Format-string Attack Lab

Ethical Hacking 2021/22, University of Padua

Eleonora Losiouk, Alessandro Brighente, Denis Donadel, Gabriele Orazi

---

 This lab was developed with the help of Hao Zhang and Kuber Kohli, graduate students in the Department of Electrical Engineering and Computer Science 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, (2) read the internal memory of the program, (3) modify the internal memory of the program. This lab covers the format string vulnerability.

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, 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.

---

1 Environment Setup

1.1 Turning off countermeasures

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
```

1.2 The Vulnerable Program

The vulnerable program used in this lab is called format.c, which can be found in the server-code folder. 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.

```
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);
| ^~~~~~
```

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.

1.3 Container Setup and Commands

Please, download the `Labsetup.zip` file to your VM from Moodle, 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. 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.
```

2 Lab Tasks

2.1 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.
```

Task. 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.

2.2 Task 2: Printing Out the Server Program's Memory

The objective of this task is to get the server to print out some data from its memory (we will continue to use `10.9.0.5`). The data will be printed out on the server side, so the attacker cannot see it. Therefore, this is not a meaningful attack, but the technique used in this task will be essential for the subsequent tasks.

Task 2.A: Stack Data. The goal is to print out the data on the stack. How many `%x` format specifiers do you need so you can get the server program to print out the first four bytes of your input? You can put some unique numbers (4 bytes) there, so when they are printed out, you can immediately tell. This number will be essential for most of the subsequent tasks, so make sure you get it right.

Task 2.B: Heap Data. There is a secret message (a string) stored in the heap area, and you can find the address of this string from the server printout. Your job is to print out this secret message. To achieve this goal, you need to place the address (in the binary form) of the secret message in the format string. Most computers are small-endian machines, so to store an address `0xAABBCCDD` (four bytes on a 32-bit machine) in memory, the least significant byte `0xDD` is stored in the lower address, while the most significant byte `0xAA` is stored in the higher address. Therefore, when we store the address in a buffer, we need to save it using this order: `0xDD`, `0xCC`, `0xBB`, and then `0xAA`. In Python, you can do the following:

```
number = 0xAABBCCDD
content[0:4] = (number).to_bytes(4,byteorder='little')
```

2.3 Task 3: Modifying the Server Program's Memory

The objective of this task is to modify the value of the target variable that is defined in the server program (we will continue to use `10.9.0.5`). The original value of `target` is `0x11223344`. Assume that this variable holds an important value, which can affect the control flow of the program. If remote attackers can change its value, they can change the behavior of this program. We have two sub-tasks.

Task 3.A: Change the value to a different value. In this sub-task, we need to change the content of the `target` variable to something else. Your task is considered as a success if you can change it to a different value, regardless of what value it may be. The address of the `target` variable can be found from the server printout.

Task 3.B: Change the value to 0x5000. In this sub-task, we need to change the content of the `target` variable to a specific value `0x5000`. Your task is considered as a success only if the variable's value becomes `0x5000`.