**Buffer Overﬂow Vulnerability Lab**

**1.1 Turning Off Countermeasures**

Ubuntu和其他一些Linux系统中，使用地址空间随机化来随机堆（heap）和栈（stack）的初始地址，这使得猜测准确的内存地址变得十分困难，而猜测内存地址是缓冲区溢出攻击的关键。因此本次实验中，我们使用以下命令关闭这一功能：

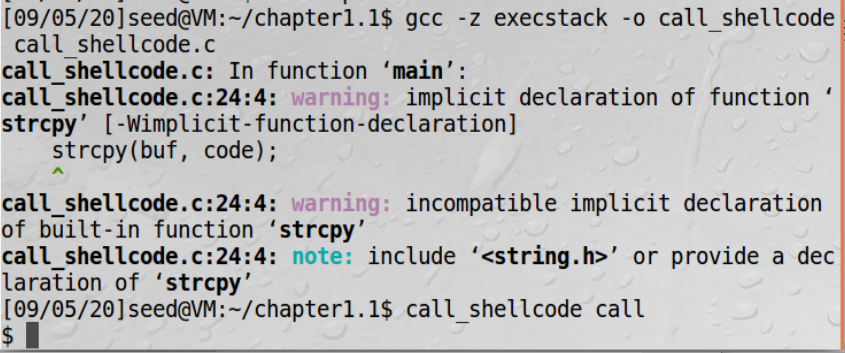


此外，为了进一步防范缓冲区溢出攻击及其它利用shell程序的攻击，许多shell程序在被调用时自动放弃它们的特权。因此，即使你能欺骗一个Set-UID程序调用一个shell，也不能在这个shell中保持root权限，这个防护措施在/bin/bash中实现。

linux系统中，/bin/sh实际是指向/bin/bash或/bin/dash的一个符号链接。为了重现这一防护措施被实现之前的情形，我们使用另一个shell程序（zsh）代替/bin/bash。下面的指令描述了如何设置zsh程序：



**1.2 Task1: Running Shellcode**

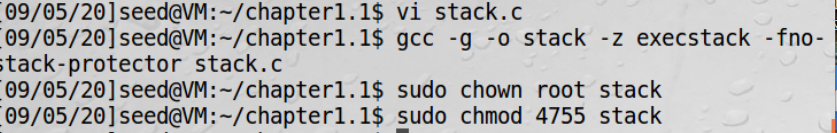


如图可见，一个新的终端被调用。

**1.3 The Vulnerable Program**

通过代码可以知道，程序会读取一个名为“badfile”的文件，并将文件内容装入“buffer”。

编译该程序，并设置SET-UID和root



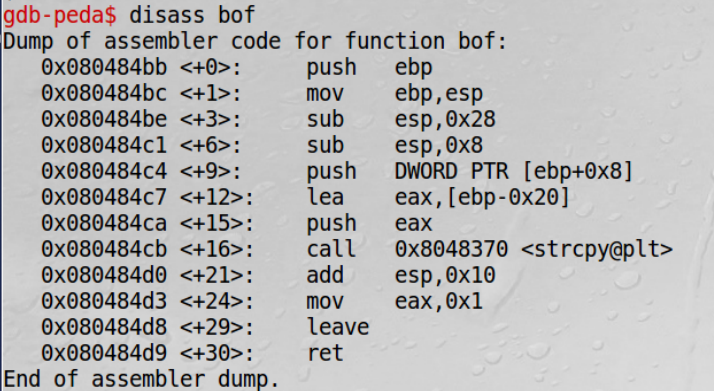
**1.4 Task2: Exploiting the Vulnerabilit**

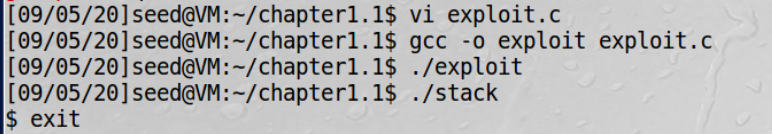
**确定shellcode 存放地址**



shellcode起始位置: **0xbfffea57+0x64=0xbfffeabb**

**确定return address：**

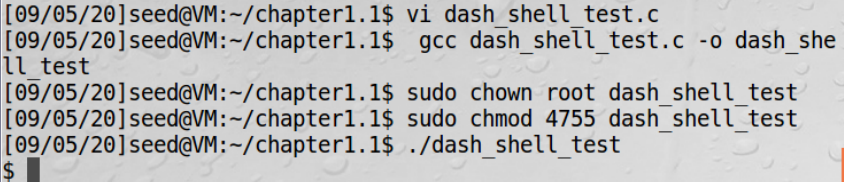




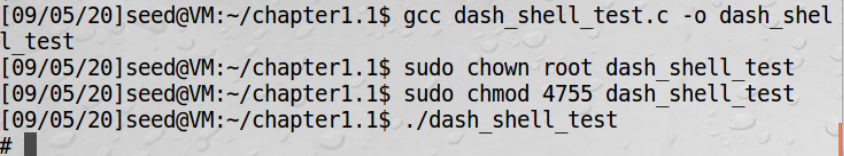
**1.5 Task3: Defeating dash’s Countermeasure**



**未取消注释：** 获取的是用户权限的shell



**取消注释：** 利用setuid(0),获取root权限shell。



**1.6 Task4: Defeating Address Randomization**

在32位linux机器上，栈只有19bit的entropy（不知道啥意思），这意味着栈的基地址有2^19=524288种可能性，这个数字并不高，可以通过暴力破解轻松耗尽，在本次任务，我们使用这样的方法在32位机器上来击败address randomization对策，首先我们使用如下命令打开Ubuntu的address randomization，接着使用与任务2同样的方法攻击stack.c。



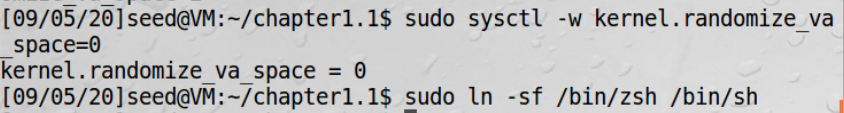
然后我们使用暴力手段重复攻击漏洞程序，希望我们放在badfile中的地址最终可以正确。你可以使用如下的shell脚本来循环运行漏洞程序，如果你的攻击成功，这个脚本会停止，否则它会一直运行。

**1.7 Task5: Turn on the Stack Guard Protection**

**1.8 Task6: Turn on the Non-executable Stack Protection**

**Return-to-libc Attack Lab**

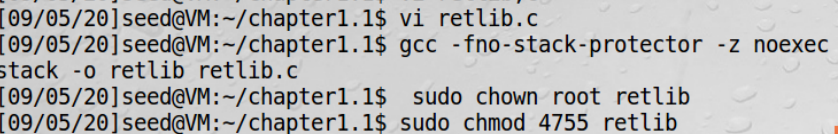
**1.1 Turning off countermeasures**

****

**1.2 The Vulnerable Program**

此程序存在缓冲区溢出漏洞，它首先从badfile文件中读取40字节的输入到一个12字节的缓冲区（buffer）中，由于函数fread（）没有检查buffer边界，BufferOverflow将会发生。这个程序是一个拥有者为root的Set-UID程序，所以如果一个正常用户可以利用这个bufferoverflow漏洞，用户便能够获取root shell，需要注意的是程序从一个角度badfile的文件中获取输入，而这个badfile是由用户提供的，所以，我们可以构造一个文件，当漏洞程序复制文件内容到它的buffer中时，roote shell便可以获得。

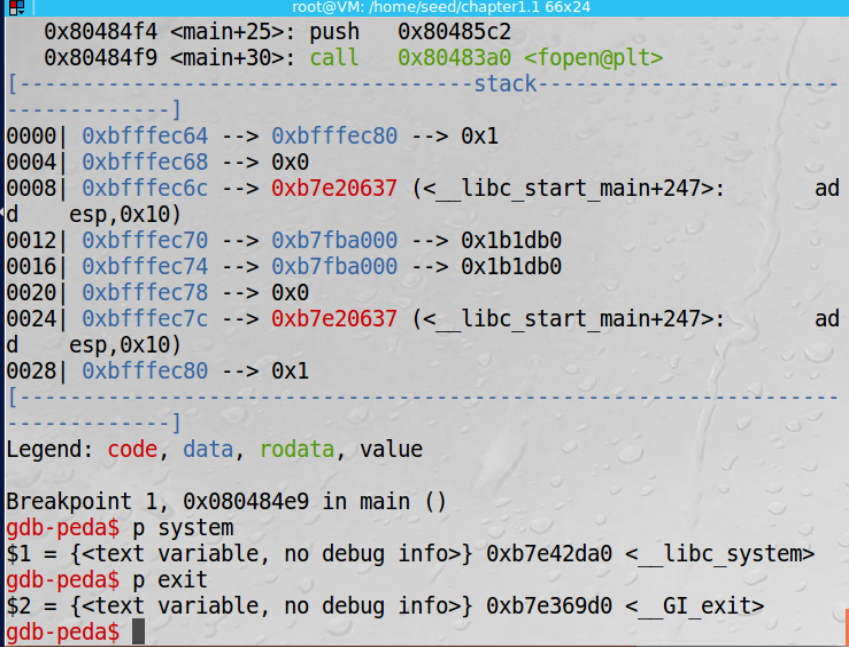
首先编译代码并且将其改变为拥有者为root的Set-UID程序，不要忘记使用 -fno-stack-protector 选项来关闭 StackGuard保护 和使用 -z noexecstack选项来打开non-exeutable stack保护。



**2.3 Task1: Finding out the addresses of libc functions**

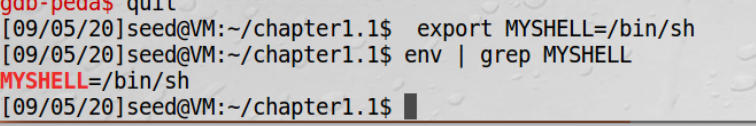
在ReturnToLibc攻击里，我们需要跳转到一些现存的已经被加载到内存的代码中，我们在攻击中将会使用Libc函数库中的system()和exit()函数，所以我么需要找到他们的地址，有许多方法可以确定，但是使用gdb调试器会简单一点

在上述gdb命令里，我们首先在main函数中设置了一个断点，接着使用命令r运行debugged程序，程序将会断在断点处，我们可以打印出system()和exit()函数的地址

****

**2.4 Task2: Putting the shell string in the memory**

我们的攻击策略是跳转到system()函数并且通过它来执行任意命令，我们希望通过system()函数执行/bin/sh程序，因此，命令字符串“/bin/sh”必须首先被放入内存，并且我们需要知道它的地址(这个地址需要传递给system()函数)，有许多方法可以实现这个目标，我们选择通过**环境变量**来实现。  
当我们从shell prompt执行一个程序时,shell通常生成一个子进程来执行该程序，所有导出的shell变量将会变成子进程的环境变量，这使得我们可以任意的放置一些任意的字符串到子进程的内存中。我们定义一个新的shell变量MYSHELL，并且使它包含字符串/bin/sh。从下列命令，我们可以确保使字符串进入子进程中，并且通过运行在子进程里的env命令打印出来。

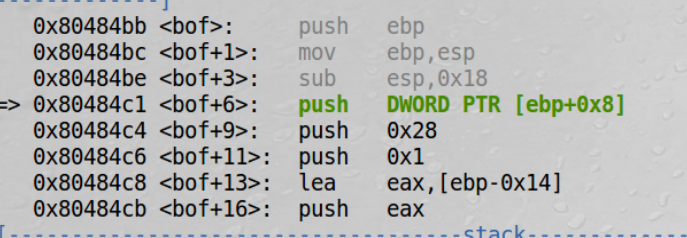
****

我们将会使用这个变量的地址作为system()调用的参数

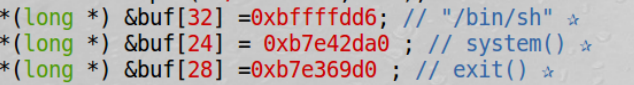
****

**2.5 Task3: Exploiting the buffer-overﬂow vulnerability**

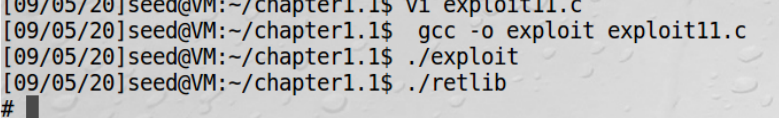
我们的目标是溢出缓冲区，覆盖bof函数的返回地址，将其指向system()函数，并且将/bin/sh作为system()的参数，最后执行exit()函数范围，故我们必须知道bof函数的返回地址，通过如下操作可以知晓：

****

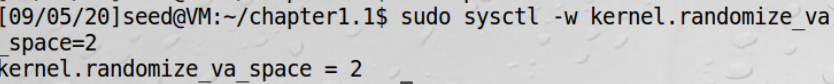
补充完整代码：

****

实验结果如下：

****

**2.6 Task4: Turning on address randomization**

****

**2.7 Task5: Defeat Shell’scountermeasure**

**2.8 Task6: Defeat Shell’scountermeasure without putting zeros in input**