Simon Fraser University
Cybersecurity Lab II

Lab 2

Shellcode Development

The goal of this lab is to implement sample Assembly programs using the two techniques we discussed in the lecture and lab.

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

Prerequisites

To build an Assembly code: `nasm -f elf32 prog.asm`
To produce a binary: `ld prog.o -o prog`
To enable a writable code segment: `ld --omagic prog.o -o prog`

Notes:

- Your code should *not* maintain strings in the data segment.
- You are welcome to modify the code if needed as long as you show how to build and run it.

Task 1 Printing on the Screen [30%]

Your **task** is to implement two Assembly programs to print "Hello, world!" (with CRLF characters) to the standard output. The first program should use the *Relative Addressing* technique, while the second one *pushes string bytes* to the stack.

Startup code for both programs is provided (`print_rel.asm` and `print_stk.asm`), and you need to fill in the missing parts.

Questions

(a) In `print_stk.asm`, explain how the line "`push 0x000a0d21`" works. Show a screenshot from `gdb` to support your explanation.
(b) Also, in the same file, explain how you got the string address. Show a screenshot from `gdb` to support your explanation.

Task 2 Spawning a Shell [70%]

Startup Code (labsh.asm) [10%]

To spawn a new shell, the provided code builds arguments of `execve` to call the "`/bin//sh`" program.
Recall that the `sys_execve` interface is:

```
asmlinkage long sys_execve(const char __user *filename,
                           const char __user *const __user *argv,
                           const char __user *const __user *envp);
```

Currently, the code just spawns a new shell with no arguments to the new process or environment variables. That is, the `envp` array is set to `NULL`, and the `argv` array contains two items: The first one is the address of the command string, and the second one is `NULL`.

Your **task** is to build this program and show a screenshot of a successful run.
A valid screenshot should at least show:

1. The process number of both the calling shell and the spawned shell using "`echo $$`".
2. The passed environment variables to the spawned shell using "`/usr/bin/env`"

Providing Arguments to `/bin/sh` [20%]

Your **task** is to provide additional arguments to the spawned shell. Specifically, in this task, your program needs to run the following command: `/bin/sh -c "ls -la"`
In this new program (call it `labsh_args.asm`), the `argv` array should have the following four elements, all of which need to be constructed on the stack. Modify `labsh.asm` and demonstrate your execution results.

```
argv[3] = 0
argv[2] = "ls -la"
argv[1] = "-c"
argv[0] = "/bin/sh"
```

Providing Env. Variables to `/bin/sh` [20%]

The third parameter for the `execve` system call is a pointer to the environment variable array, and it allows us to pass environment variables to the program. In `labsh.asm`, we pass a null pointer to `execve`, so no environment variable is passed to the program.

In this **task**, you will write a program called `labsh_env.asm`. When this program is executed and you run `/usr/bin/env` inside the shell, it needs to show the following three environment variables:

```
$ /usr/bin/env
aaaa=1234
bbbb=5678
cccc=1234
```

To write such a shellcode, you need to construct an environment variable array on the stack, and store the address of this array to the `edx` register, before calling `execve`. Basically, you first store the actual environment variable strings on the stack. Each string has a format of name=value, and it is terminated by a zero byte. You need to get the addresses of these strings. Then, you construct the environment variable array, also on the stack, and store the addresses of the strings in this array. The array should look like the following (the order of elements does not matter):

```
env[3] = 0 // 0 marks the end of the array
env[2] = address to the "cccc=1234" string
env[1] = address to the "bbbb=5678" string
env[0] = address to the "aaaa=1234" string
```

Using the Relative Addressing Technique [20%]

In this **task**, you need to implement spawning a shell using the Relative Address technique. A startup code is provided for you, and you need to complete the missing parts.

You need to provide a detailed explanation for each line of the code in `labsh_rel.asm`, and explain why this code would successfully execute the `/bin/sh` program, how the `argv` array is constructed, etc. You need to include screenshots while running `gdb` as well.

3. Submission

You are required to submit:
(1) All source code files that you developed.
(2) A detailed lab report.

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