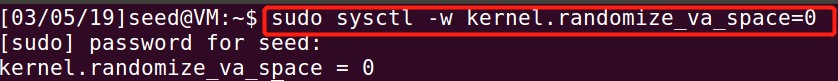
2.1 Turning Off Countermeasures

Linux保护机制有： Address Randomization 地址空间随机化、Non-executable Stack

不可执行栈、“Stack Guard”三种。在利用缓冲区溢出漏洞实现攻击或者单独考量某种保护机制的有效性的过程中，我们需要使某些机制失效。

1. Address Randomization 地址空间随机化



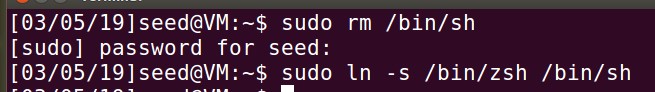
1. “Stack Guard” Protection Scheme

to compile a program example.c with StackGuard disabled ：$ gcc -fno-stackprotector example.c

1. Non-executable Stack 不可执行栈

when compiling programs：For executable stack: $ gcc -z execstack -o test test.c

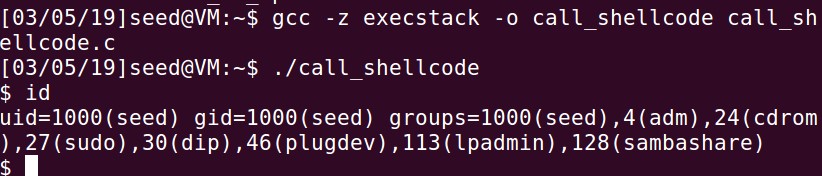
1. Configuring /bin/sh (Ubuntu 16.04 VM only)：



* 1. Task 1: Running Shellcode

Shellcode就是启动shell的code，编译shellcode并运行结果如下，发现成功启动

shell

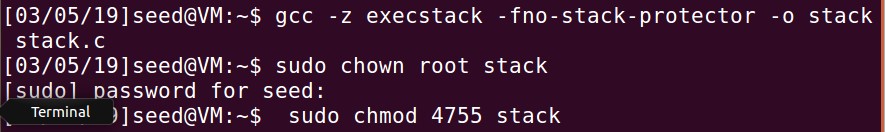


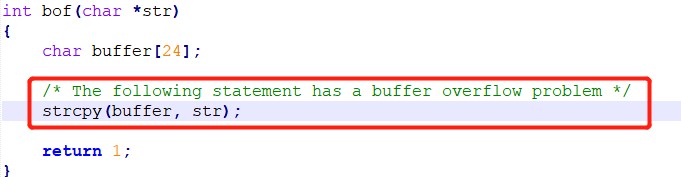
2.3 The Vulnerable Program

实验给出了一个有缓冲区溢出漏洞的程序stack.c，我们需要利用这个漏洞获得root权限。

编译stack.c（在关闭stack guard、允许可执行栈的情况下），将程序变成root拥有的

Set-UID程序





此次缓冲区溢出实验，其本质是利用了strcpy()函数在复制字符串时不检测源字符串长度是否超过了为目的字符串分配的空间的大小，而出现了源字符串数容覆盖其他内存空间的缺陷。

上面的程序从badfile中读取输入，然后传递到在bof（）的另一个buffer中；最初的输入可以有517bytes，但是bof（）中的buffer只有24bytes。因为strcpy（）不进行边界检查，所以就会出现buffer overflow。我们的目标就是填充内容到badfile，使得当程序执行时，可以启动root shell

2.4 Task 2: Exploiting the Vulnerability

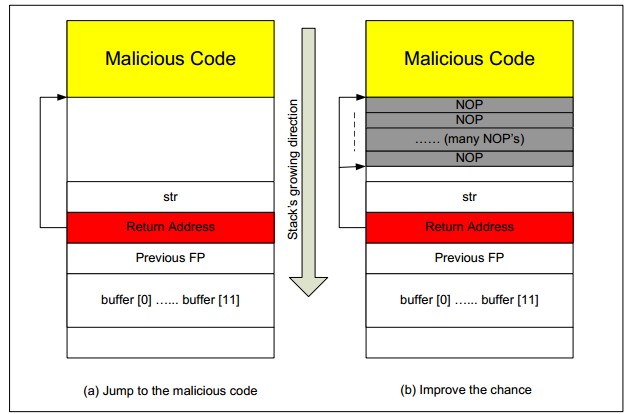
思路：利用exploit.c的buffer[517]合理的填充badfile文件的内容，实现在用户程序 stack.c读取该文件并拷贝到自己的缓冲区（即bof（char \*str）函数栈帧存储空间 buffer[24]）后，由于缓冲区溢出，执行bof函数的返回地址内存单元被覆盖且对应换成

了shellcode的首地址，接下来用户程序执行shellcode并启动了带有用户程序权限的 shell。

具体做法：

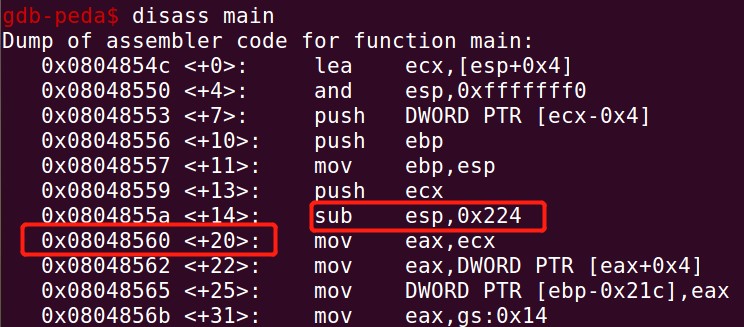
现在找不到return address的地址，所以准备把badfile的下面全部填充成某个地址，让其在return address的地方跳转到return address，然后填充一些nop，然后运行shellcode 这种情况下要找到buffer的基地址，也就是main的str在memory的基址

然后此种情况下，我将buffer以上的100都覆盖成buffer+200的地址，buffer+100-buffer+200之间填充nop，shellcode放在buffer+300的地方

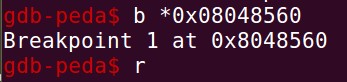


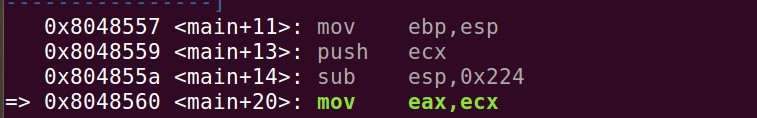
Step1：获得str在内存中的地址，在exploit.c中，shellcode保存在exploit.c的 buffer的某处；在stack.c中，将exploit.c中的buffer完整写入stack.c的str中





图中右边的框就是str起始地址，在左边框地址处设置断点

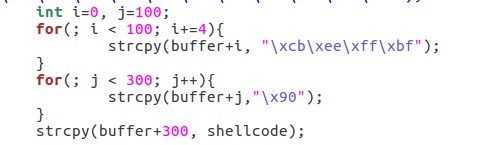


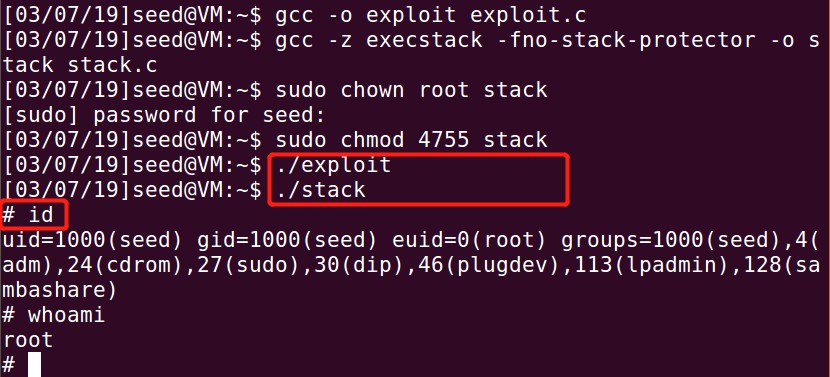




可以得到，漏洞程序读取badfile 文件到缓冲区str，且str的地址为0xbfffee00

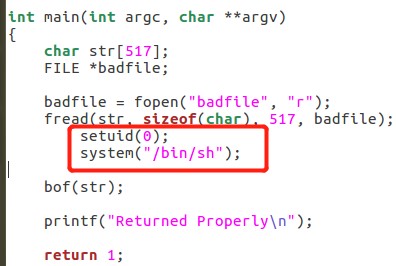
Step2：构造exploit.c，将shellcode的地址写到stack.c中的bof的返回地址处；nop 指令是空指令（0x90），意味着什么都不做

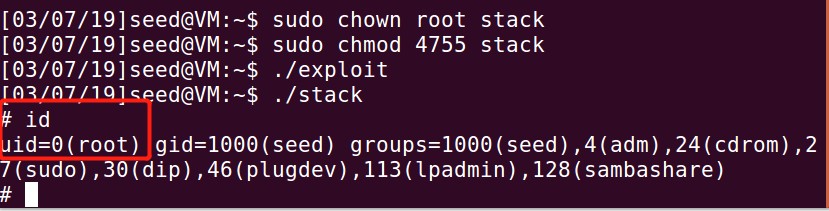




real user ID (uid): 实际用户ID,指的是进程执行者是谁；effective user ID (euid): 有效用户ID,指进程执行时对文件的访问权限；gid：group id step4：修改上面的uid为0

方法：在stack.c中添加setuid(0); system("/bin/sh")

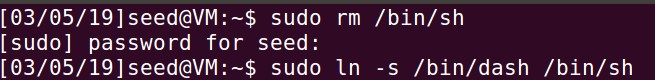




2.5 Task 3: Defeating dash’s Countermeasure

采用另外一个方法：在引入dash程序之前改变user的id为0；方法是在执行shellcode 中execve（）之前，引入setuid（0）

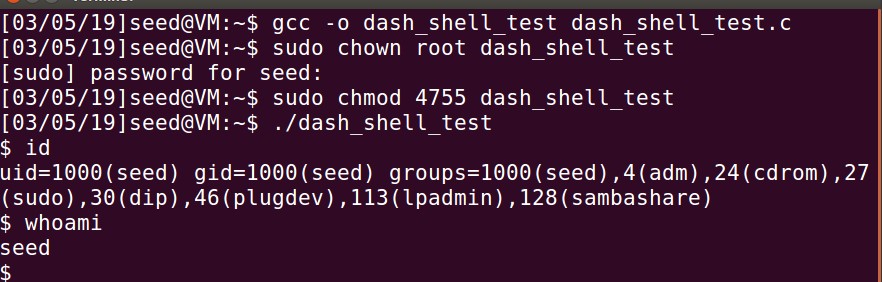
1. 改变/bin/sh，使其链接回到/bin/dash



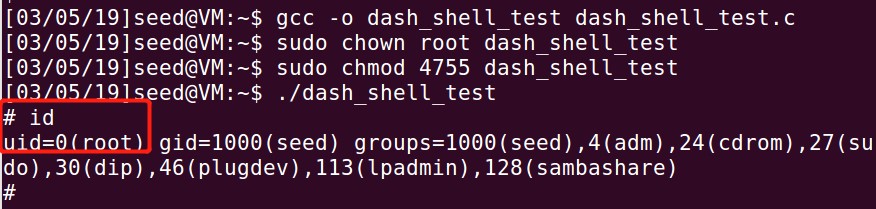
1. 为了查看dash是怎么防止系统调用setuid(0)执行的，在下列条件下运行如下的程序，描述你的观察



* + 1. 注释掉 setuid

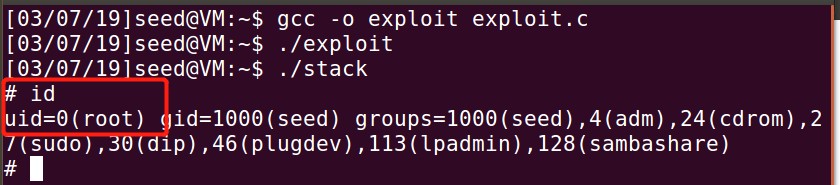


* + 1. 不注释setuid



观察可得，加上setuid(0)则可以成功将uid设置为0，即可以得到root权限；反之不可以

1. 更换shellcode，重新执行task2，描述并解释结果

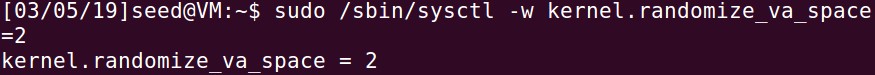


解释：更换后的shellcode在调用/bin/sh之前就调用了setuid(0)，获得了root权限，所以可以调用/bin/sh（root权限下）

2.6 Task 4: Defeating Address Randomization

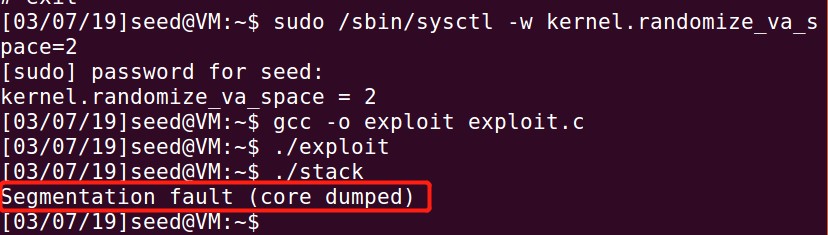
1. 打开ubuntu的地址随机化选项

$ sudo /sbin/sysctl -w kernel.randomize\_va\_space=2

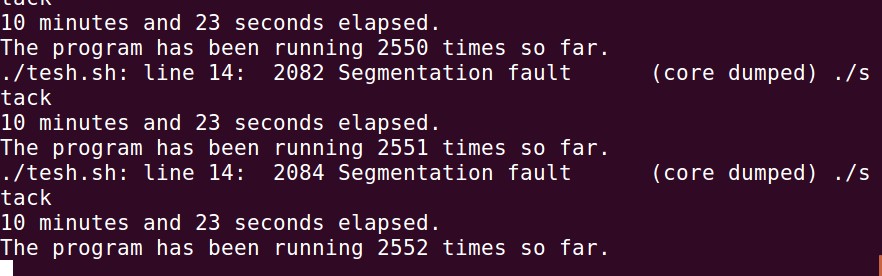


1. 执行task2，描述现象并解释

在运用同样的方法执行task2的时候，因为task2的攻击方法有指定的地址，而task4中使用了地址随机化，使得我们在return address中覆盖的地址并不是shellcode，而不知道指向了哪里，所以出现了错误



1. 循环执行shell脚本，观察现象



在执行了10多分钟后，发现还是没有获得正确的地址，攻击不成功，说明地址随机化这个选项有效地阻止了攻击者暴力穷举进行攻击

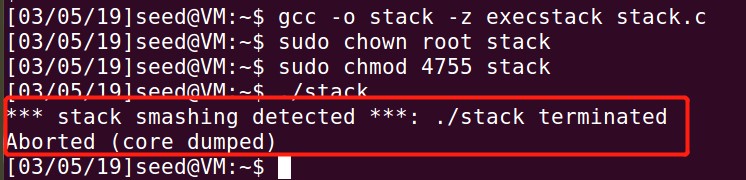
2.7 Task 5: Turn on the StackGuard Protection

1. 关闭地址随机选项

$ sudo sysctl -w kernel.randomize\_va\_space=0

1. 在无-fno-stack-protector选项的情况下编译程序，重复执行task1，观察（可以写

出报告的错误信息）

 攻击不成功，程序崩溃，说明linux有栈保护机制

2.8 Task 6: Turn on the Non-executable Stack Protection

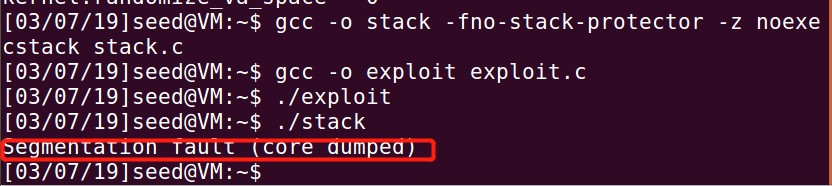
1. 关闭地址随机选项

$ sudo sysctl -w kernel.randomize\_va\_space=0

1. 打开栈不可执行保护机制

$ gcc -o stack -fno-stack-protector -z noexecstack stack.c

1. 重复task2，可以获得shell

 结果显示不能成功；因为开启了栈不可执行机制，使得对栈的使用有边界限制，当越界发生的时候，程序就会触发段错误，所以可以避免攻击者通过修改栈利用buffer overflow 来进行恶意攻击