CYBR 435: Cyber Risk
Spring 2022

Lab Assignment #7: Format String Vulnerability Attack

- Name only: _____
- Release date: Mar 24, 2022 (Thursday), 2:00 pm
- Due date: Mar 31, 2022 (Thursday), 2:00 pm
- Assignment should be **SUBMITTED on Blackboard before Due Date**. Other submission methods will NOT be accepted.
- **LATE Submission will NOT Be Accepted** on Blackboard since the submission link will be closed automatically after due date;
    - Additional submission for missing answer **will NOT Be Accepted**.
- It should be done INDIVIDUALLY; **Show ALL your work and evidence to support your answers**.
    - Answer only without evidence receives half credits.
- Total: 10 pts
- The Lab is adopted from Dr. Wenliang Du at Syracuse University.

Overview

The printf() function in C is used to print out a string according to a format. Its first argument is called format string, which defines how the string should be formatted. Format strings use placeholders marked by the % character for the printf() function to fill in data during the printing. The use of format strings is not only limited to the printf() function; many other functions, such as sprintf(), fprintf(), and scanf(), also use format strings. Some programs allow users to provide the entire or part of the contents in a format string. If such contents are not sanitized, malicious users can use this opportunity to get the program to run arbitrary code. A problem like this is called format string vulnerability.

The objective of this lab is for students to gain the first-hand experience on format string vulnerabilities by putting what they have learned about the vulnerability from class into actions. Students will be given a program with a format string vulnerability; their task is to exploit the vulnerability to achieve the following damage: (1) crash the program. This lab covers the following topics:

- Format string vulnerability, and code injection
- Stack layout

Lab Environment

This lab has been tested on the SEED Ubuntu 20.04 VM. You can download a pre-built image from the SEED website (https://seedsecuritylabs.org/Labs_20.04/Software/Format_String/), and run the SEED VM on your own computer. However, most of the SEED labs can be conducted on the cloud, and you can follow our instruction to create a SEED VM on the cloud.

Environment Setup

Turning of Countermeasure

Modern operating systems uses address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of the format-string attack. To simplify the tasks in this lab, we turn off the address randomization using the following command:

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

The Vulnerable Program

The vulnerable program used in this lab is called format.c, which can be found in the server-code folder (https://seedsecuritylabs.org/Labs_20.04/Software/Format_String/). This program has a format-string vulnerability, and your job is to exploit this vulnerability. The code listed below has the non-essential information removed, so it is different from what you get from the lab setup file.

Listing 1: The vulnerable program format.c (with non-essential information removed)

```
unsigned int  target = 0x11223344;
char *secret = "A secret message\n";

void myprintf(char *msg)
{
    // This line has a format-string vulnerability
    printf(msg);
}

int main(int argc, char **argv)
{
    char buf[1500];
    int length = fread(buf, sizeof(char), 1500, stdin);
    printf("Input size: %d\n", length);

    myprintf(buf);

    return 1;
}
```

The above program reads data from the standard input, and then passes the data to myprintf(), which calls printf() to print out the data. The way how the input data is fed into the printf() function is unsafe, and it leads to a format-string vulnerability. We will exploit this vulnerability.

The program will run on a server with the root privilege, and its standard input will be redirected to a TCP connection between the server and a remote user. Therefore, the program actually gets its data from a remote user. If users can exploit this vulnerability, they can cause damages.

Compilation

We will compile the format program into both 32-bit and 64-bit binaries. Our pre-built Ubuntu 20.04 VM is a 64-bit VM, but it still supports 32-bit binaries. All we need to do is to use the -m32 option in the gcc command. For 32-bit compilation, we also use -static to generate a statically-linked binary, which is self contained and not depending on any dynamic library, because the 32-bit dynamic libraries are not installed in our containers.

The compilation commands are already provided in Makefile. To compile the code, you need to type make to execute those commands. After the compilation, we need to copy the binary into the fmt-containers folder, so they can be used by the containers. The following commands conduct compilation and installation.

```
$ make
$ make install
```

During the compilation, you will see a warning message. This warning is generated by a countermeasure implemented by the gcc compiler against format string vulnerabilities. We can ignore this warning for now.

```
format.c: In function 'myprintf':
format.c:33:5: warning: format not a string literal and no format arguments
                        [-Wformat-security]
   33 |     printf(msg);
      |     ^~~~~~
```

It should be noted that the program needs to be compiled using the "-z execstack" option, which allows the stack to be executable. Our ultimate goal is to inject code into the server program's stack, and then trigger the code. Non-executable stack is a countermeasure against stack-based code injection attacks, but it can be defeated using the return-to-libc technique, which is covered by another SEED labs. In this lab, for simplicity, we disable this defeat-able countermeasure.

The Server Program

In the server-code folder, you can find a program called server.c. This is the main entry point of the server. It listens to port 9090. When it receives a TCP connection, it invokes the format program, and sets the TCP connection as the standard input of the format program. This way, when format reads data from stdin, it actually reads from the TCP connection, i.e. the data are provided by the user on the TCP client side. It is not necessary for students to read the source code of server.c.

We have added a little bit of randomness in the server program, so different students are likely to see different values for the memory addresses and frame pointer. The values only change when the container restarts, so as long as you keep the container running, you will see the same numbers (the numbers seen by different students are still different). This randomness is different from the address-randomization countermeasure. Its sole purpose is to make students' work a little bit different.

Container Setup and Commands

Please download the Labsetup.zip file to your VM from the lab's website, unzip it, enter the Labsetup folder, and use the docker-compose.yml file to set up the lab environment. Detailed explanation of the content in this file and all the involved Dockerfile can be found from the user manual, which is linked to the website of this lab. If this is the first time you set up a SEED lab environment using containers, it is very important that you read the user manual.

In the following, we list some of the commonly used commands related to Docker and Compose. Since we are going to use these commands very frequently, we have created aliases for them in the .bashrc file (in our provided SEEDUbuntu 20.04 VM).

```
$ docker-compose build  # Build the container image
$ docker-compose up     # Start the container
$ docker-compose down   # Shut down the container

// Aliases for the Compose commands above
$ dcbuild       # Alias for: docker-compose build
$ dcup          # Alias for: docker-compose up
$ dcdown        # Alias for: docker-compose down
```

All the containers will be running in the background. To run commands on a container, we often need to get a shell on that container. We first need to use the "docker ps" command to find out the ID of the container, and then use "docker exec" to start a shell on that container. We have created aliases for them in the .bashrc file.

```
$ dockps        // Alias for: docker ps --format "{{.ID}}  {{.Names}}"
$ docksh <id>   // Alias for: docker exec -it <id> /bin/bash

// The following example shows how to get a shell inside hostC
$ dockps
b1004832e275  hostA-10.9.0.5
0af4ea7a3e2e  hostB-10.9.0.6
9652715c8e0a  hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#

// Note: If a docker command requires a container ID, you do not need to
//       type the entire ID string. Typing the first few characters will
//       be sufficient, as long as they are unique among all the containers.
```

If you encounter problems when setting up the lab environment, please read the "Common Problems" section of the manual for potential solutions.

Task 1: Crashing the Program

When we start the containers using the included docker-compose.yml file, two containers will be started, each running a vulnerable server. For this task, we will use the server running on 10.9.0.5, which runs a 32-bit program with a format-string vulnerability. Let's first send a benign message to this server. We will see the following messages printed out by the target container (the actual messages you see may be different).

```
$ echo hello | nc 10.9.0.5 9090
Press Ctrl+C

// Printouts on the container's console
server-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | Input buffer (address):        0xffffd2d0
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Input size: 6
server-10.9.0.5 | Frame Pointer inside myprintf() = 0xffffd1f8
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 | hello
server-10.9.0.5 | (^_^)(^_^) Returned properly (^_^)(^_^)
server-10.9.0.5 | The target variable's value (after):  0x11223344
```

The server will accept up to 1500 bytes of the data from you. Your main job in this lab is to construct different payloads to exploit the format-string vulnerability in the server, so you can achieve the goal specified in each task. If you save your payload in a file, you can send the payload to the server using the following command.

```
$ cat <file> | nc 10.9.0.5 9090
Press Ctrl+C  if it does not exit.
```

Your task is to provide an input to the server, such that when the server program tries to print out the user input in the myprintf() function, it will crash. You can tell whether the format program has crashed or not by looking at the container's printout. If myprintf() returns, it will print out "Returned properly" and a few smiley faces. If you don't see them, the format program has probably crashed. However, the

server program will not crash; the crashed format program runs in a child process spawned by the server program.

Since most of the format strings constructed in this lab can be quite long, it is better to use a program to do that. Inside the attack-code directory, we have prepared a sample code called build string.py for people who might not be familiar with Python. It shows how to put various types of data into a string.

Questions for the Lab

Your submission should include the following:

1. A WORD file containing
    - Carefully read the lab instructions and finish all steps (a sequence of screenshots) in Environment Setup. Take the screenshot of each major step. [4 pts]
    - Carefully read the lab instructions and finish all steps (a sequence of screenshots) in Task 1. Take the screenshot of results and explain how you are able to achieve the goal of format string vulnerability attack. [6 pts]

Happy Hacking!