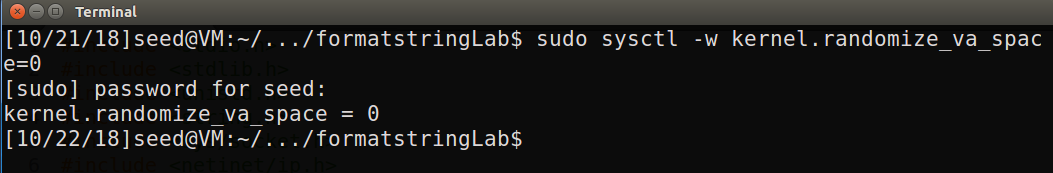
**Format String Vulnerability Lab**

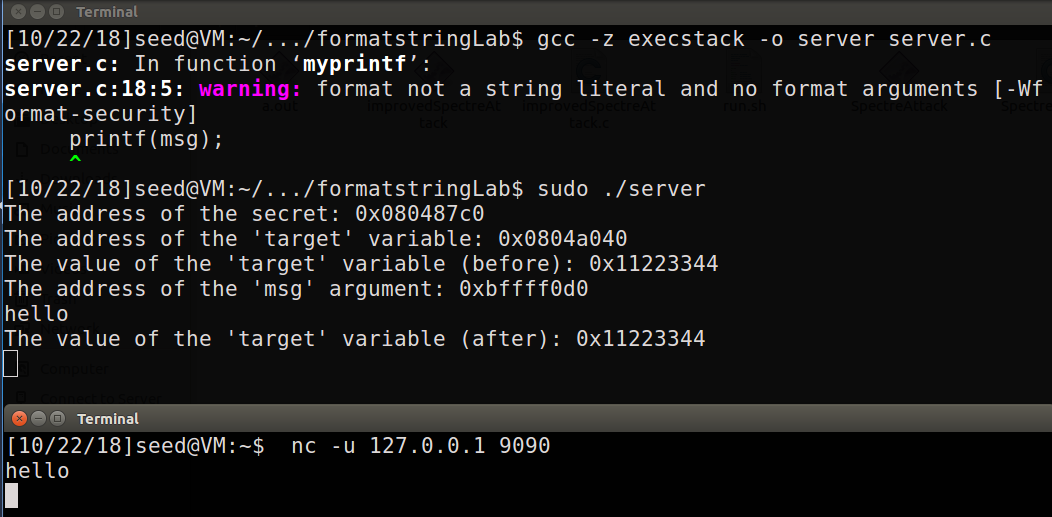
**Karan Amrutesh**

**Turning off the address randomization and compiling the program which gives a warning:**



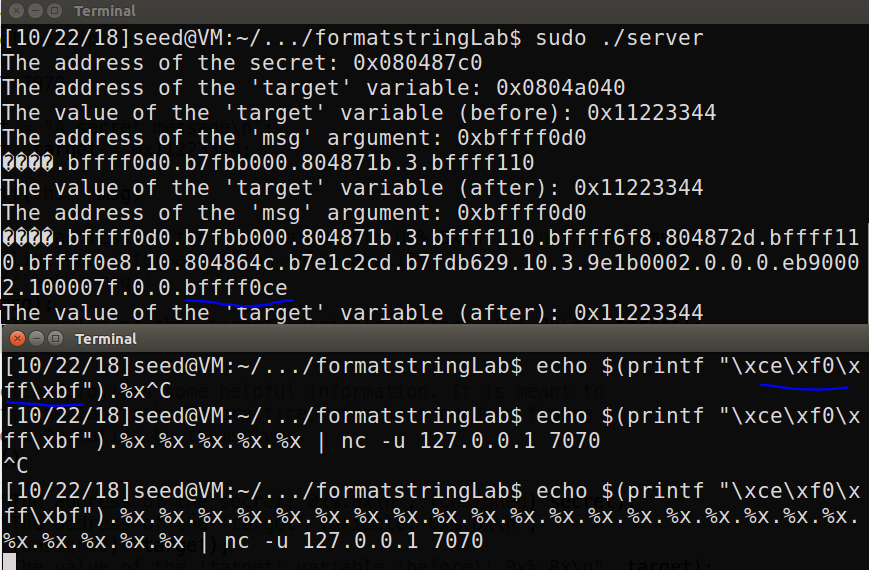
**Task 1: The Vulnerable Program**

* Sending a hello msg will give the value in the server:

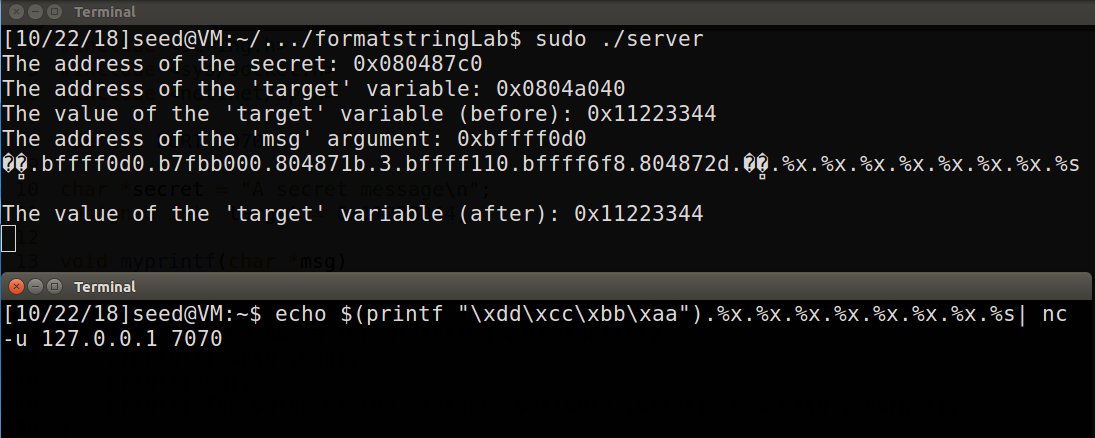


**Task 2: Understanding the Layout of the Stack**

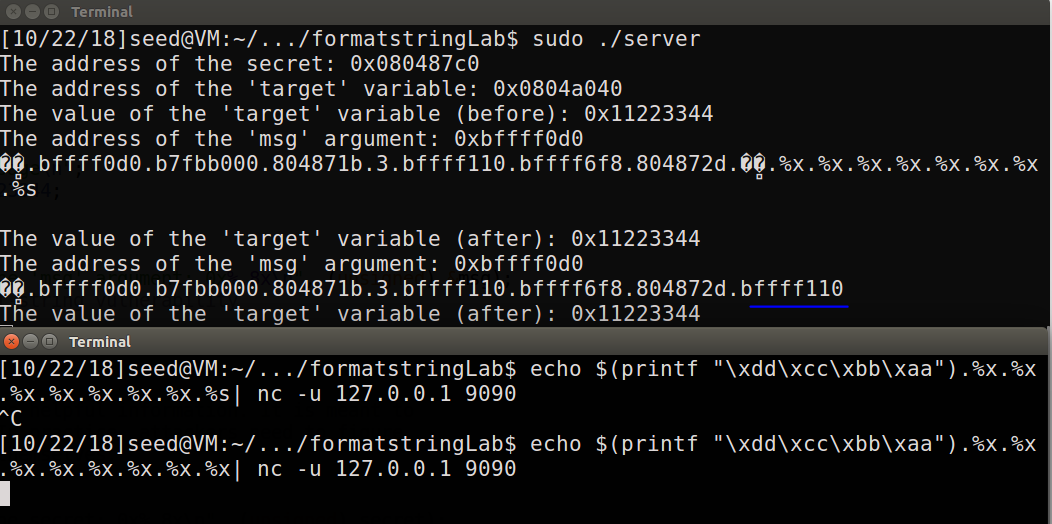
Starting with a few %x and continuing to get the user input, we can see that 24 %x will make the ap point to the user input.



We then put %x followed by %s to get the data in that address. We can see that after 7 %x and 1 %s we get our input.



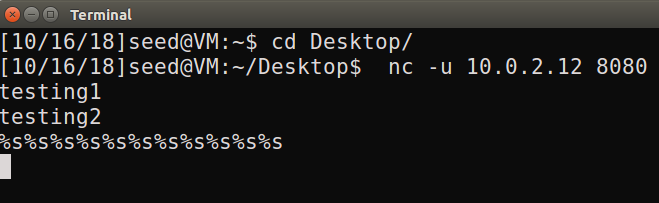
Replacing the %s with %x will give the address which corresponds to ➌

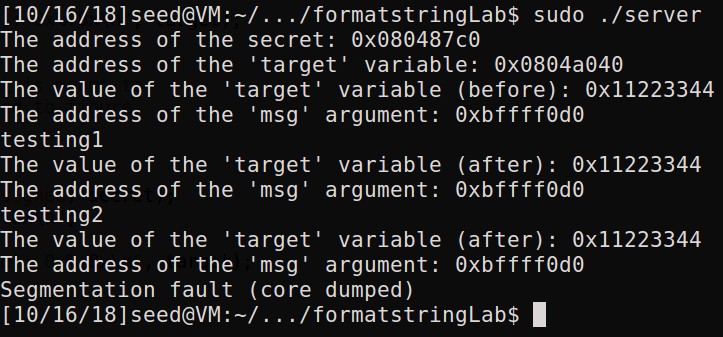


1. It takes 24 %x to print get the first 4 bytes. So the difference between ➊ and ➌ is 23 %x which is 23\*4 = 92 bytes. Therefore ➌ - 92 i.e. *bffff110 – 5c* = ***0xbffff0b4***. This is ➊
2. This can be calculated using the address of the msg argument – 4. Which is bffff0d0 – 4 = ***bffff0cc***
3. From the above, ➌ is ***bffff110***

**Task 3: Crash the Program**

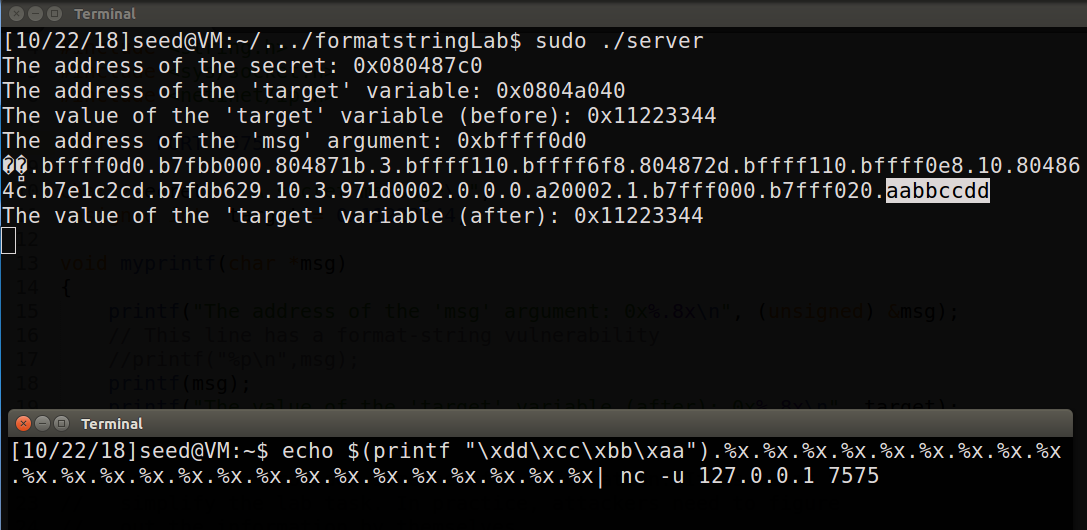
* Using %s causes a read from the memory address accessed by the app pointer. So when a memory access fails due to access of a protected memory, or a non-existing memory, the program crashes.





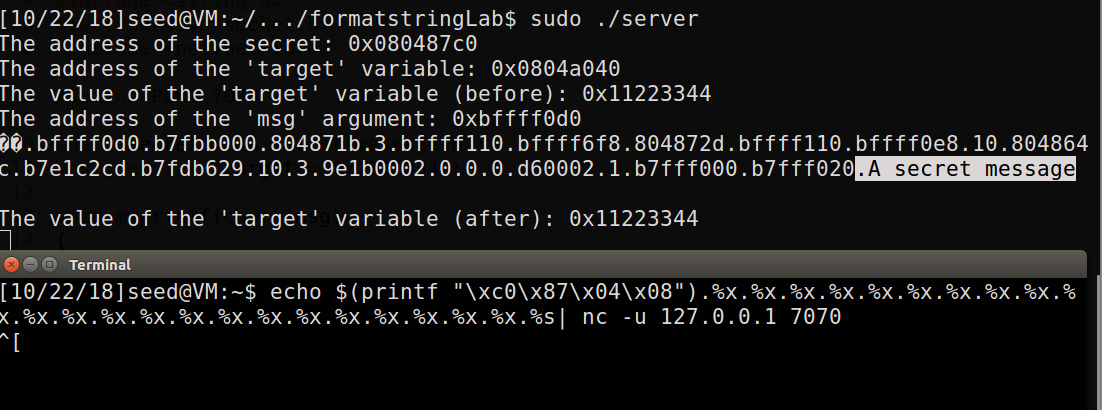
**Task 4: Print Out the Server Program’s Memory**

**Task 4.A: Stack Data**



From the calculations in task 2, we know the difference between ➊ and ➌ is 92 bytes, which is 23 %x to point to the address and the 24th %x will print our address.

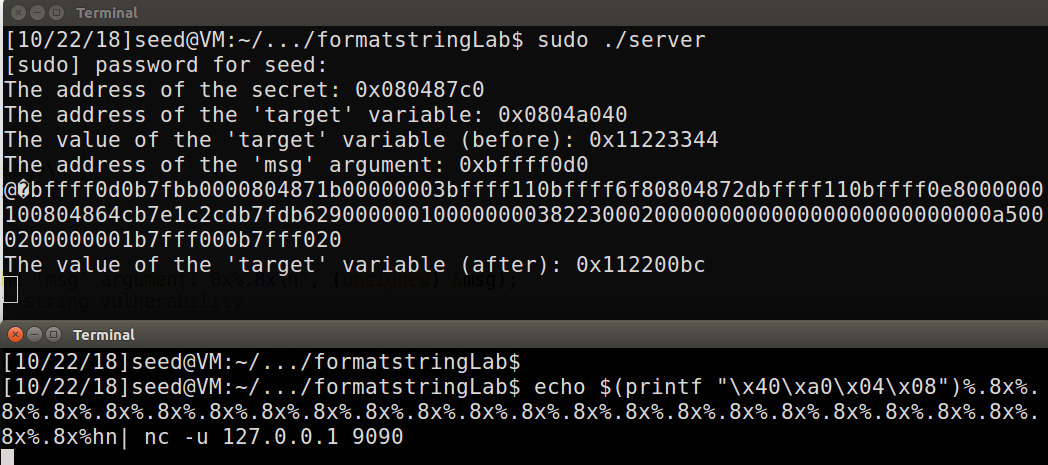
**Task 4.B: Heap Data**



* The secret variable is stored in the heap and we know that address. So we put that address in the beginning so that it goes to the user input. Then we use 23 %x followed by %s to read the data from that address.
* We can see that the contents of the secret var is printed out.

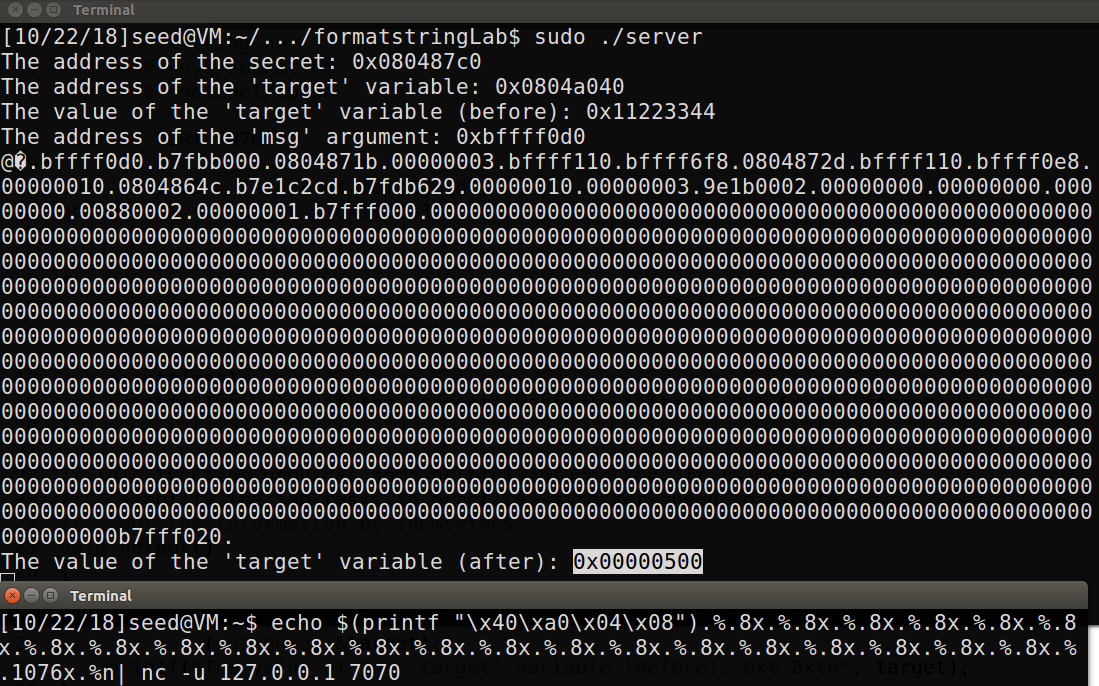
**Task 5: Change the Server Program’s Memory**

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



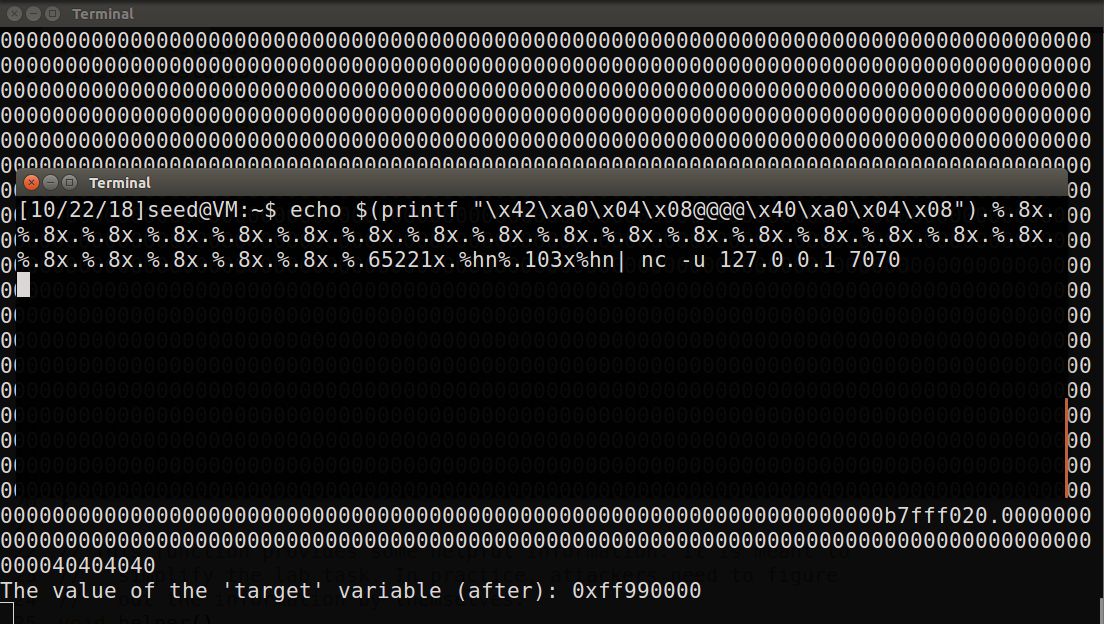
* Changing the value stored in the buf can be changed by using %n which writes the number of characters already printed by the printf statement.
* Here, the value of the target initially was 11223344. Before it writes to the address of target, it prints out 4 bytes of address, then 23\*8 (for %x).
* Therefore 4 + (23\*8) = BC. %hn writes only 2 bytes. So the first two bytes of the target variable is changed to BC. Hence the target variable is 0x112200bc

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



* Similar to the calculation in task 5.a:   
  0x500 = 1280 in decimal. Hence 4 + 24(dots) + 22\*8(%x) + w = 1280.   
  w = 1076 which is value for 23rd %x.
* %n will change all the bytes of the target variable. So we can see that it is changed.

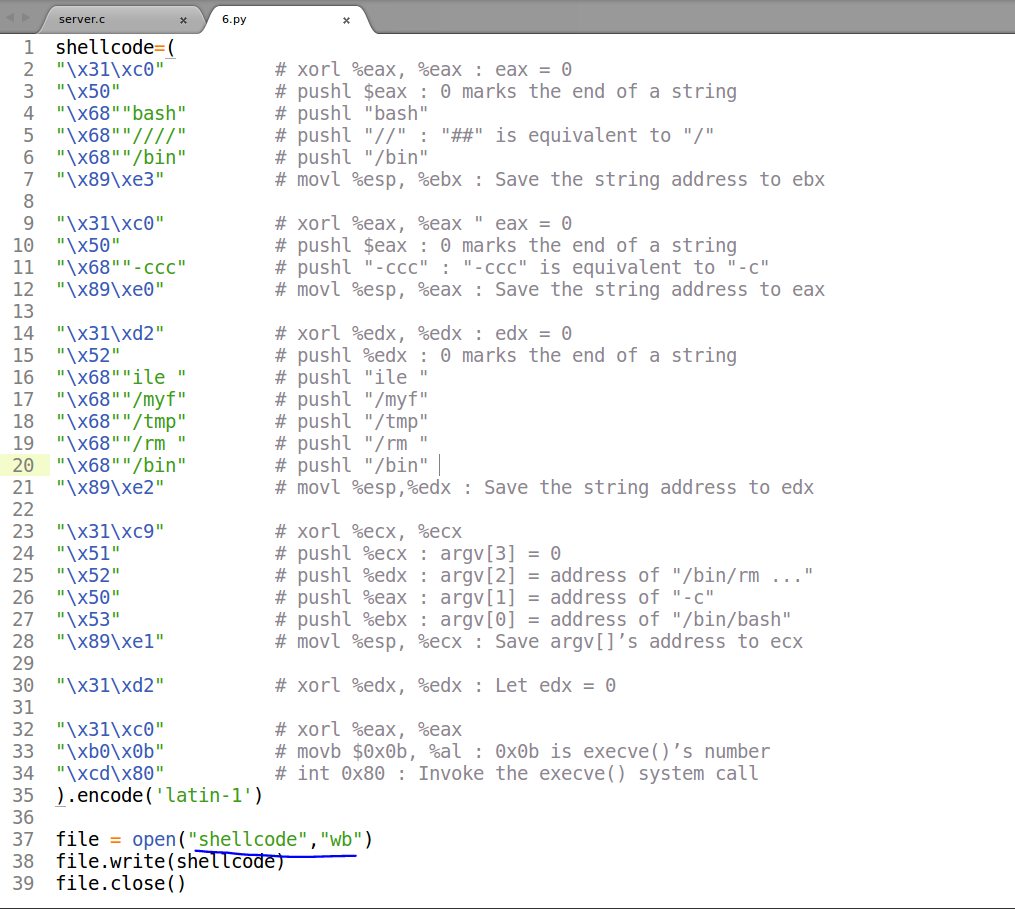
Task 5.C: Change the value to 0xFF990000

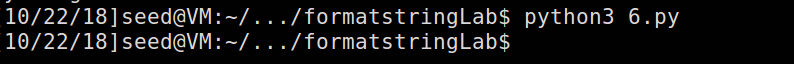


* Here we change 2 bytes at time using %hn. Since it is little Endian format, we give the address where the higher bytes are stored, i.e. where ff99 is stored.
* Then we separate this address with @@@@ and give the address of where the higher two bytes are stored, i.e. where 0000 is sotred.
* 0xff99 = 65433 in decimal. Hence 12 + 24(dots) + 22\*8(%x) + w = 65433.   
  w = 65221 which is value for 23rd %x. %hn that follows this will write this value (ff99) to the first address given, that is 0x08044042
* To get 0000, we need to reset the counter of %n. We can do this by making the counter to 65536. Hence 65433 + t = 65536.   
  t = 103 which is value for 23rd %x. %hn that follows this will write this value (0000) to the second address given, that is 0x08044040

**Task 6: Inject Malicious Code into the Server Program**

* The following python code writes the shell code to remove the file into file named ‘*shellcode’* in binary format.

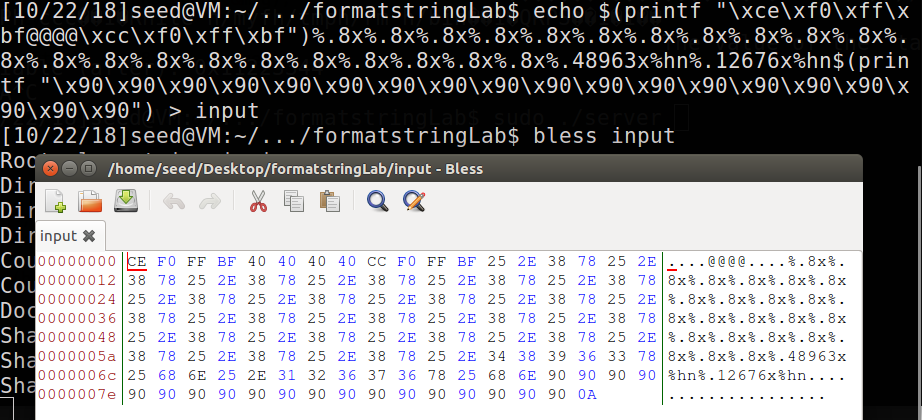




* Then, we calculate the address of the malicious code we should put in the format string using the offset of *NOP.* We use bless to see the input file. So,

Address where buf starts(➌-bffff110) + offset of a NOP (0x7b) = malicious code address

Therefore we get the Malicious code address as ***bffff18b***



* The format string is constructed as follows:

First, we split the return address (➋) that we obtained in Task 2 into 2 addresses in order to write 2 bytes at a time.

Split as: bfffff0ce and bfffff0cc

* Next, we need a number to write into bffff0ce using %n, which is calculated as:

12 (8 for the addresses, 4 for @@@@) + 8\*22 (%x) + w = 49151 (bfff)

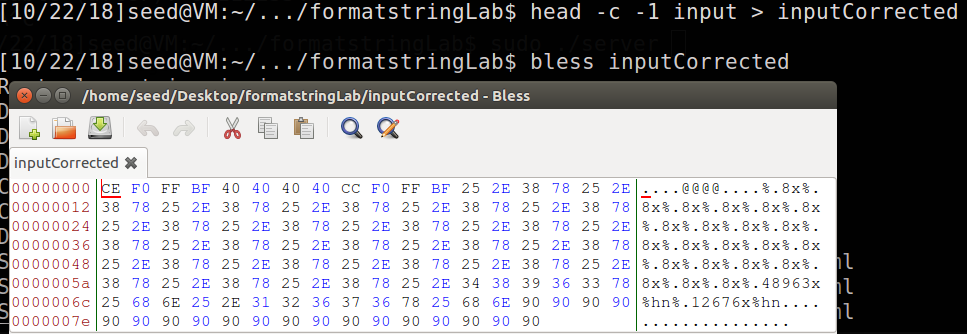
We get w = 48963

* Then we calculate for writing into bffff0cc:

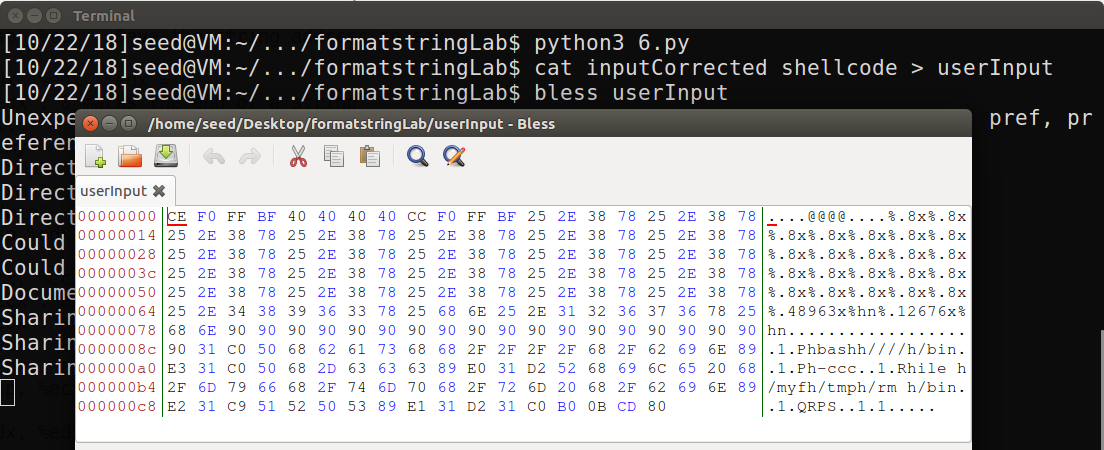
49151 + t = f18b

t = 12676

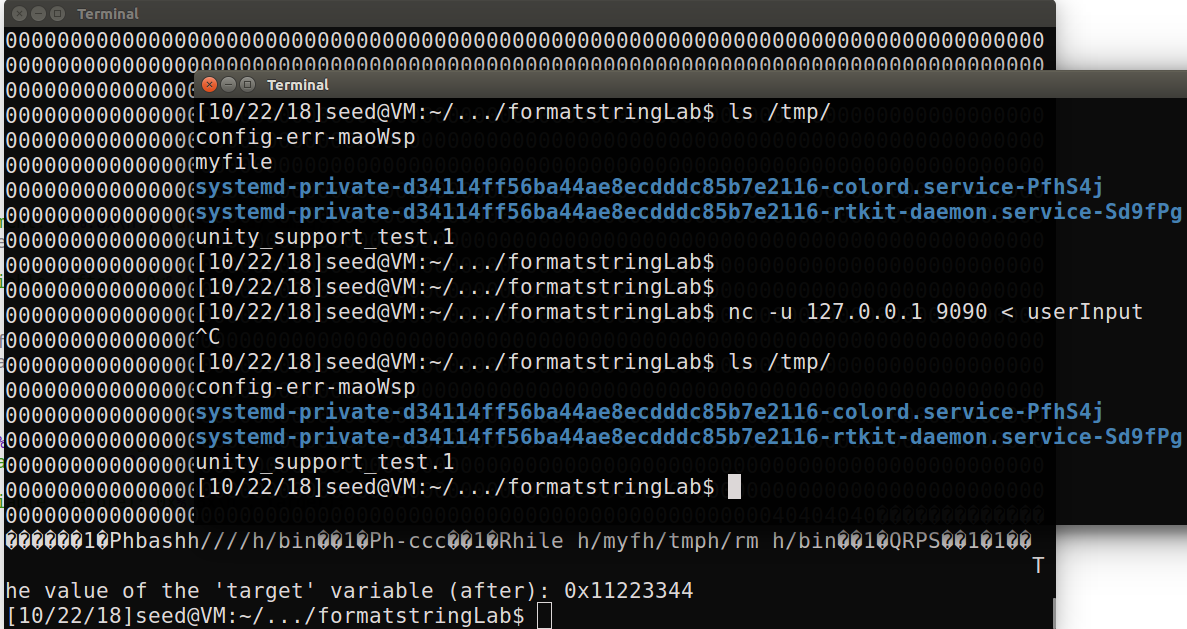
* We write this format string into an ‘*input’* file. On seeing the contents of this file using bless, we see that there is an additional ‘XA’ character at the end. We remove that using the following command.



* This is stored into a file named ‘*inputCorrected’*
* Next step is to append the shellcode file to the end of this format string and store it in ‘*userInput’:*

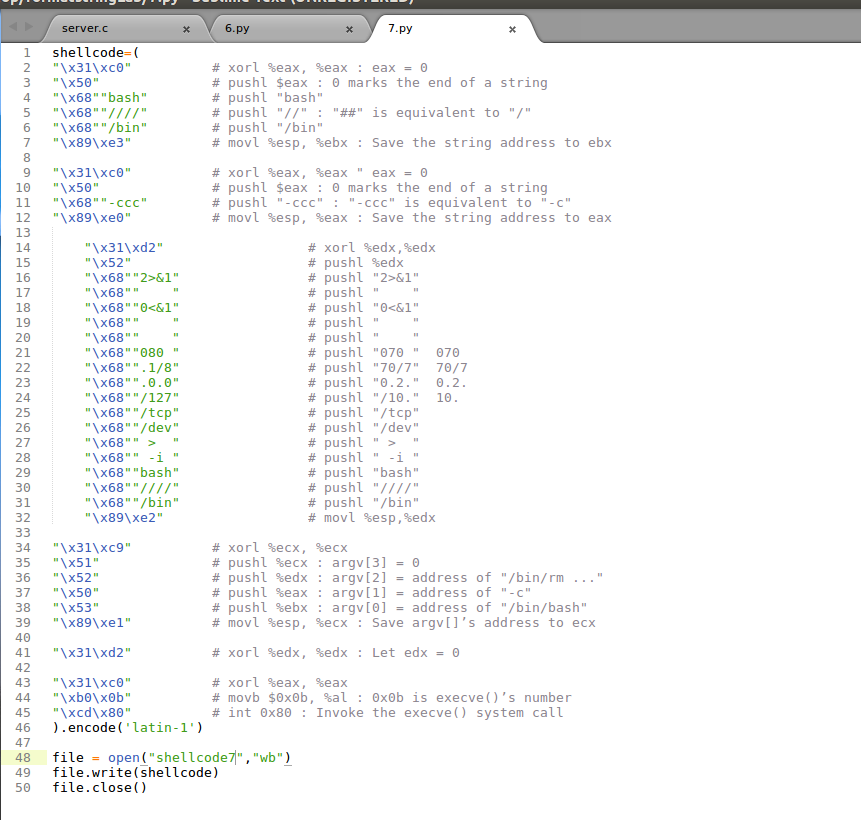


* Next we sent the format string to the server:

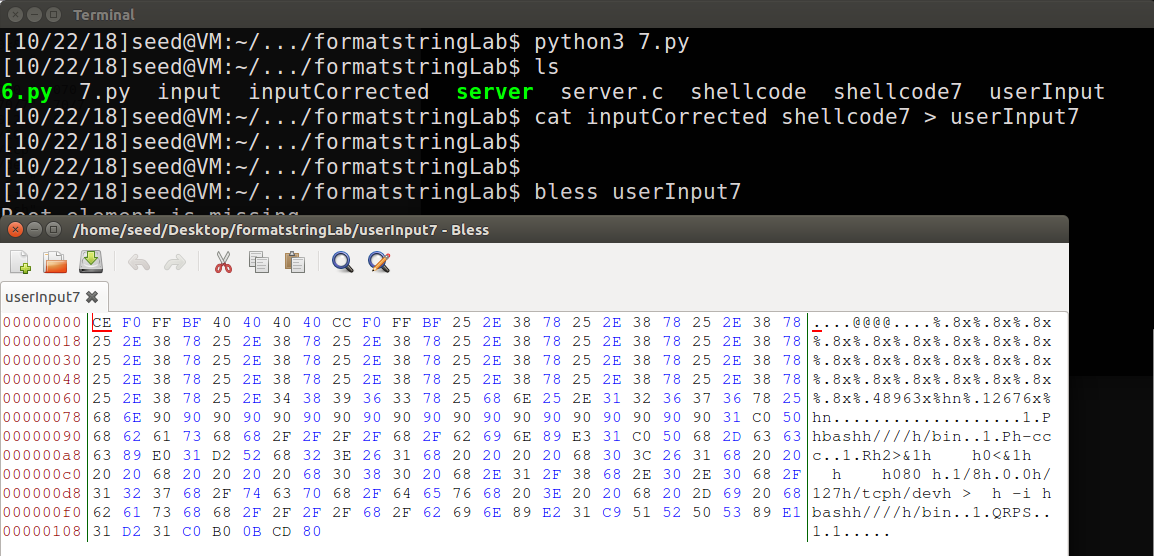


* We can see that the /tmp/myfile has been removed. This implies that the shellcode placed at the malicious code address was executed when the printf method returned (using the return address that we replaced).

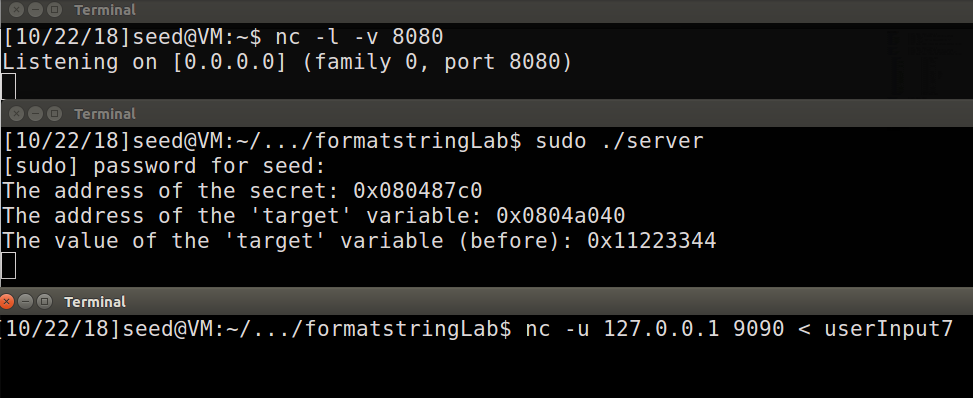
**Task 7: Getting a reverse shell**

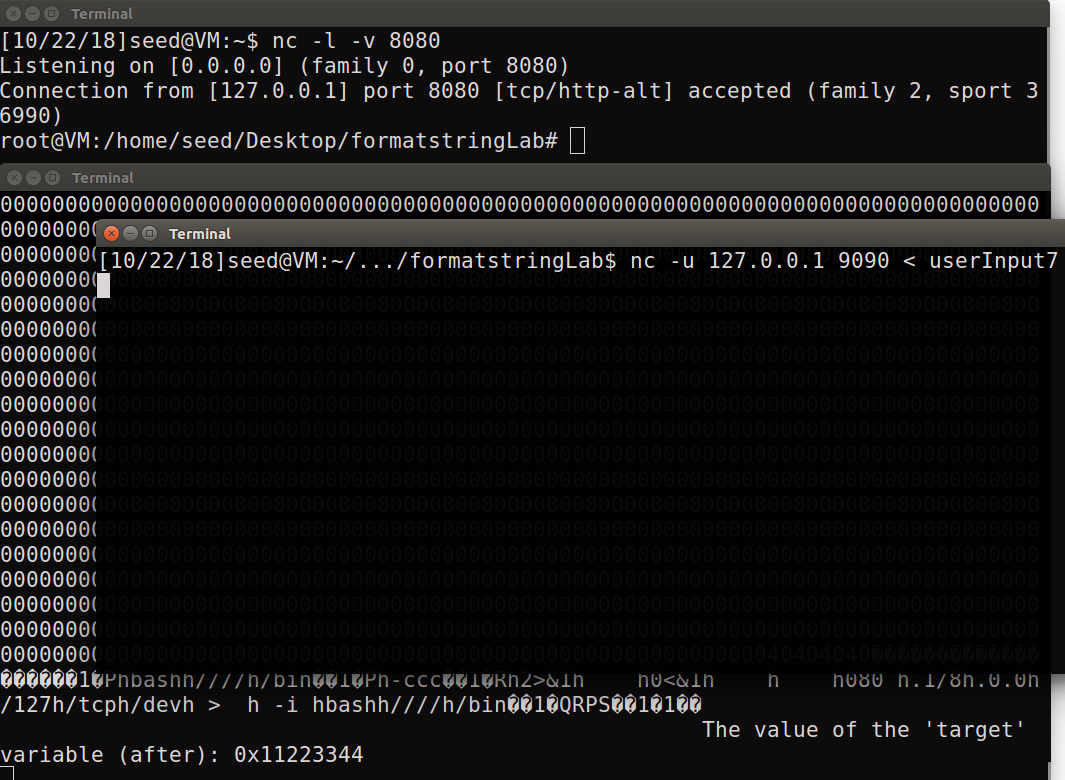


* We have made the changes to the previous shellcode to get a reverse shell. We run the python program.
* Here, we have used the same input file from the previous tasked and appended the new shellcode to create the new userInput file.



* We use 3 terminals to carry out this task: 1 for server, 1 for sending format string from client and one more for listening for reverse shell



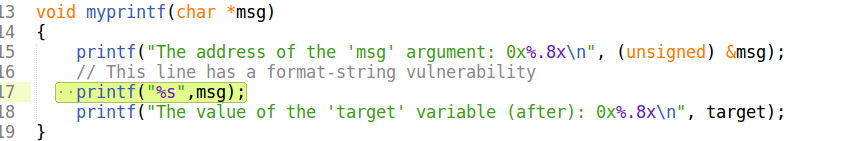


* We can see that we get the reverse shell implying that the malicious code has been executed. Since the server is running in root privilege, we have got a root shell.

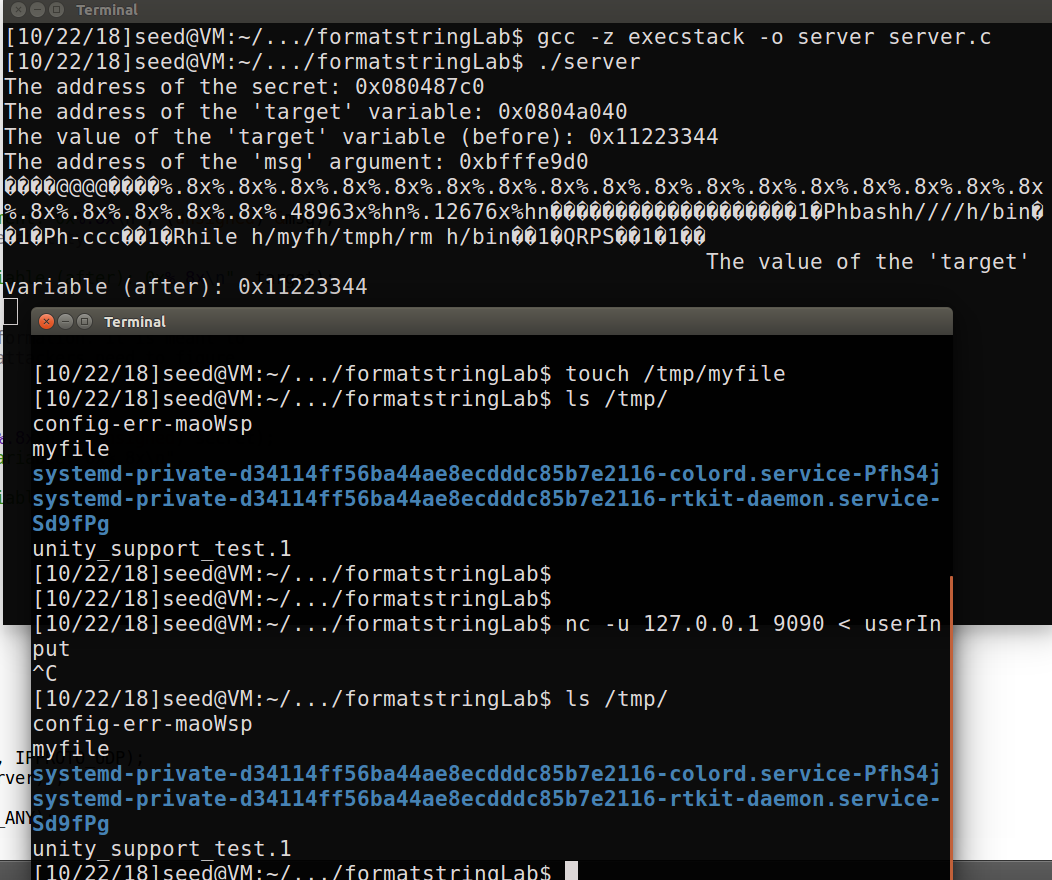
**Task 8: Fixing the Problem:**

* The warning states by passing the string literal directly to printf function instead of a format specifier and then a string literal, it cannot check for safety.

* To correct it, we need to provide a format string for the string literal being passed:



* Doing this, makes the warning go away.



* Repeating the task 6 to remove /tmp/myfile, doesn’t work as the server just prints out the msg received by the client. The entire input passed by the client is treated as only one input and thus its prints it out.