SysCall Exploration

1 Introduction

In this assignment, you will bridge the gap between User Space (where your programs run) and Kernel Space (where the operating system manages hardware).

High-level programming languages use standard libraries (like standard I/O in C) to perform tasks such as reading files or printing to the screen. Under the hood, these libraries must ask the Operating System kernel to perform these privileged actions on their behalf. The mechanism for this request is called a **System Call** (syscall).

You will use two powerful tools standard on Linux systems:

- **objdump**: A tool that displays information about object files (executables). We will use it to view the assembly code of your program without running it (Static Analysis).
- **GDB (GNU Debugger)**: A tool that allows you to run your program step-by-step, even one assembly instruction at a time, to watch CPU registers change in real-time (Dynamic Analysis).

1.1 Prerequisites

This assignment must be done in a Linux environment (standard Ubuntu/Debian VM or WSL2 is acceptable) with the 'build-essential' and 'gdb' packages installed.

To install prerequisites, run:

```
$ sudo apt update
$ sudo apt install build-essential gdb manpages-dev
```

2 The Test Program

We will first write a simple program that makes some system calls. We will use two different ways to print text to the screen to illustrate different layers of abstraction:

- `printf()`: A high-level standard C library function that handles formatting and buffering for you.
- `write()`: A low-level POSIX function that maps very directly to the kernel's system call, doing exactly what you tell it to do with no extra help.

2.1 Write the Code

Create a file named `syscalldemo.c` with the following content:

```
1 #include <unistd.h>
2 #include <string.h>
```

```c
#include <stdio.h>

int main() {
    char *msg = "Hello, Kernel!\n";
    int len = strlen(msg);

    // Using printf (standard C library, high level)
    printf("First via standard library, then print %d another way\n", len)
    ;

    // Using write() directly to stdout (file descriptor 1)
    // Arguments: write(fd, buffer, count)
    ssize_t bytes_written = write(1, msg, len);

    return 0;
}
```

Listing 1: syscalldemo.c

2.2 Compile with Debug Symbols

Next, let's compile the program using gcc with the `-g` flag, which tells gcc to include *debug symbols*. These symbols allow GDB to know which line of C source code corresponds to which chunk of assembly machine code.

Run the following in your terminal:

```
$ gcc -g -o syscalldemo syscalldemo.c
```

Verify it works:

```
$ ./syscalldemo
First via standard library, then print 15 another way
Hello, Kernel!
```

3 Static Analysis with objdump

Before running it, let's look at the executable file to see how the compiler set up the call to `write()`.

3.1 Disassembly

Run the following command to disassemble the binary. We'll pipe it to `less` so you can scroll through it.

```
$ objdump -d -M intel syscalldemo | less
```

(Note: `-M intel` displays assembly in Intel syntax, which is generally easier to read than the default AT&T syntax. e.g., `mov destination, source`)

3.2 Locate Main (6 points)

In `less`, type `/main` and press Enter to search for the main function. You'll have to look for the correct spot - the word main appears more than once. Look for something that looks like this (addresses will vary):

```
0000000000001149 <main>:
    1149:       55                      push   rbp
    114a:       48 89 e5                mov    rbp,rsp
    114d:       48 83 ec 10             sub    rsp,0x10
    ...
```

Scroll down until you see the calls to `printf` and `write`. They won't call the functions directly; they will likely call a "PLT" (Procedure Linkage Table) entry, looking something like this:

```
    118e:       e8 cd fe ff ff          call   1060 <write@plt>
```

What to do

1. (2 points) Submit a screen shot of the disassembly lines that include the PLT calls.
2. (2 points) What is the exact instruction used to call `printf` in your specific binary? What is the hexadecimal address of that instruction?
3. (2 points) What is the exact instruction used to call `write` in your specific binary? What is the hexadecimal address of that instruction?

4 Dynamic Analysis with GDB

Now we will watch the system call happen in real-time.

4.1 Loading GDB

Start GDB with your program:

```
$ gdb -q ./syscalldemo
Reading symbols from ./syscalldemo...
(gdb)
```

Allow GDB to retrieve sources from online if it asks.

4.2 Setting Initial Breakpoints

We will start by pausing at 'main'.

```
(gdb) break main
Breakpoint 1 at 0x1155: file syscalldemo.c, line 6.
(gdb) run
```

```
Starting program: /home/student/syscalldemo

Breakpoint 1, main () at syscalldemo.c:6
6           char *msg = "Hello, Kernel!\n";
```

The program is now paused at the very beginning of main.

4.3 Peeking inside printf (3 points)

Before we look at the simple 'write' syscall, let's see why we aren't analyzing 'printf'. Advance execution to the 'printf' line by typing `next` (or `n`):

```
(gdb) next
7           int len = strlen(msg);
(gdb) next
10          printf("First via standard library, then print %d another way\
   n", len);
```

Now, let's step **inside** 'printf' using `stepi` (or `si`). You may need to do this several times to get past the PLT (Procedure Linkage Table) wrapper, just like we will later with 'write'.

```
(gdb) si
0x0000555555555050 in printf@plt ()
(gdb) si
... (repeat until you see __vfprintf_internal or similar) ...
(gdb) si
__vfprintf_internal (s=0x7ffff7e045c0 <_IO_2_1_stdout_>, format=0
   x555555556018 "First via standard library, then print %d another way\n
   ", ap=ap@entry=0x7fffffffdda0, mode_flags=mode_flags@entry=0)
    at ./stdio-common/vfprintf-internal.c:1522
```

Now that you are inside the real 'printf' function, let's look at the assembly code for it:

```
(gdb) disassemble
```

You will see a huge amount of assembly code. 'printf' is complex! It handles formatting (%d, %s), buffering (holding data until it has enough to be efficient), and locking (for multithreading). Finding the actual syscall instruction in here is difficult.

What to do

1. (3 points) Submit a screen shot of the printf disassembly

Let's escape from 'printf' and get back to our simple main function. Use the 'finish' command until we're back in main:

```
(gdb) finish
Run till exit from #0  0x00007ffff7dfc104 in __printf (
    format=0x555555556018 "First via standard library, then print %d
       another way\n") at ./stdio-common/printf.c:28
First via standard library, then print 15 another way
```

```
main () at syscalldemo.c:15
15          ssize_t bytes_written = write(1, msg, len);
```

We have now returned to 'main' and are sitting at the 'write' function call.

4.4 Stepping into the write wrapper

We are now at the precipice of the 'write' call. Unlike 'printf', this is a thin wrapper around the kernel call. Use `si` to enter the function.

```
(gdb) si
0x0000555555555060 in write@plt ()
```

You are now in the PLT linker stub. Keep typing `si` and pressing enter. You will eventually land in the real function:

```
(gdb) si
write () at ../sysdeps/unix/syscall-template.S:78
```

4.5 The Syscall Assembly (4 points)

Now that we are inside the real `write` function in libc, let's look around.

```
(gdb) disassemble
Dump of assembler code for function __GI___libc_write:
=> 0x00007ffff7eb8590 <+0>:     endbr64
   0x00007ffff7eb8594 <+4>:     cmpb   $0x0,0xeeaa5(%rip)        # 0
      x7ffff7fa7040 <__libc_single_threaded>
   0x00007ffff7eb859b <+11>:    je     0x7ffff7eb85b0 <__GI___libc_write
      +32>
   0x00007ffff7eb859d <+13>:    mov    $0x1,%eax
   0x00007ffff7eb85a2 <+18>:    syscall
   ...
```

What to do

1. (2 points) Submit a screen shot of the complete disassembly of the write function (it might be called `__GI___libc_write`).
2. (2 points) Compare the 'disassemble' output of 'write' to the one you saw for 'printf'. Approximately how many lines of assembly does the 'write' wrapper function have compared to 'printf'?

Use `si` repeatedly until the little arrow `=>` is pointing immediately at the `syscall` instruction.

4.6 Inspecting Registers before the jump (10 points)

STOP immediately before executing the `syscall` instruction.

On x86-64 Linux, system calls use specific registers to pass arguments to the kernel:

- **RAX**: Holds the System Call Number (tells the kernel *which* action to take).
- **RDI**: First argument (Standard Output file descriptor, which is 1).
- **RSI**: Second argument (Address of our message buffer).
- **RDX**: Third argument (Length of message, which is 15).

Let's verify this. Run:

```
(gdb) info registers rax rdi rsi rdx
```

What to do

1. (2 points) Submit a screen shot of the output from the 'info registers' command.
2. (3 points) What is the value in RAX? Look up a "Linux x64 syscall table" online (*e.g.,* `https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/`). Does this value correspond to `sys_write`? Explain.
3. (3 points) Does RDI match the file descriptor we passed to 'write' (1)? Explain.
4. (3 points) RDX should be the length of "Hello, Kernel!\n". Does it match? Explain.

4.7 Crossing the boundary (2 points)

Now, take the final step. This single instruction causes the CPU to switch to privileged mode and jump into the kernel.

```
(gdb) si
Hello, Kernel!
```

You should see "Hello, Kernel!" appear on your screen. The kernel performed the operation and returned control to your program.

Check the return value. The kernel returns standard values in RAX.

```
(gdb) info registers rax
```

It should show the number of bytes successfully written.

What to do

1. (2 points) Submit a screen shot of the output from the 'info registers' command.

4.8 Exiting

You can now type `continue` to let the program finish, and `quit` to exit GDB.