Cybersecurity Lab II

Lab 4

Format String Vulnerability

The main objective of this lab is to exploit multiple format string vulnerabilities.

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 execstack -fno-stack-protector prog.c
```

(c) Running the program

```
$ ./prog $(cat payload)
```

2. Tasks

The provided C program has a format string vulnerability. Your task is to exploit multiple format string vulnerabilities that we discussed in the lecture.
You should set the `BUF_SIZE` to be 100+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.

To aid you with the tasks, the code of a generic payload generator is provided (called `build_string.py`). For each of the tasks from 2 to 4, you will need to develop a different Python code based on that provided code. In addition to a detailed lab report, all Python scripts and generated payload files need to be submitted.
As usual, your lab report should include all steps you performed to generate the payload files with sufficient explanations and screenshots.

Task 1: Inspect the Program [20%]

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 format string vulnerability? Explain.
(b) Draw a simplified stack layout when the second `printf` function is invoked from the `details` function. You also need to show where the `buf` array is located (i.e., what stack frame is it located at?). Your drawing doesn't need to show absolute addresses.

Task 2: Print Values from the Stack [20%]

Your **task** is to write a `build_string_print.py` that generates `payload_print`. When build_string_print.py is called with

```
$ python3 build_string_print.py x
```

It should generate a payload that prints x bytes from the stack in a readable format. x is a positive integer.

Task 3: Print a Value from the Heap [20%]

Similar to the previous tasks, you need to write a python script called `build_string_heap.py` that generates a `payload_heap` file.
Your **task** is to read the secret value from the heap. To help you with this task, the address of the variable is given to you. If you succeed, the secret message should be printed on the screen.

Task 4: Modify a Value on the Stack [40%]

This **task** has four subtasks. For each subtask, you need to write a script called build_string_modifyX.py to generate payload_modifyX, where X is the subtask number.

Subtask 1. You need to modify the value of the `target` variable to any arbitrary value.

Subtask 2. You need to modify the value of the `target` variable to be `0x400`.

Subtask 3. You need to modify the value of the `target` variable to be `0x0904bc04`.

Subtask 4. You need to modify the value of the `target` variable to be `0xff990000`.

Challenges. For the third and fourth subtasks, the main challenges are to write a (i) large value, and (ii) zero value to the `target` variable.

Solution Direction. For the first challenge, if you simply reuse the same idea from the second subtask, then you will need to write millions or even billions of bytes on the screen. Obviously, this is not efficient and it will take a lot of time to print these numbers of bytes! To address this challenge, you need to divide the 4-byte value into two 2-byte values, and write each one of them separately to the right memory location. This is because it's more efficient to print up to 2^{16} bytes than to print up to 2^{32} ones. To do so, your payload first needs to include two different addresses, each of which points to a different two-byte component of the original `target` variable. This can be easily achieved as you know the address of the 4-byte value. Second, you need to take advantage of the *length modifier* in format strings while you're using `%n`. Length modifiers, e.g., `%ln, %hn` or others, would point to an argument of a specific length. For example, the `%ln` length modifier instructs the `printf` function to point to a long argument. For more details, read the `printf` man page.

For the second challenge, you need to find a way to write a zero value *after* the `printf` has already written some number of bytes. You need to consider the size of two-byte components as well as number representation.

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.