Format String Exercise

1. Overview

The learning objective of this lab is to gain first-hand experience on format-string vulnerability by putting what you have learned about this vulnerability into action. The goal is to observe that the attacker is capable of doing the following when such vulnerability is present: (1) crash the program, (2) view values from memory, and (3) modify the values of an arbitrary memory place.

Lab environment: Use the pre-built Ubuntu machine that has been provided for this class. **Note:** If you are using a shared folder between your host machine and your VM, make sure to not run the codes from inside the shared folder. Copy them to some other location on Ubuntu and run from there.

Submission: You need to submit a detailed lab report to describe what you have done and what you have observed. Follow the tasks and for each task answer the **Q#** specifically in your report. You may provide an explanation of the observations that are interesting or surprising. **You should always add any code that you modified and screenshots of what you have observed to your report.** You are encouraged to pursue further investigation, beyond what is required by the lab description. Only submit typed reports electronically. No handwritten reports are accepted.

2. Tasks

Task 0: Disable countermeasures

In this section, in order to make the attack easier, we will first disable the address space randomization. In order to disable address randomization temporarily use the following command:

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

Remember that the changes we make are temporary, so if you reboot your system you will need to run this command again. To view the current settings for ASLR on your system you can use this command:

```
$ sysctl -a --pattern randomize
```

If the value is 2, it means full randomization is on, and if it is 0 it means that randomization is disabled.

Task 1: Understand the code and Crash the program

Compile the vulnerable code given to you using the -m32 option:

```
$ gcc -m32 vul.c -o vul
```

Note that as always we assume that the program is only readable/executable by you, and there is no way you can modify the code. However, in order to simplify our attack we assume we have access to the source code to be able to design the attack. Take a look at vul.c code, understand what it is doing, run the code to see what it does.

Q1-A: Find the line of code that has the format string vulnerability and mention it in your report.

Draw a draft of the stack frame. Your stack frame should include the stack frames of myprint() and printf(), but only the

printf() that has the vulnerability. Do not report your stack frame it at this point, but you need to complete it and submit it for the next step. Drawing the stack frame is really important to understand what you have to do in your next steps.

Q1-B: Find out what you should enter as the input string in order to crash the program (get Segmentation fault or Illegal Instruction or a similar error). Show a screenshot of your input and the result.

Task 2: View Memory Values and Complete the Stack Frame

Your first goal here is to enter a string as the input to the program such that you can see what is inside the memory locations of the stack frame. This will help you complete your drawing of the stack frame. As you take advantage of the format string vulnerability in this code to view the content of the memory, you will notice that there are some other values pushed into the stack in between the stack frames of printf() and myprint() functions. The `target=0x11111111` variable will guide you to figure out how many values are pushed in between the two stack frames. This is very important for your next step. The `target` variable holds a hex value and so you will see the exact value of 11111111 printed out.

Q2: Show the screenshot of your result. You need to show what you have entered, the result that is printed out to the screen, and submit the drawing of your completed stack frame.

Task 3: Write an Integer to Memory

In this task, your goal is to write something to the memory. The idea is as the attacker, we can write an integer into ANY memory address using %n. Your goal is to modify the value of the variable `target`. Since we need to know its address, we are "cheating" by printing out the address of this variable in the vulnerable code. Of course, in real attacks, the attackers have to find this address by other means.

You need to create a string as the input to the program and include the address of `target` as part of your input, such that, you modify the value of `target` variable. As you can see the code prints out the value of `target` before and after the line of code that has the vulnerability and this is how you can check if you were able to successfully modify this variable. The exact stack frame that you drew in the previous task, can help you in carefully crafting your input string.

In order to be able to do this, you need to provide the address of `target` as part of your input string. Here are the explanations on how to create an input string that contains an address.

HOW TO: As we discussed in class and in the previous lab, most of our computers are little-endian machines, which means that the least significant byte of an address is stored in the lower address. So in order to store the address 0xAABBCCDD in memory, we need to save it in this order: 0xDD, 0xCC, 0xBB, 0xAA.

If we enter 0x05 as part of the input string, the program will take the ASCII values of '0x05', rather than the hex value 0x05. The challenge is that there is no keyboard character that can be typed that is equivalent to 0x05, so we are not able to type it in as the input.
One way to solve this problem is to write our input string in a file, and then use the file as the input to the program. The printf() function can actually write the address for us. Here is how it works. Suppose you want to enter the following string as your input: 0xffbfcd05.%x.Hello.This.is.me
You can type the following command on your terminal that creates this string and puts it into the file called `inputfile`.

```
$ echo $(printf "\x05\xcd\xbf\xff").%x.Hello.This.is.me > inputfile
```

In order to run the vulnerable program and use the content of the inputfile as the input string use the following command on the terminal:

```
$ ./vul < inputfile
```

Q3: If you manage to create your string correctly the value of the target variable will be changed and you will see that as the value of target after change as a hex value. Show a screenshot of this result. Show how you created your string. Then think about how %n works and what it writes into the memory and explain what this new value represents and where it is coming from.