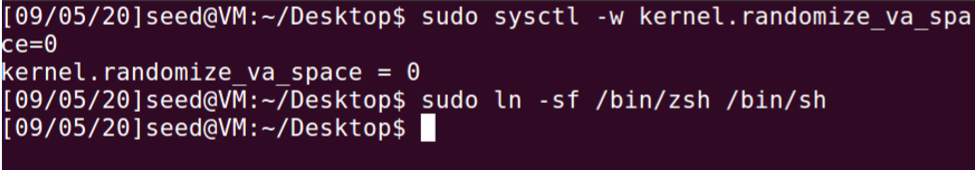
**Lab2-Report**

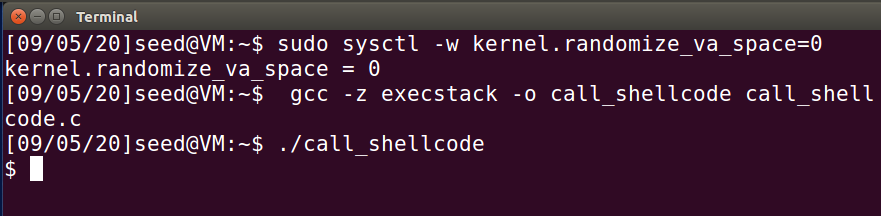
**57118103 郭欣然**

**关闭对抗措施**



**Task1：Running Shellcode**

编译运行shellcode，发现系统启动了shell。



在默认情况下，一个程序的栈是不可执行的，-z execstack选项选项允许从堆栈执行代码。Shellcode中调用execve()系统调用来执行/bin/sh。因此能够启动shell。

**分析shellcode.c需要注意几个问题：**

1、第三条指令将“//sh”而不是“/sh”压入堆栈，是因为我们在这里需要一个32位数字，而“/sh”只有24位。而 “//”等效于“/”，因此我们可以避免使用双斜杠符号。

2、在调用execve()系统调用之前，我们需要将name[0]（字符串的地址），name（数组的地址）和NULL分别存储到％ebx，％ecx和％edx寄存器中。第5行将name [0]存储到％ebx；第8行将名称存储到％ecx；第9行将％edx设置为零。

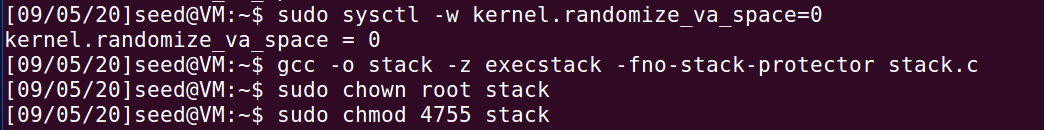
3、当我们将％al设置为11并执行“int $ 0x80”时，系统调用execve()被调用。

**Task2：Exploiting the Vulnerability**

1、关闭地址空间随机化。地址空间随机化（ASLR）是针对缓冲区溢出攻击的防御措施之一，ASLR对程序内存中的一些关键数据区域进行随机化，包括栈的位置、堆和库的位置等，目的让攻击者难以猜测所注入的恶意代码在内存中的具体位置。

2、编译stack.c程序，添加-fno-stack-protector和“-z execstack”选项，以关闭StackGuard和不可执行的堆栈保护。

后将程序改为root拥有的setuid程序，使得当恶意代码被执行后启动的是root shell。



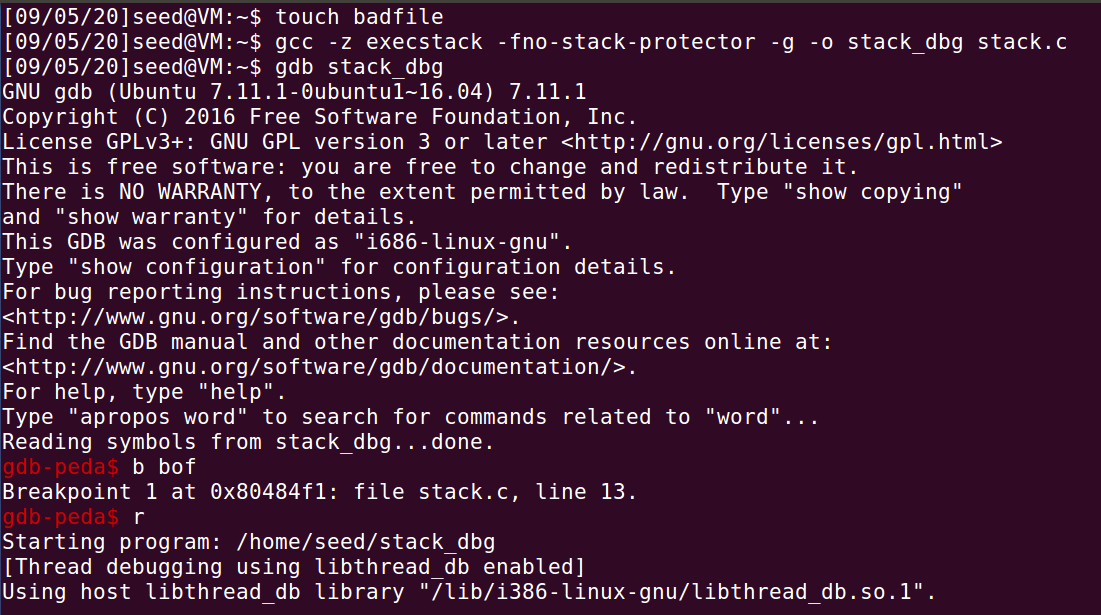
上述stack程序有一个缓冲区溢出漏洞。它首先从名为badfile的文件中读取517个字节到str，然后将str传递到函数bof()中的另一个缓冲区buffer，其中buffer只有24个字节。由于strcpy()不检查边界，因此会发生缓冲区溢出。

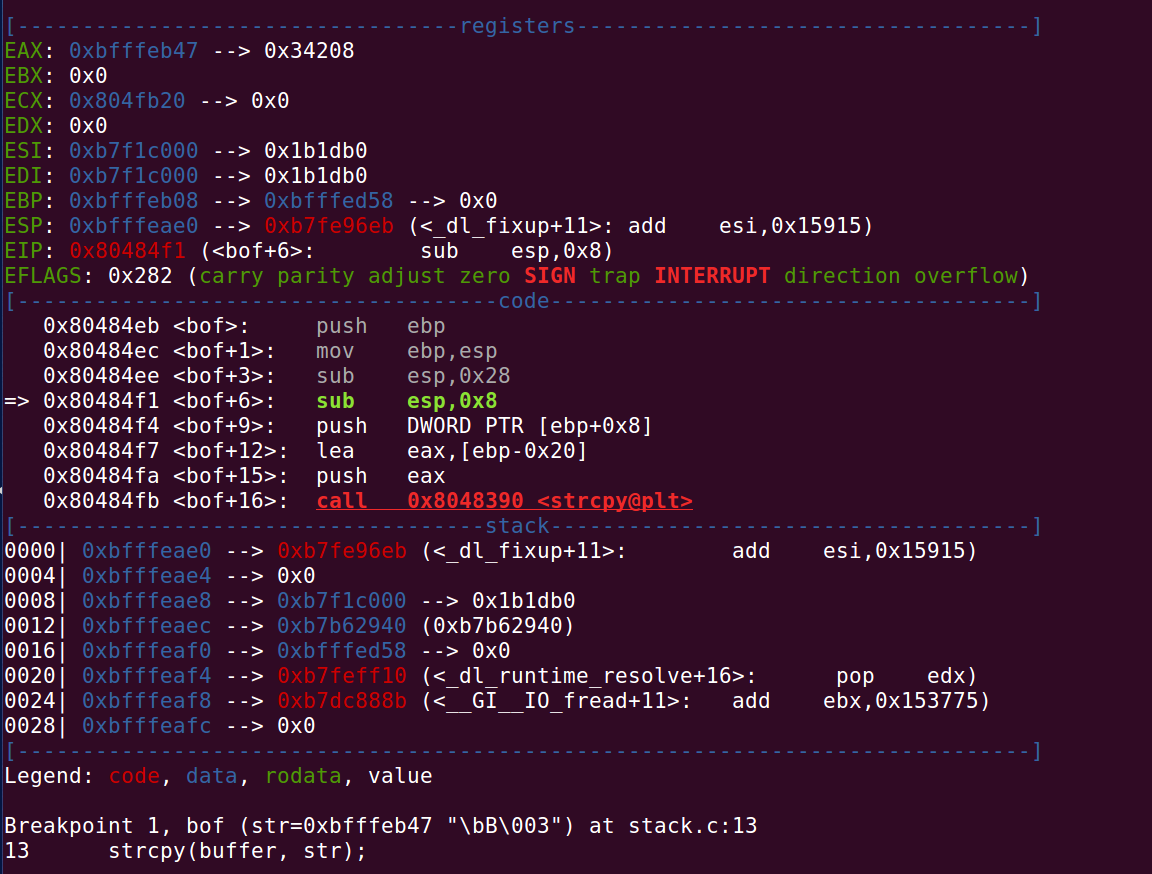
由于该程序是setuid程序，因此，如果普通用户可以利用此缓冲区溢出漏洞，则普通用户可能能够获得root shell。

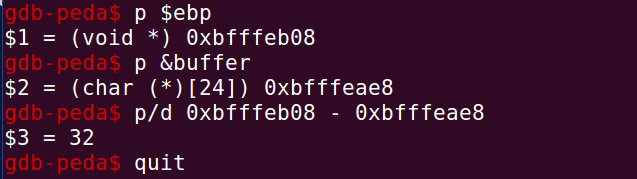
badfile文件受用户控制。因此，我们在badfile中添加一些内容，使得当易受攻击的程序将内容复制到其缓冲区中时，可以生成根shell。

为了实现这一目的，我们可以利用代码中的缓冲区溢出问题修改返回地址，使得foo()函数返回时，程序就跳转到恶意代码存放的地址。因此如果知道恶意代码存放的位置，就能利用这个地址来覆盖返回地址所在的内存区域达到需求。

3、调试stack程序，以获得所需的几个地址和相对位置。







加入-g选项重新编译stack程序进行调试，

① 通过b foo命令在bof函数处设置一个断点，接着用r命令来运行程序，程序将在bof()函数内停下来，再p命令以十六进制来打印bof栈首址ebp的值以及buffer地址。

② 从上面的结果可以看出，ebp的地址为0xbffffeb08，因此返回地址保存在0xbffffeb08+4，并且第一个NOP指令在0xbffffeb08+8，因此可以将0xbffffeb08+n（n≥8）作为恶意代码的入口地址，并将其写入返回地址字段中。

③ 由于badfile中的内容将被复制到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badfile中输入数据的相对位置。

③ 通过计算，可以看到从buffer起始地址为0xbffffeae8，到ebp之间的距离为32，由于返回地址区域在ebp向上4~8字节处，由此可以知道返回地址区域到buffer之间的距离为36.

4、构造输入文件badfile

① badfile中的shellcode中存放恶意代码，首先创建了一个长度为517个字节的数组content，并用0x90（NOP）填充整个数组，最后把恶意代码放到数组的尾部。

② 我们选择使用0xbffffeb08+80作为返回值，需要把它填入到content数组对应栈中的返回值区域。由之前的gdb调试结果可知，返回值区域从第36个字节开始，到第40个字节结束，不包括第40个字节，所以设置offset = 36，用[offset:offset+4]来存储返回地址。

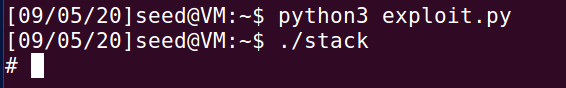
③ 在x86等体系结构的计算机使用的是小端字节顺序，一个多字节组成的数据在内存中存放时，最低位字节放到低地址处，因此把一个4字节的地址写入内存时，设定使用小端字节顺序。

④ 返回地址的内容没有使用0xbffffeb08+8是因为，buffer在创建时操作系统可能多预留一些地址，因此0xbffffeb08+8可能并不是返回地址的首址。因此将n设置的大一些，这里设置n=80。即返回地址为0xbffffeb08+80.

⑤ 注意0xbffffeb08 + n不应该使用任何字节包含0，因为’\0’的ASCII码值为0x00，strcpy函数碰到00会停止复制内容到缓冲区，因此会造成缓冲区溢出攻击的失败。

5、运行程序进行攻击

运行exploit.py与stack进行攻击，可以看到得到了root shell，攻击成功。



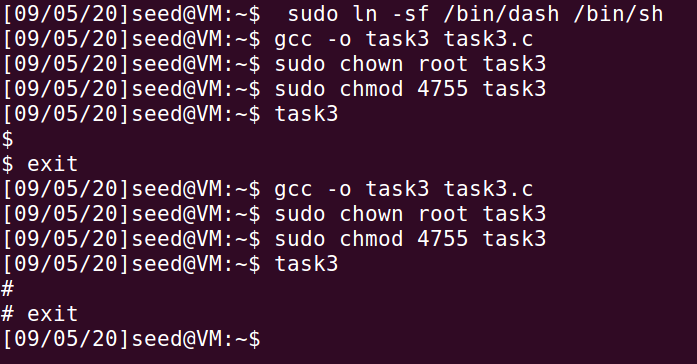
**Task 3: Defeating dash’s Countermeasure**

当检测到RUID≠EUID时，系统会将dash shell权限降级。

为了不使权限降级，其中一种方法是在execve（）前调用setuid（0）

1、对setuid（0）系统调用的功能进行测试

首先更改连接指向/bin/dash

、

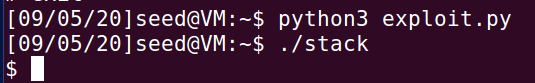
在注释 setuid（0） 的条件下编译并使之成为 root 拥有的 setuid 程序时，运行程序，发现权限被降低。

取消注释后再次做相同操作，发现成功获得 root 权限。

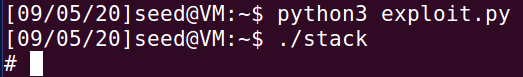
说明 setuid（0）可以有效抵抗对抗措施。

2、实施攻击

按照task2的方法进行栈溢出攻击，系统自动判断 euid 和 ruid 不相等，因此root权限降低为用户权限，攻击失败。

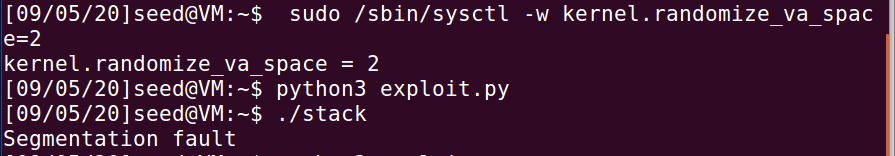


更改shellcode，在execve（）前增加了setuid（）系统调用，进行栈溢出攻击，发现权限未被降低，得到了rootshell，攻击成功。这是因为在调用execve之前setuid使ruid=0（root），使系统误判为 root 用户的使用，所以权限不会降级，获取到 root 权限

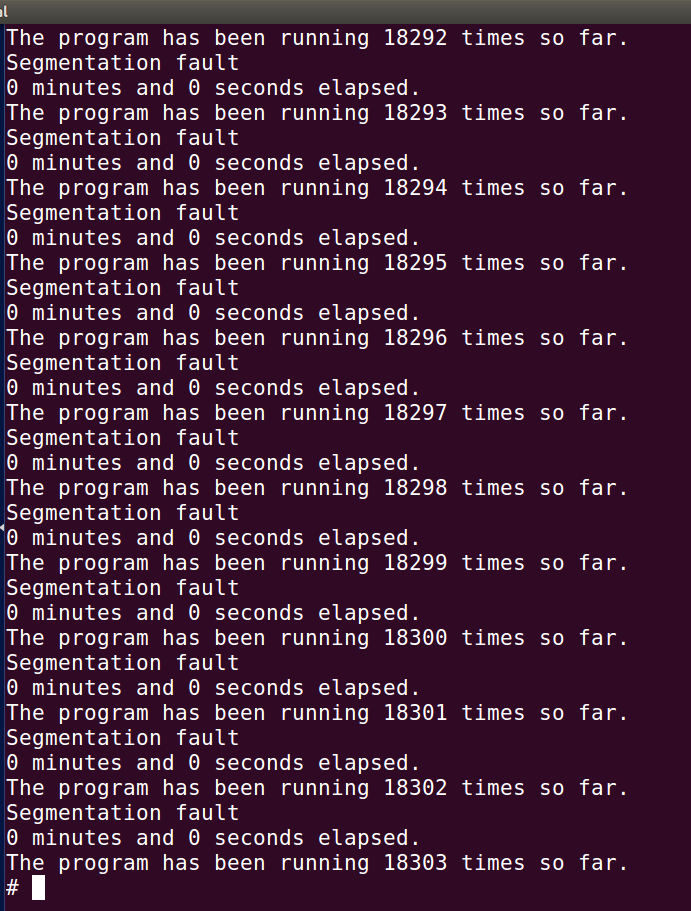


**Task4：Defeating Address Randomization**

使用 sudo /sbin/sysctl -w kernel.randomize\_va\_space=2 命令启用地址随机化，按照task2中相同方法进行攻击发现出现段错误。这是因为启用了地址随机化，让攻击者难以猜测所注入的恶意代码在内存中的具体位置。



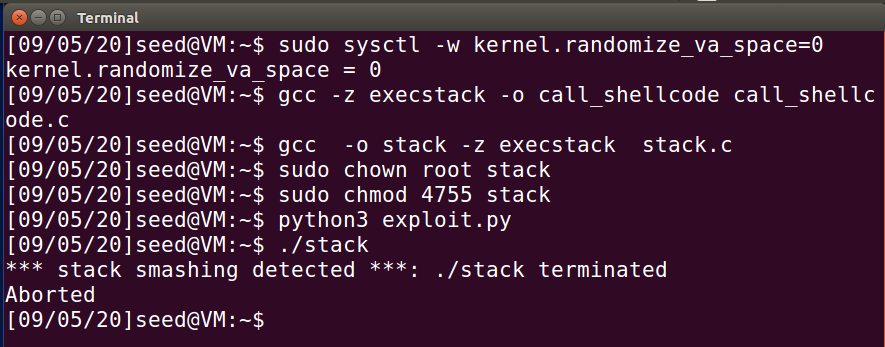
采用暴力破解方法获取shell，复制命令并运行。



破解成功，获得了root shell，共运行了18303次。

**Task5：Turn on the StackGuard Protection**

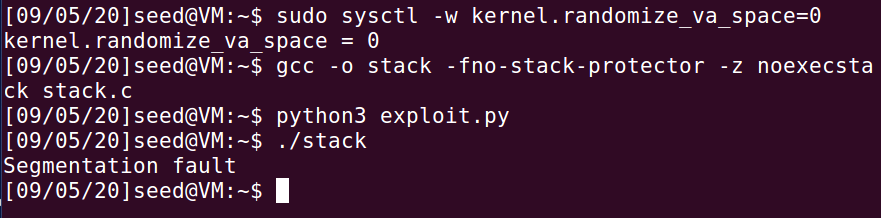
关闭地址随机化，并去除-fno-stack-protector。意思是启用StackGuard的保护机制，它能够抵御基于栈的缓冲区溢出攻击，它的主要思想是在代码中添加一些特殊数据和检查机制，从而可以检测到缓冲区溢出的发生。



重新编译stack.c，再次执行task1，运行task2进行栈溢出攻击，发现系统会检测到栈被破坏，程序异常退出

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

关闭地址随机化，开启不可执行栈保护机制。



重新编译stack.c，运行task2进行栈溢出攻击，发现出现了段错误，这是由于启用了栈不可执行机制使得 shellcode 不能在栈上运行导致失败。