Cybersecurity Lab II

Lab 5

Return-to-libc Attacks

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 C program

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

(c) Running the program

```
$ ./prog payload
```

2. Tasks

The provided program is vulnerable to return-to-libc attacks. Your task is to exploit these vulnerabilities in order to open a shell in two main scenarios. Recall that return-to-libc attacks are useful when the stack region is not executable, and thus, the attacker cannot run shellcode on the stack.
You should set the `BUF_SIZE` to be `12+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.

Important Note: Unlike some of the previous labs, a generic payload generator is **not** provided. However, you need to explain in detail how you generate the payload files. Specifically, your lab report should include all steps you performed to generate the payload files with sufficient explanations and screenshots. You also need to explicitly mention the used tools to complete the lab tasks.

Task 1: Inspect the Program [15%]

Your first **task** is to inspect the provided program and answer the following questions.

(a) What is the piece of code that may result in a return-to-libc vulnerability? Explain.
(b) Suggest a code change to protect against return-to-libc vulnerability (for the provided program).
(c) Can the size of the payload be larger than 300 bytes? Why?

Task 2: Open a shell using `system` function [40%]

Subtask 1. Your **task** is to generate `payload_sys_1`. This payload should call the system function with "`/bin/sh`" as an argument. The program **should not** exit gracefully, instead it needs to jump to `0xdeadbeef`. The "`/bin/sh`" string should be stored as an environment variable.
In addition to screenshots and explanation, you need to show a screenshot of the last line from the output of `dmesg.`

Subtask 2. Similar to Subtask 1, your **task** is to generate `payload_sys_2` that calls the `system` function but exits gracefully. You also need to include representative screenshots from a `gdb` session to show what happens behind the scene. The "`/bin/sh`" string **should** be stored as an environment variable.

Subtask 3. Similar to Subtask 2, your **task** is to generate `payload_sys_3` to open a shell using the `system` function and exit gracefully. The main difference is that the "`/bin/sh`" string **should not** be maintained as an environment variable. *Hint: Think where else you can find such a string!*
In addition, you need to include representative screenshots from a `gdb` session to show what happens behind the scene.
Subtask 4. In this subtask, you need to **enable** ASLR as follows:

```
$ sudo sysctl -w kernel.randomize_va_space=2
```

Your **task** is to exploit the return-to-libc vulnerability for subtask 3 while the ASLR is enabled. First, you need to study how the libc addresses are assigned across different runs of the same program. Then, you need to design a strategy to exploit the return-to-libc vulnerability with ASLR enabled. Finally, you need to realize this strategy and show that you can open a shell when ASLR is enabled.
Your lab report should show the details of these steps.

Note: `gdb` disables ASLR even if the OS enables it. So, if you inspect the output of `vmmap` in a `gdb` session, libc will have the same base address. You need to enable randomization in `gdb` if you plan to use `vmmap`: `(gdb) set disable-randomization off`
Don't forget to disable it again when not needed.

Task 3: Open a shell using `execl` function [45%]

In this task, you need to **disable** ASLR again as follows:

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

Your **task** is to generate `payload_execl` to open a shell using the `execl` function. Unlike the system function, you should use the execl function as follows: `execl("/bin/sh", "/bin/sh", NULL);`
More details are here.
As we discussed in the lecture, the generated payload should use this chain of function calls: `printf` → `execl` → `exit`. The generated payload should also set the expected inputs and return addresses for the function calls.

In your report, you need to mention all the tools you used to build the payload. In addition, you should include representative screenshots from a `gdb` session to show what happens behind the scene. Specifically, you need to show *at least* one screenshot per function call that includes the expected inputs and return address on the stack.
Note: You *may* not be able to open a shell although the call chain is correct. In this case, valid screenshots accompanied with proper explanation are sufficient to show your success in this task.

3. Submission

You need to submit:
(1) All source code files that you developed, and all payload files that you produced.
(2) A detailed lab report.
The files should be compressed in a single (.zip) archive. The code should compile and run without any errors.