Acknowledgement

The following lab is a reproduction of SEED Labs material. The original Copyright notice is as follows:

Format String Vulnerability

Lab 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 place-holders 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:

1) crash the program,
2) read the internal memory of the program,
3) modify the internal memory of the program.

Disable ASLR

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
```

Task 1: The Vulnerable Program

You are given a vulnerable program that has a format string vulnerability. 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.

You can download the server's code from SEED labs by running the following command in the SEED VM terminal:

```
wget https://seedsecuritylabs.org/Labs_16.04/Software/Format_String_Server/files/server.c
```

The `server.c` code, for your reference and review, is as follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>

#define PORT 9090

/* Changing this size will change the layout of the stack.
 * We have added 2 dummy arrays: in main() and myprintf().
 * Instructors can change this value each year, so students
 * won't be able to use the solutions from the past.
 * Suggested value: between 0 and 300  */
#ifndef DUMMY_SIZE
#define DUMMY_SIZE 100
#endif

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

void myprintf(char *msg)
{
    uintptr_t framep;
    // Copy the ebp value into framep, and print it out
    asm("movl %%ebp, %0" : "=r"(framep));
    printf("The ebp value inside myprintf() is: 0x%.8x\n", framep);

    /* Change the size of the dummy array to randomize the parameters
       for this lab. Need to use the array at least once */
    char dummy[DUMMY_SIZE];  memset(dummy, 0, DUMMY_SIZE);

    // This line has a format-string vulnerability
    printf(msg);
    printf("The value of the 'target' variable (after): 0x%.8x\n", target);
}

/* This function provides some helpful information. It is meant to
 *   simplify the lab tasks. In practice, attackers need to figure
 *   out the information by themselves. */
void helper()
```

```
{
    printf("The address of the secret: 0x%.8x\n", (unsigned) secret);
    printf("The address of the 'target' variable: 0x%.8x\n",
            (unsigned) &target);
    printf("The value of the 'target' variable (before): 0x%.8x\n", target);
}

void main()
{
    struct sockaddr_in server;
    struct sockaddr_in client;
    int clientLen;
    char buf[1500];

    /* Change the size of the dummy array to randomize the parameters
       for this lab. Need to use the array at least once */
    char dummy[DUMMY_SIZE];  memset(dummy, 0, DUMMY_SIZE);

    printf("The address of the input array: 0x%.8x\n", (unsigned) buf);

    helper();

    int sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    memset((char *) &server, 0, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = htonl(INADDR_ANY);
    server.sin_port = htons(PORT);

    if (bind(sock, (struct sockaddr *) &server, sizeof(server)) < 0)
        perror("ERROR on binding");

    while (1) {
        bzero(buf, 1500);
        recvfrom(sock, buf, 1500-1, 0,
                 (struct sockaddr *) &client, &clientLen);
        myprintf(buf);
    }
    close(sock);
}
```

Compilation

Compile the above program. 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.

```
gcc -DDUMMY_SIZE=0 -z execstack server.c  -o server
```

Running the Server

We will run the server on the SEED VM (32-bit Ubuntu 16.04) in one terminal window and the client in another. To emulate a program running as a system `daemon` with the root privilege we will run the program with `sudo` command. From the other terminal playing the role of a client machine 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.

In the first terminal (the role of the server) run:

```
sudo ./server
```

In the second terminal (the client role) run:

```
echo hello | nc -u 127.0.0.1 9090
```

You can also send the content of file to the server as follows (in client terminal):

```
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. In the rest of this lab, we are going to exploit this vulnerability.

Task 2: Understanding the Stack Layout

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. We intentionally print out some information in the server code to help simplify the investigation. Based on the investigation, students should answer the following questions:

Question 1) What are the memory addresses at the locations marked by **1**, **2**, and **3**?

Question 2) What is the distance between the locations marked by **1** and **3**?

Task 3: Crash the 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.

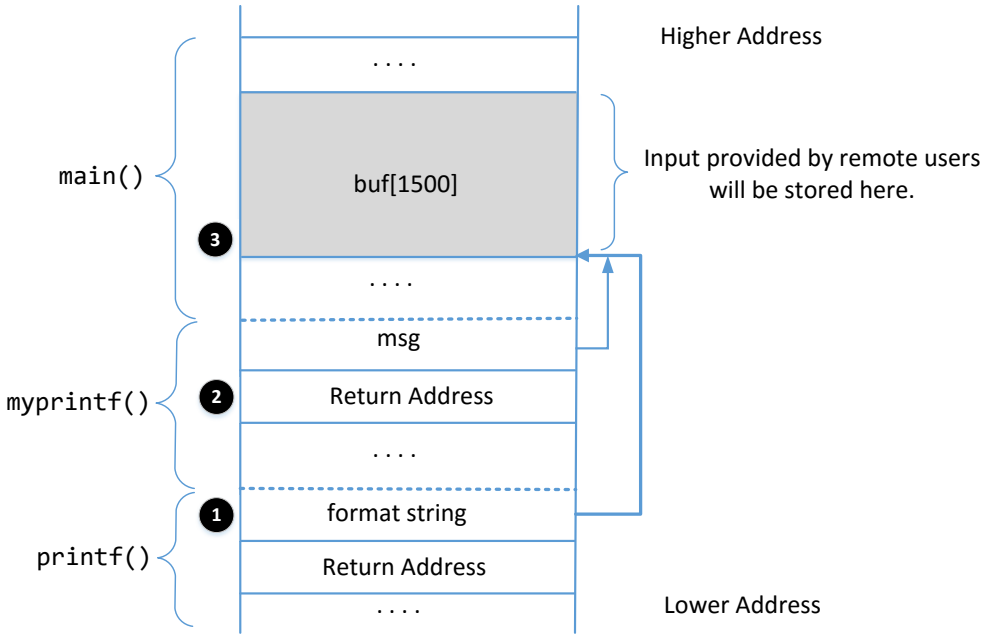

Figure 1: The stack layout when `printf()` is invoked from inside of the `myprintf()` function.

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.

A) Print 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`?

B) Print 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 to do that.

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

or we can save the data in a file and then pass the file to `nc`:

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

It should be noted that 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`.

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 behaviour of this program.

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.

B) Change the value to `0x500`

In this sub-task, we 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`.