Project 1: System Call

- **Handed out:** Friday, August 26, 2022
- **Due:** Friday, September 9, 2022

Introduction

The goal of this project is to learn Linux development environment and modify Linux kernel source code. The project is split into two parts. The first part is analyzing a given user-space socket application and adding *printk()* to the system call entries in Linux kernel. The second part is to add one new system call which encrypts a given string and prints out encrypted string to kernel message.

Make a folder named with your SBU ID (e.g., 112233445), put all your submission files in the folder, create a single gzip-ed tarball named `[SBU ID].tar.gz`, and turn the gzip-ed tarball to Brightspace.

```
$  tar czvf 112233445.tar.gz 112233445/
112233445/
112233445/code-socket.tar.gz
112233445/printk.patch
112233445/printk.png
112233445/syscall.patch
112233445/syscall.tar.gz
112233445/syscall.png
```

Recommended Background Reading

- Socket Programming in C/C++
- GNU Make in Detail for Beginners
- printk()
- git diff
- ERRNO(3)
- GETOPT(3)

Part 1. Adding `printk()`

The attached source code `code-socket.tar.gz` is a simple socket application that exchanges messages between a server and client. It consists of three files as shown in below.

```
$ cd code-socket
$ ls
client.c  Makefile  server.c
```

P1.1: Understanding source code

[10 points] Carefully read above three source file to understand how a simple network client/server works. Once you completely understand each file, add comments of each line (M1-M10, S1-S9, and C1-C7) explaining what the line means. Turn in the gzip-ed tarball named `code-socket.tar.gz`.

```
$  tar czvf code-socket.tar.gz code-socket/
code-socket/
code-socket/client.c
code-socket/server.c
code-socket/Makefile
```

P1.2: Adding `printk()`

[10 points] Now you understand how the user-space application works. To understand how a kernel system call is called, print any message at the very beginning of system call implementations of `accept()` and `connect()` in Linux kernel v5.15. Check if the modified kernel prints out messages you added when you run the network client/server. Then create a patch against kernel v5.15 using `git diff` command and turn in the patch named `printk.patch`.

P1.3: Test your kernel

[5 points] Take the screenshot of your kernel debug message using `dmesg` while running the network client/server. Run `server`, `client`, and `dmesg` in one ssh session using `tmux`. Turn in the screenshot named `printk.png`.

Part 2: Adding a new system call

Add a new system call named `sys_s2_encrypt()` that takes two arguments, a NULL-terminated string and an encryption key, which is a positive integer between 1 and 5. The system call encrypts the given string by simply adding the given integer number. For example, it encrypts "hello" to "ifmmp" by adding 1 to each character of the string. After encryption, it prints out the encrypted string using `printk()`. The system call returns 0 when everything is okay. If the encryption key is out of bound, it returns `EINVAL`.

P2.1: Implementing the system call

[10 points] You should implement the system call in a separate file under linux/kernel directory. Then create a patch against kernel v5.15 using `git diff` command and turn in the patch named `syscall.patch`.

P2.2: Writing a test program

[10 points] Write a simple test program takes two options, `-s string -k key` and calls `sys_s2_encrypt()` with the string and key from the command line. You should implement command line argument processing using GETOPT(3) and the code should be able to build using `make` and clean using `make clean`. The program prints out the return value of `sys_s2_encrypt()`. Turn in gzip-ed tarball of the source code and Makefile, named `syscall.tar.gz`.

P2.3: Test your system call

[5 points] Take a screenshot of your kernel debug message using `dmesg` while running your test program. Run your test program and `dmesg` in one ssh session using `tmux`. Turn in the screenshot named `syscall.png`.