Exercise 2: Finding syscalls

As we've already discussed, syscalls are the only way processes have to talk to the world outside their abstraction box. It's how they ask the OS to perform operations outside their sandbox. As such, if you could peer into the syscalls that a process performs, you'd probably be able to figure out what said process is doing overall, if it's sending data over the network, reading or modifying certain files, and many other things. That's what we're gonna be doing next.

Setup

For this exercise, we'll be using the `pementorship/w3_ex2` image. Again, if you can't remember how to start it:

```
$ sudo docker run -it pementorship/w3_ex2
```

Mystery binary

Let's take a look at what we provided you in this image:

```
[me@35a412f61df0 ~]$ ls
Makefile  mystery.c
```

Alright, so we have a c program and a Makefile to build it. Let's see if we can figure out what that program does without looking at the code. Let's build and run it:

```
[me@35a412f61df0 ~]$ make
gcc mystery.c -o mystery
[me@35a412f61df0 ~]$ ./mystery
[me@35a412f61df0 ~]$
```

Huh... Did it even do anything? We see no output, and even if we take a look at the directory where we ran this:

```
[me@35a412f61df0 ~]$ ls
Makefile  mystery  mystery.c
```

No new files!

Strace

So, how do we figure out what a binary does? We use `strace`. `strace` lets you look at what system calls a given process makes in real time. Let's run the program using that:

```
[me@35a412f61df0 ~]$ strace ./mystery
execve("./mystery", ["./mystery"], 0x7ffec9caad60 /* 10 vars */) = 0
brk(NULL)                               = 0x255e000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffcb1ecf580) = -1 EINVAL (Invalid argument)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=28613, ...}) = 0
mmap(NULL, 28613, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f3d64cf1000
close(3)                                = 0
openat(AT_FDCWD, "/lib64/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
```

```
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\2607\2\0\0\0\0\0"..., 832) =
fstat(3, {st_mode=S_IFREG|0755, st_size=4176104, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f3d64cc
mmap(NULL, 3938144, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f3d647
mprotect(0x7f3d648c7000, 2093056, PROT_NONE) = 0
mmap(0x7f3d64ac6000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE
mmap(0x7f3d64acc000, 14176, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS
1, 0) = 0x7f3d64acc000
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7f3d64cf0500) = 0
mprotect(0x7f3d64ac6000, 16384, PROT_READ) = 0
mprotect(0x600000, 4096, PROT_READ)     = 0
mprotect(0x7f3d64cf8000, 4096, PROT_READ) = 0
munmap(0x7f3d64cf1000, 28613)           = 0
brk(NULL)                               = 0x255e000
brk(0x257f000)                          = 0x257f000
brk(NULL)                               = 0x257f000
openat(AT_FDCWD, "/tmp/data.txt", O_RDWR|O_CREAT|O_TRUNC, 0666) = 3
fstat(3, {st_mode=S_IFREG|0664, st_size=0, ...}) = 0
write(3, "Hello darkness my old friend...\n", 32) = 32
close(3)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

(make sure you're also running this to see how the output spews out).

That's... A lot of syscalls. Let's see if we can figure out what's important here:

```
execve("./mystery", ["./mystery"], 0x7ffec9caad60 /* 10 vars */) = 0
```

This syscall starts up the binary, loading it into the current process, hence why it's the first thing you see.

There's a lot of things happening at the top of the file that we don't really care about. It's loading in dynamic libraries, doing some memory setup, among other things. We're gonna skip those and take a look at what's going on near the end:

```
openat(AT_FDCWD, "/tmp/data.txt", O_RDWR|O_CREAT|O_TRUNC, 0666) = 3
fstat(3, {st_mode=S_IFREG|0664, st_size=0, ...}) = 0
write(3, "Hello darkness my old friend...\n", 32) = 32
close(3)                                = 0
exit_group(0)                           = ?
```

This is the bulk of the program. It's relatively easy to see from here what we're doing; we run `openat()` to open the file `/tmp/data.txt`, and it returns a file descriptor of 3 (we'll ignore the other parameters for now).

We later use `fstat` to get information on the file, and most importantly, we write out "Hello darkness my old friend...\n" to that same file, finally calling `close()` on it and exiting entirely with `exit_group()`.

So, simply put, this program writes the string "Hello darkness my old friend...\n" to the file "/tmp/data.txt". Let's confirm:

```
[me@35a412f61df0 ~]$ cat /tmp/data.txt
Hello darkness my old friend...
```

And indeed the data is there!

What next?

`strace` is an extremely useful tool to keep in your debugging belt. It lets you figure out how other programs work and what they're trying to do without ever having to take a look at the source code. You can also attach strace on already running programs using the `-p` flag, by providing a PID. This is extremely powerful, but it doesn't come for free. `strace` does incur a significant performance hit on any program you attach it to because it essentially needs to pause the program every time a syscall is made. Because of that, always be careful when using this tool in production.

Next up, take a look at `problems.pdf` for some more complicated examples whose solutions you can discuss with your mentor!