**Buffer Overflow**

姓名：钱煜 学号：57118105

1. 实验目的

了解缓冲区溢出的特性。

1. 实验内容

预设：关闭保护对策。

1.运行shellcode；

2.利用漏洞；

3.击败dash 的对策；

4.解决地址随机化；

5.

6.

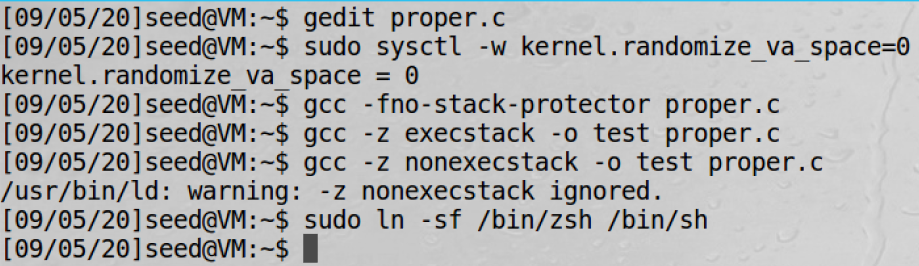
1. 实验环境

VMware workstation 15.0x，Seed Ubuntu 16.04

1. 实验

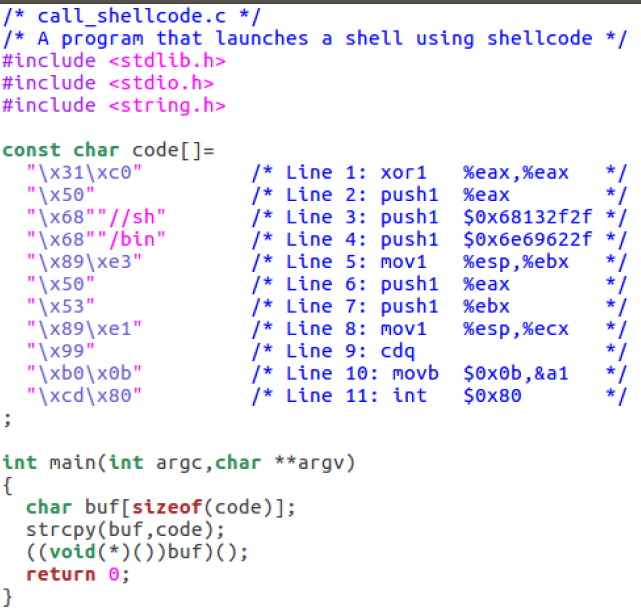
**预设.关闭保护措施**

按照要求尝试：

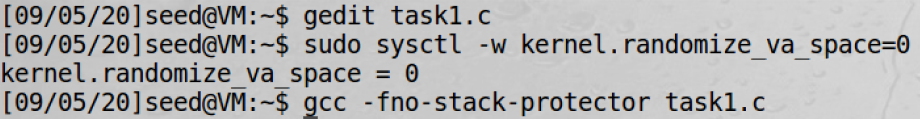


**Task 1. 运行shellcode**

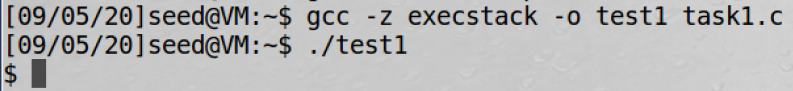
按照要求编写task1.c：



配置解除保护措施：

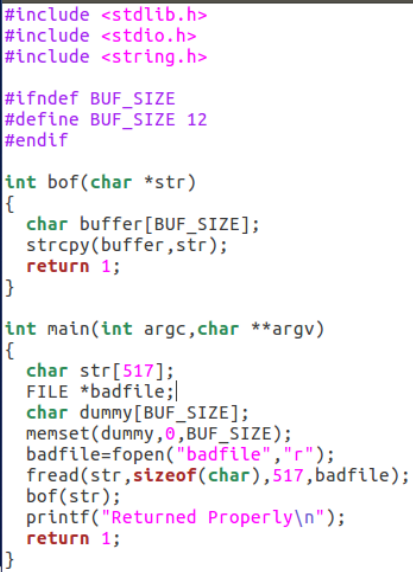


打开堆栈运行：



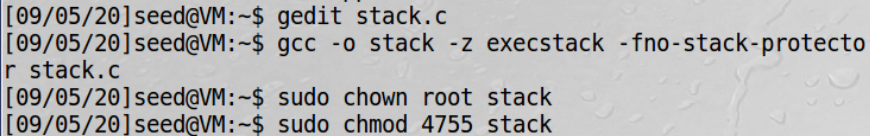
**Task 2.利用漏洞**

编写stack.c：



程序从badfile中读取517字节的数据，然后把它们复制到长度为24字节的缓冲区buffer中。将简单的恶意代码放入badfile中，当程序读取文件时，恶意代码被载入str数组，当程序将str中的内容复制到目标缓冲区时，恶意代码就被存入栈中，恶意代码被放到badfile文件的末尾。之后需要迫使程序跳转到内存的恶意代码，为了达到这个目的，可以利用代码中的缓冲区溢出问题修改返回地址，如果知道恶意代码存放的位置，就能够简单地使用这个地址来覆盖返回地址所在的内存区域，当foo()函数返回时，程序就会跳转到恶意代码存放的地址。

关闭空间地址随机化，stack.c文件的StackGuard关闭，可调用栈，使stack文件成为Set-UID程序：



重新编译，加入调试：

**[09/05/20]seed@VM:~$ gcc -z execstack -fno-stack-protector -g -o stack\_dbg stack.c**

**[09/05/20]seed@VM:~$ touch badfile**

**[09/05/20]seed@VM:~$ gdb stack\_dbg**

**GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1**

**Copyright (C) 2016 Free Software Foundation, Inc.**

**License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>;**

**This is free software: you are free to change and redistribute it.**

**There is NO WARRANTY, to the extent permitted by law. Type "show copying"**

**and "show warranty" for details.**

**This GDB was configured as "i686-linux-gnu".**

**Type "show configuration" for configuration details.**

**For bug reporting instructions, please see:**

**<http://www.gnu.org/software/gdb/bugs/>;.**

**Find the GDB manual and other documentation resources online at:**

**<http://www.gnu.org/software/gdb/documentation/&gt;.**

**For help, type "help".**

**Type "apropos word" to search for commands related to "word"...**

**Reading symbols from stack\_dbg...done.**

**gdb-peda$ b bof**

**Breakpoint 1 at 0x80484f1: file stack.c, line 12.**

**gdb-peda$ run**

**Starting program: /home/seed/stack\_dbg**

**[Thread debugging using libthread\_db enabled]**

**Using host libthread\_db library "/lib/i386-linux-gnu/libthread\_db.so.1".**

**[----------------------------------registers-----------------------------------]**

**EAX: 0xbfffea77 --> 0x34208**

**EBX: 0x0**

**ECX: 0x804fb20 --> 0x0**

**EDX: 0x0**

**ESI: 0xb7f1c000 --> 0x1b1db0**

**EDI: 0xb7f1c000 --> 0x1b1db0**

**EBP: 0xbfffea48 --> 0xbfffec88 --> 0x0**

**ESP: 0xbfffea30 --> 0xbfffec88 --> 0x0**

**EIP: 0x80484f1 (<bof+6>: sub esp,0x8)**

**EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)**

**[-------------------------------------code-------------------------------------]**

**0x80484eb <bof>: push ebp**

**0x80484ec <bof+1>: mov ebp,esp**

**0x80484ee <bof+3>: sub esp,0x18**

**=> 0x80484f1 <bof+6>: sub esp,0x8**

**0x80484f4 <bof+9>: push DWORD PTR [ebp+0x8]**

**0x80484f7 <bof+12>: lea eax,[ebp-0x14]**

**0x80484fa <bof+15>: push eax**

**0x80484fb <bof+16>: call 0x8048390 <strcpy@plt>**

**[------------------------------------stack-------------------------------------]**

**0000| 0xbfffea30 --> 0xbfffec88 --> 0x0**

**0004| 0xbfffea34 --> 0xb7feff10 (<\_dl\_runtime\_resolve+16>: pop edx)**

**0008| 0xbfffea38 --> 0xb7dc888b (<\_\_GI\_\_IO\_fread+11>: add ebx,0x153775)**

**0012| 0xbfffea3c --> 0x0**

**0016| 0xbfffea40 --> 0xb7f1c000 --> 0x1b1db0**

**0020| 0xbfffea44 --> 0xb7f1c000 --> 0x1b1db0**

**0024| 0xbfffea48 --> 0xbfffec88 --> 0x0**

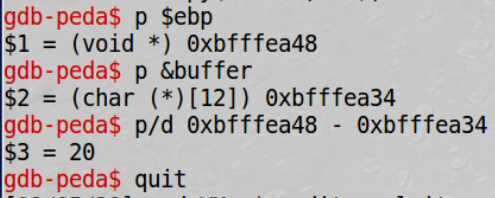
**0028| 0xbfffea4c --> 0x8048574 (<main+106>: add esp,0x10)**

**[------------------------------------------------------------------------------]**

**Legend: code, data, rodata, value**

**Breakpoint 1, bof (str=0xbfffea77 "\bB\003") at stack.c:12**

**12 strcpy(buffer,str);**



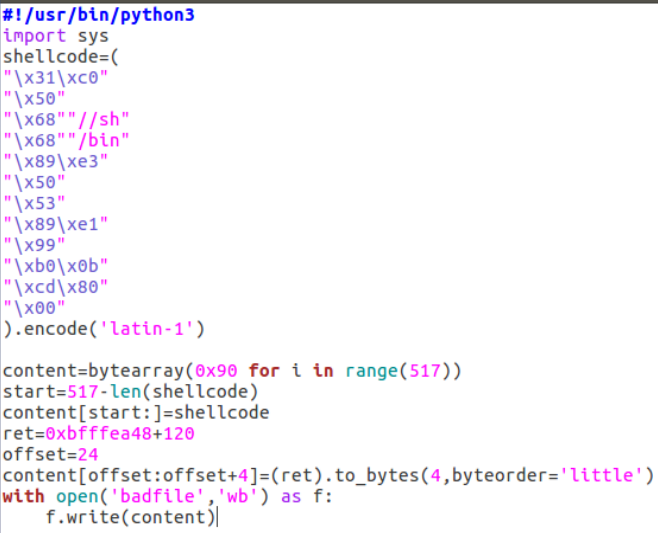
在gdb中，通过b foo命令在bof()函数处设置一个断点，接着用run命令来运行程序，程序将在bof()函数内停下来，这时可以使用gdb的p指令（p指令默认用六十进制打印，p/d表示用十进制打印）来打印帧指针ebp的值以及buffer地址。

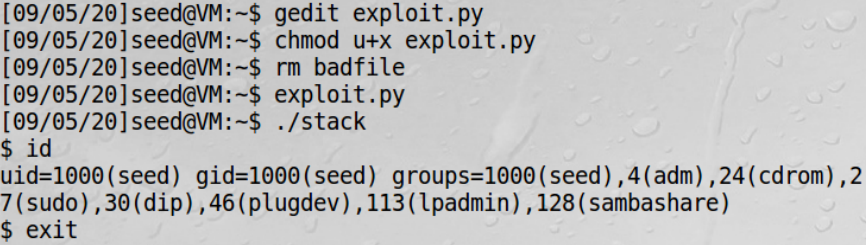
从上面的结果可以看出，帧指针的值是0xbffff168，因此返回地址保存在0xbffff168+4，并且第一个NOP指令在0xbffff168+8，因此可以将0xbffff168+8作为恶意代码的入口地址，把它写入返回地址字段中。

由于输入将被复制到buffer中，为了让输入的返回地址字段准确地覆盖栈中的返回地址区域，需要知道栈中buffer和返回地址之间的距离，这个距离就是返回地址字段在输入数据中的相对位置。

通过计算，可以看到从buffer起始地址到ebp之间的距离为32，由于返回地址区域在ebp指向位置上面的4字节，由此可以知道返回地址区域到buffer之间的距离为36.

编写exploit.py:

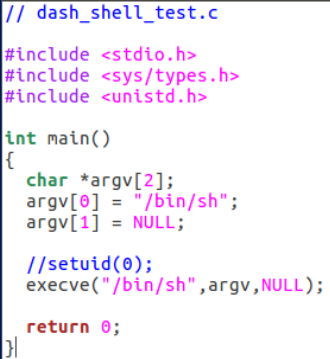


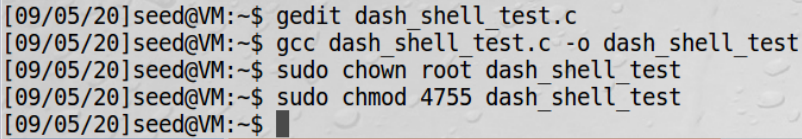


成功获得root的shell权限。

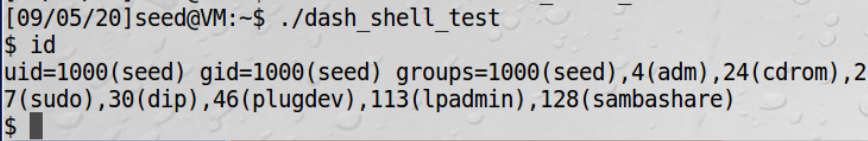
**Task 3.击败dash的对策**

编写程序，设置编译：

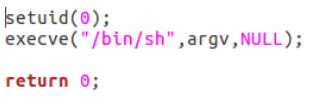




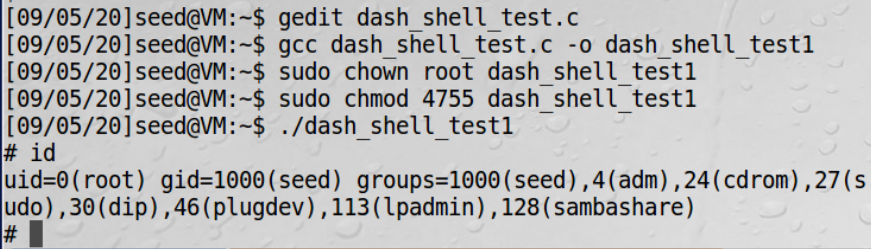
运行dash\_shell\_test：



取消注释：

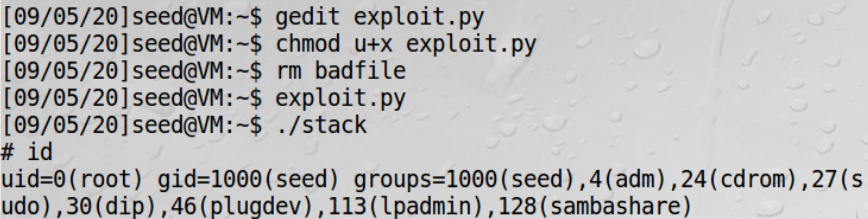


再次编译运行，查看uid:



注释掉setuid(0)之前的uid=1000(seed)，而注释取消之后的uid=0(root)，发生变化。

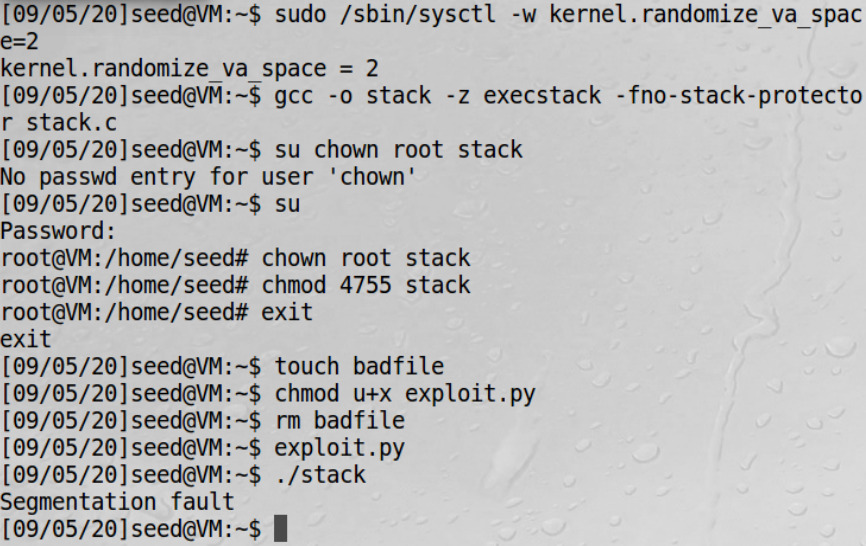
改变shellcode之后再次进行Task 2 攻击：



成功，且发生了与Task 3中前后注释两次setuid一致的变化。

Task 4.解决地址随机化

进行Task2的攻击：



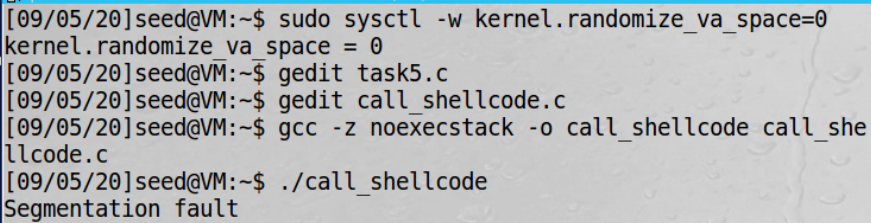
未能成功获取shellcode

未打开shellcode，没有进入蛮力攻击……

\*Point：蛮力攻击未完成。

**Task 5.打开Stack Guard**

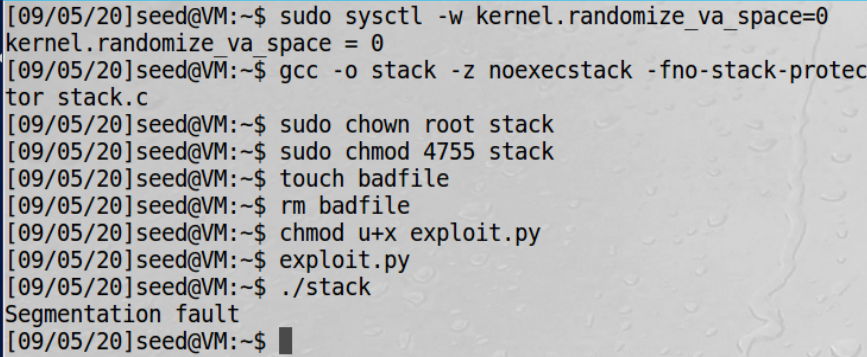
在关闭地址随机化的情况下进行非栈调用的Task 1：



未能成功调用shellcode

Task 6.打开非可执行堆栈保护

关闭地址随机化，进行非可执行堆栈保护TASK 2：



未能成功调用shellcode