# 10

# *Lab 8: Tracing programs*

*This short assignment introduces you to a tool widely used in debugging and digital forensics:* `strace`*. The* `strace` *tool lets a user eavesdrop on all inputs and outputs from a target program. Such information, called a "trace" can be used for debugging: is my program really reading the right file? Traces can also be used for security: is this program accessing files it should not be?*

## *10.1 Learning Goals*

In this lab, you will practice:

- writing simple programs that do I/O and observing their traces;
- understanding the system call boundary in an operating system; and
- using strace to perform a "black box" analysis on a program.

## *10.2 Requirements*

*Collaboration*. This is an ungraded assignment. You are encouraged to work with a partner.

*Platform*. This assignment must be completed on your Raspberry Pi, as `strace` is only available on Linux.

## 10.3 *Part 1: A program that does nothing. Or does it?*

Let's start with a simple program, `prog1.c`, that does not obviously read or write anything.

```
int main() {
  return 127;
}
```

Compile the above code in the usual way and run it, then check its output using the following.

```
$ ./prog1
$ echo $?
127
```

You should see the return value of 127. Now let's run this program under `strace`.

```
$ strace ./prog1
```

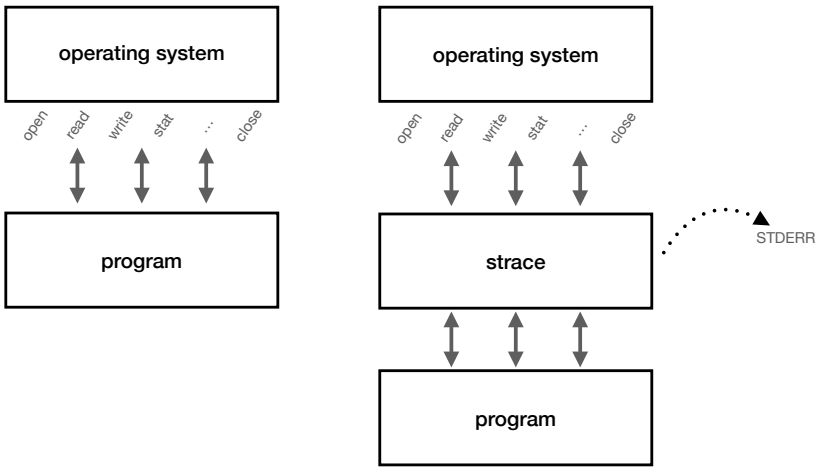
You should see a lot of output, something like this:

```
execve("./step1", ["./step1"], 0xbefff6b0 /* 22 vars */) = 0
brk(NULL)                               = 0x22000
uname({sysname="Linux", nodename="raspberrypi", ...}) = 0
access("/etc/ld.so.preload", R_OK)      = 0
openat(AT_FDCWD, "/etc/ld.so.preload", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=54, ...}) = 0
mmap2(NULL, 54, PROT_READ|PROT_WRITE, MAP_PRIVATE, 3, 0) = 0xb6ffc000
close(3)                                = 0
readlink("/proc/self/exe", "/home/pi/Documents/Code/strace_1"..., 4096) = 41
openat(AT_FDCWD, "/usr/lib/arm-linux-gnueabihf/libarmmem-v6l.so", O_RDONLY|O_LARGEFILE|O_CLOEXEC) =
     3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0(\0\1\0\0\0\250\3\0\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0644, st_size=9512, ...}) = 0
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb6ffa000
mmap2(NULL, 73772, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb6fbb000
mprotect(0xb6fbd000, 61440, PROT_NONE)  = 0
mmap2(0xb6fcc000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) = 0
    xb6fcc000
close(3)                                = 0
munmap(0xb6ffc000, 54)                  = 0
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=39242, ...}) = 0
mmap2(NULL, 39242, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb6ff0000
close(3)                                = 0
openat(AT_FDCWD, "/lib/arm-linux-gnueabihf/libc.so.6", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0(\0\1\0\0\0\274x\1\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1296004, ...}) = 0
mmap2(NULL, 1364764, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb6e6d000
mprotect(0xb6fa5000, 65536, PROT_NONE)  = 0
mmap2(0xb6fb5000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x138000) =
    0xb6fb5000
mmap2(0xb6fb8000, 8988, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0
    xb6fb8000
close(3)                                = 0
set_tls(0xb6ffaf40)                     = 0
mprotect(0xb6fb5000, 8192, PROT_READ)   = 0
mprotect(0xb6fcc000, 4096, PROT_READ)   = 0
```

```
mprotect(0x20000, 4096, PROT_READ)      = 0
mprotect(0xb6ffe000, 4096, PROT_READ)   = 0
munmap(0xb6ff0000, 39242)               = 0
exit_group(127)                         = ?
+++ exited with 127 +++
```

There's a lot of information there, so let's step back and discuss what we're looking at. What `strace` gives you is a *system call trace*. A *system call* is a fundamental operation in an operating system. The purpose of an operating system is to provide an abstraction between hardware and software. The OS, and to a lesser extent, a programming language, are what make it possible to call `open` and `read` to read from a file without having to worry about whether that file is on a spinning magnetic disk, a solid state disk, or a network-mounted file share. The tradeoff is that all input and output operations that a program performs must be handled by the operating system.[1]

`strace` is able to produce a system call trace by *interposing* on the system call interface between a program and the operating system. Interposition is when a program inserts itself between two other programs, intercepting calls from one made to the other. `strace` intercepts all system calls, writing them out as log messages on `stderr`.




Since `strace` writes messages to `stderr`, you can capture them to a file by running it like so:

```
$ strace ./prog1 2> trace.txt
```

where the `2>` instructs the operating system to redirect `stderr` to the file, `trace.txt`.

The important takeaway from our trace above is that every line is a system call made by `prog1`. And as we see, although our `prog1.c` program does not read or write to anything, the program does indeed perform some I/O. We aren't going to dig in deeply into why, exactly, this program performs these operations except to say that what you see in this trace is what every program, at some level must do in order to start

[1] Calling the operating system performs what is called a *context switch*. A context switch sets aside the running program to perform work in the kernel. To do this safely, the operating system must invalidate the processor's caches, switch the processor from *user mode* to *kernel mode*, and perform a substantial amount of bookkeeping work. Context switches are costly.

up. Key parts shown in the trace are where the shell starts the program (`execve`), where the stack is initialized (`brk`), where the program loads the program loader (`openat` of `ld.so.preload` through `mmap2` where the loader's `TEXT` section is placed in memory), and where the C runtime itself is loaded (`openat` of `libc.so.6`, etc.). You can also see which sections of the program are marked read-only in order to prevent program corruption and control-flow attacks (`mprotect`).

## 10.4 *Part 2: A program that really does something.*

Let's move on to a dangerous program, `prog2`. To make this interesting, you do not have access to `prog2`'s source code. Instead, download the `prog2` binary from the course website.

```
$ wget https://williams-cs.github.io/cs331-f21-www/assets/labs/lab8/prog2
$ chmod +x prog2
```

This program is already compiled. Just run it using `strace`.

```
$ strace ./prog2 2> trace2.txt
```

What does the program try to do? Use `strace` to find out.

This program is destructive. The only reason this program is "safe" to use in this lab is that its destructive actions will not succeed when run as a normal, unprivileged user. *Do not run this program with* `sudo`!

## 10.5 *Part 3: Tracing a program that launches other programs.*

Finally, `strace` is a flexible program that can perform many functions, but one of its most useful functions is to trace a program *and all the programs it launches.* To do that, we call `strace` with the `-f` flag.

Recall one of our earlier labs where we explored how to call each component of the compiler separately? In other words, instead of calling `gcc`, we called the C translator, `cc1`, the assembler, `as`, and the linker `ld`. Ever wonder how I figured out how to call those subprograms? It wasn't because I read a book. I used `strace` to learn how `gcc` itself called them so that I could observe what it did.

```
$ strace -f gcc -Wall prog1.c -o prog1 2> gcctrace.txt
```

The captured trace, `gcctrace.txt` contains all the information you need to figure this out yourself. Be aware that `gcctrace.txt` contains *a lot* of information! But by focusing our attention on the right things, we can learn a lot. Let's filter out everything that isn't a program launch. Program launches are done with the `execve` system call.

```
$ grep execve gcctrace.txt
```

Focusing our attention like this reveals the programs I described above, `cc1`, `as`, and `ld`.[2] This is how I "reversed engineered" the actions of the `gcc` wrapper program, without having to read any of its source code.

[2] And the mystery program, `collect2`. Bonus: what does `collect2` do?

Mastering `strace` requires a little practice—and familiarity with the system call interface can help a lot—but using it can reveal a lot about what a program does. This is particularly useful if you want to establish a baseline to see what an uncorrupted program *should* do. This information can be used to minimize the damage a program can do by restricting the syscalls that a program can perform. For example, the `pledge`[3] utility from OpenBSD takes a list of permitted system calls; any program that calls a syscall not on the list is immediately terminated. `pledge` can be coupled with `unveil`[4] to restrict access to specified parts of the filesystem, and `strace` can also help determine which files are part of a program's normal operation. Sadly, `pledge` and `unveil` are not available on Linux, however efforts are underway to bring similar features to Linux.[5].

[3] https://man.openbsd.org/pledge.2

[4] https://man.openbsd.org/unveil.2

[5] For example, Landlock: https://landlock.io/

## 10.6 *Part 4: ptrace*

This lab is just a little taste of what you can do with program tracing utilities like `strace`. Have a look at the `man` page to see what else you can do.

Other operating systems also have similar utilities. The `macOS` has a utility called `dtrace`[6] and a popular option on Windows is the Process Monitor tool from Sysinternals.

[6] Running tracing tools in the macOS requires disabling the operating systems's *system integrity protection* feature. Be warned!

The `strace` tool is built on a much more powerful interface called `ptrace`, which is built into most UNIX operating systems.[7] `ptrace` allows programmers not to just intercept and log system calls like `strace`, but to actually intercept and *change* system call results. Although a reference monitor is best implemented as a part of a kernel's design, one could implement a lightweight reference monitor using `ptrace`. Amazingly, `ptrace` requires no special privileges—it runs entirely at the privilege level of the user. Although `gdb` does not use `ptrace` for portability reasons[8], one could use `ptrace` to build a debugger.

[7] `ptrace` is a part of POSIX, the UNIX standard.

[8] For example, `gdb` runs on Windows, which does not have `ptrace`.

If you're curious, have a look at `man ptrace`.