and so we loop to start it again until it hits a syscall.

And that's all there is to it. You can find the version I just posted on github if you want to download and try it out.

Making it more useful

While it works, the previous version isn't exactly what I'd call particularly helpful. You have to decode the syscall numbers by hand, and you don't get any syscall arguments.

It's a little long to include in the post, but I've pushed a slightly more functional version to master in the same github repository. It includes a Python script to scan the Linux source to pick up syscall numbers and argument counts and types, and it knows how to decode string arguments, so that you can see filenames and `read` and `write` data.

Reading arguments is easy – on i386, they're passed in registers, so it's just another `PTRACE_GETUSER` for each argument. Perhaps the most interesting piece is the `read_string` function, which is used to read a NULL-terminated string from the child process. (Of course, NULL-terminated isn't quite right – the real strace knows about the `count` arguments to `read()` and `write()`, for instance. But it's close enough for a demo):

```
char *read_string(pid_t child, unsigned long addr) {
```

`read_string` takes a child to read from, and the address of the string it's going to read.

```
    char *val = malloc(4096);
    int allocated = 4096, read;
    unsigned long tmp;
```

We need some variables. A buffer to copy the string into, counters of how much data we've copied and allocated, and a temporary variable for reading memory.

```
    while (1) {
        if (read + sizeof tmp > allocated) {
            allocated *= 2;
            val = realloc(val, allocated);
        }
```

We grow the buffer if necessary. We read data one word at a time.

```
        tmp = ptrace(PTRACE_PEEKDATA, child, addr + read);
        if(errno != 0) {
            val[read] = 0;
            break;
        }
```

`PTRACE_PEEKDATA` returns a work of data from the child at the specified offset. Because it uses its return for the value, we need to check `errno` to tell if it failed. If it did (perhaps because the child passed an invalid pointer), we just return the string we've got so far, making sure to add our own NULL at the end.

```
        memcpy(val + read, &tmp, sizeof tmp);
        if (memchr(&tmp, 0, sizeof tmp) != NULL)
```

```
            break;
        read += sizeof tmp;
```

Then it's a simple matter of appending the data we read, and breaking out if we found a terminating NULL, or else looping to read another word.

```
    }
    return val;
}
```

« Navigating the Linux Kernel How is duct tape like the force? »