



*Never use any of the provided code on a network connected to the Internet.*

## 1. Prerequisites

- (a) Disable address space randomization

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

- (b) Building the vulnerable x64 C program

```
$ gcc -z noexecstack -fno-stack-protector -o prog prog.c
```

- (c) Running the program

```
$ ./prog payload
```

## 2. Tasks

Your task is to build three ROP chains for the x64 program to perform specific operations. Recall that ROP chains are useful when the stack region is not executable, and thus, the attacker cannot run shellcode from the stack. In this lab, you can search for the ROP gadgets in the generated binary and its dependencies.

You should set the `BUF_SIZE` to be `300+x`, where `x` is the least significant two digits in your SFU ID. If these two digits are zeros, choose the next significant two digits.

**Note 1:** You should not use any automation tools to generate the ROP chains, but you may use tools to search for gadgets.

**Note 2:** You need to explain in detail how you generate the ROP chains. Specifically, your lab report should include all steps you performed to generate the ROP chains with sufficient explanations and screenshots. **For every generated chain, you need to list the gadgets you used in the right order.** You also need to explicitly mention the used tools to complete the lab tasks.

**Note 3:** The only allowed zero bytes in your ROP chains are for referring to memory addresses. In other words, no instructions, numbers, or literal strings in your ROP chains should include `0x00`. You may check your payloads for `0x00` bytes using this command:

```
xxd -i payload | grep 0x00
```

### Task 1: Setting `rax` Value [10%]

Your **task** is to create a ROP chain file called `chain_1` to set the `rax` value to 21 (`0x15`).

**Note:** You need to zero out `rax` first. Also, you should not use `inc rax;` or `inc eax;`

**Hint:** Think of different arithmetic operations that you can perform to achieve `rax=0x15` without having `0x00` bytes in your ROP chains. For example, one of the many ways may be setting `rax` to `0x15FFFFFFFFFFFFFFFF` and shifting bits to the right (assuming you can find the required gadgets).

### Task 2: Open a Shell [45%]

Your **task** is to generate a ROP chain file, called `chain_2`, to open a shell using the `execve` system call with `"/bin/sh"` as an argument. Recall the steps you need to perform to invoke `execve`:

1. `rdi` = address of null-terminated string
2. `rsi` = `NULL`
3. `rdx` = `NULL`
4. `rax` = `0x3b`

### 5. Invoke “int 0x80”

**Hint:** Recall that a ROP chain should not push items to the stack. So, you need to think of another segment to insert a string into *without* changing the stack!

### Task 3: Open a Reverse Shell (ROP + Shellcode) [45%]

Assume that the binary is running at a victim machine. Your **task** is to generate a payload called `chain_3` to start a reverse shell at the victim machine. The reverse shell should execute from a shellcode injected to the stack. However, recall that we enabled the NX bit for the vulnerable binary!

**Hint 1:** You need to think of a *way* to bypass the NX bit using ROP. Then, the following shellcode should start the reverse shell.

**Hint 2:** Recall that the execution model of ROP is different from that of Shellcode. So, after the ROP chain is done, you need to find a way to move the `eip` to the beginning of the shellcode.

**Note:** To create a reverse shell, you may use an existing shellcode from online resources or other tools.

## 3. Submission

You need to submit:

(1) All source code files that you developed, and all ROP chain files that you produced: `chain_1`, `chain_2` and `chain_3`.

(2) A detailed lab report.

The files should be compressed in a single (.zip) archive. The code should compile and run without any errors.