**实验二 Buffer Overflow Vulnerability Lab**

**Return-to-libc Attack Lab**

**实验报告**

57117202 朱惟璇

**一、Buffer Overflow Vulnerability Lab（缓冲区溢出漏洞实验）**

**The BUF SIZE value for this lab is: 100**

**1、Turning Off Countermeasures**

Ubuntu和其他Linux发行版已经实现了几种安全机制，以使缓冲区溢出攻击变得困难。为了简化攻击，需要先禁用它们。稍后再启用它们，查看攻击是否仍然可以成功。

**地址空间随机化。**Ubuntu和其他几个基于Linux的系统使用地址空间随机化来随机化堆和栈的起始地址。这使得猜测确切的地址变得困难。猜测地址是缓冲区溢出攻击的关键步骤之一。在本实验中，使用以下命令禁用此功能：

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

**StackGuard保护方案。**GCC编译器实现了一种称为StackGuard的安全机制，以防止缓冲区溢出。在这种保护的情况下，缓冲区溢出攻击将不起作用。可以在编译期间使用-fno-stack-protector选项禁用此保护。例如，要在禁用StackGuard的情况下编译程序example.c，可以执行以下操作：

$ gcc -fno-stack-protector example.c

**不可执行的堆栈。**Ubuntu曾经允许可执行堆栈，但是现在已经发生了变化：程序（和共享库）的二进制映像必须声明它们是否需要可执行堆栈，即它们需要在程序标头中标记一个字段。内核或动态链接器使用此标记来决定是使此正在运行的程序的堆栈是可执行的还是不可执行的。标记是由最新版本的gcc自动完成的，默认情况下，堆栈设置为不可执行。要更改此设置，请在编译程序时使用以下选项：

For executable stack:

$ gcc -z execstack -o test test.c

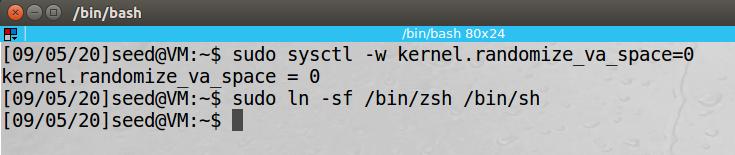
For non-executable stack:

$ gcc -z noexecstack -o test test.c

**配置/bin/sh（仅适用于Ubuntu 16.04 VM）。**

$ sudo ln -sf /bin/zsh /bin/sh

配置如图：

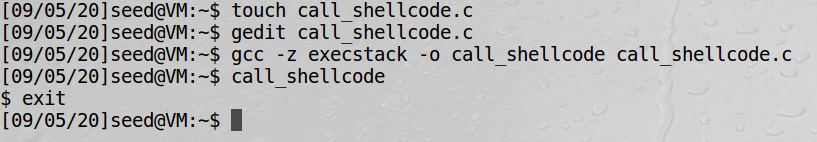


**2、Task 1: Running Shellcode**

Shellcode是启动Shell的代码。必须将其加载到内存中，以便迫使易受攻击的程序跳转至该内存。以下程序显示了如何通过执行存储在缓冲区中的shellcode来启动shell。请编译并运行以下代码，并查看是否调用了shell。

使用以下gcc命令编译以上代码。运行程序并描述观察结果。请不要忘记使用execstack选项，该选项允许从堆栈执行代码。没有此选项，程序将失败。

$ gcc -z execstack -o call\_shellcode call\_shellcode.c



如图所示，可以启动shell。

**3、The Vulnerable Program**

提供以下程序，该程序在Line➀中具有缓冲区溢出漏洞。利用此漏洞并获得root特权。

上面的程序有一个缓冲区溢出漏洞。它首先从名为badfile的文件中读取输入，然后将该输入传递到函数bof（）中的另一个缓冲区。原始输入的最大长度可以为517个字节，但bof（）中的缓冲区只有BUF SIZE个字节长，小于517。由于strcpy（）不检查边界，因此会发生缓冲区溢出。由于此程序是root拥有的Set-UID程序，因此，如果普通用户可以利用此缓冲区溢出漏洞，则该用户可能能够获得root shell。应当注意，程序从名为badfile的文件获取输入。此文件受用户控制。现在，我们的目标是为badfile创建内容，以便当易受攻击的程序将内容复制到其缓冲区中时，可以生成一个根shell。

编译。要编译上述易受攻击的程序，请不要忘记使用-fno-stack-protector和“ -z execstack”选项关闭StackGuard和不可执行的堆栈保护。编译之后，使该程序成为root拥有的Set-UID程序。通过首先将程序的所有权更改为root，然后将权限更改为4755以启用Set-UID位，可以实现此目的。应该注意的是，更改所有权必须在打开Set-UID位之前完成，因为所有权更改将导致Set-UID位被关闭。

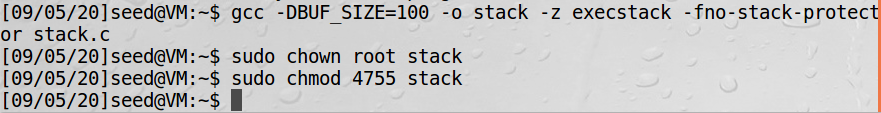
// Note: N should be replaced by the value set by the instructor

$ gcc -DBUF\_SIZE=N -o stack -z execstack -fno-stack-protector stack.c

$ sudo chown root stack ➀

$ sudo chmod 4755 stack ➁

如图所示，可以运行：



**4、Task 2: Exploiting the Vulnerability（利用漏洞）**

提供了部分完成的利用代码 “exploit.c”。该代码的目的是为badfile构造内容。在此代码中已提供shellcode。需要开发其余部分。

完成上述程序后，编译并运行它。这将生成badfile的内容。然后运行易受攻击的程序堆栈。如果漏洞利用程序正确实施，则应该能够获得root shell。

重要提示：请先编译易受攻击的程序。请注意，生成badfile的exploit.c程序可以在启用了默认StackGuard保护的情况下进行编译。这是因为不会在该程序中溢出缓冲区。将溢出stack.c中的缓冲区，该缓冲区是在禁用StackGuard保护的情况下编译的。

$ gcc -o exploit exploit.c

$./exploit // create the badfile

$./stack // launch the attack by running the vulnerable program

# <---- Bingo! You’ve got a root shell!

尽管已经获得了“＃”提示，但是真实用户ID仍然是自己（有效用户ID现在是root）。可以通过键入以下内容进行检查：

# id

uid=(500) euid=0(root)

如果将许多命令作为Set-UID root进程而不是作为root进程执行，则它们的行为会有所不同，因为它们会识别出真实的用户ID不是root。为解决这个问题，您可以运行以下程序将真实用户ID转换为root。这样，您将拥有一个真正的root进程，该进程更加强大。

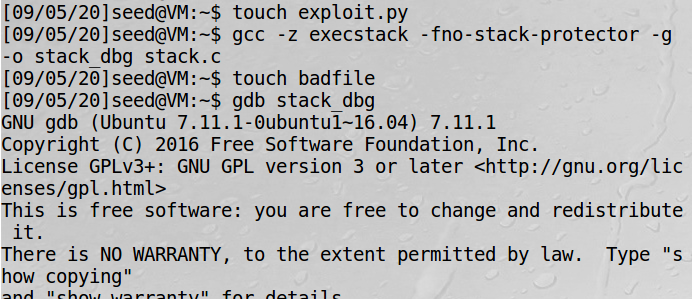
void main()

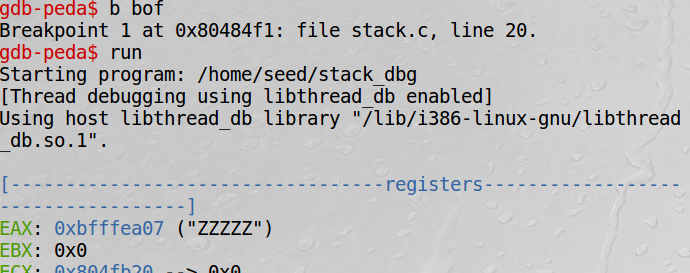
{

setuid(0); system("/bin/sh");

}

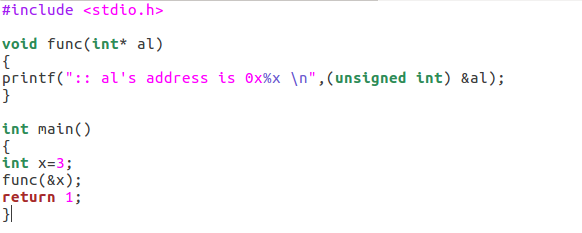
首先使用gdb找到当前指针所指地址以及返回值地址，得到缓冲区地址与返回地址之间的距离，可以算出offset为112：





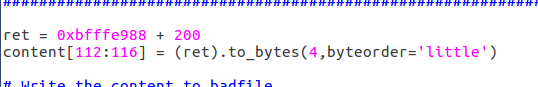


恶意代码的地址：恶意代码写在badfile中，并作为参数传递给易受攻击的函数

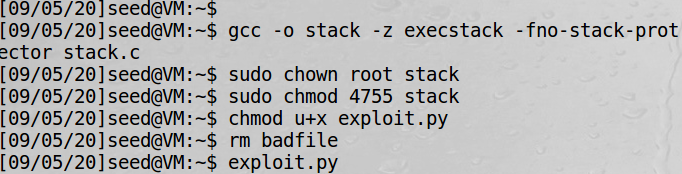


![C:\Users\zhuweixuan\AppData\Roaming\Tencent\Users\467718102\QQ\WinTemp\RichOle\@IS`%WUE$O)NB{E](MVR`PI.png](data:image/png;base64,)

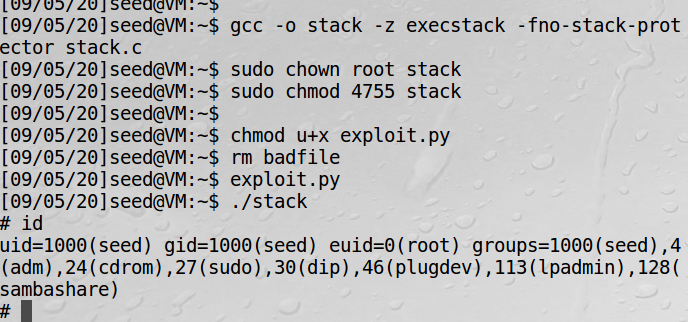
根据上面得到的地址和结论修改exploit.py：



进行编译、并运行生成badfile文件：



结果如图所示，运行stack可执行程序，拿到了root权限：



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

如前所述，Ubuntu 16.04中的dash外壳会在检测到有效UID不等于实际UID时放弃特权。这可以从dash程序的变更日志中观察到。可以在第①行中看到另一个检查，该检查比较了真实和有效的用户/组ID。

dash中实施的对策可以被击败。一种方法是不在shellcode中调用/bin/sh。相反，我们可以调用另一个Shell程序。此方法要求系统中存在另一个Shell程序，例如zsh。另一种方法是在调用dash程序之前将受害者进程的实际用户ID更改为零。可以通过在shellcode中执行execve（）之前调用setuid（0）来实现此目的。在此任务中将使用这种方法。首先更改/bin/sh符号链接，使其指向/bin/dash：

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

要查看dash中的对策如何工作以及如何使用系统调用setuid（0）消除对策，编写了以下C程序。首先注释掉行➀，然后将其作为Set-UID程序运行（所有者应为root）；请描述观察。然后取消注释第➀行，然后再次运行该程序；请描述观察。

可以使用以下命令来编译和设置以上程序（需要使其成为根用户拥有的Set-UID程序）：

$ gcc dash\_shell\_test.c -o dash\_shell\_test

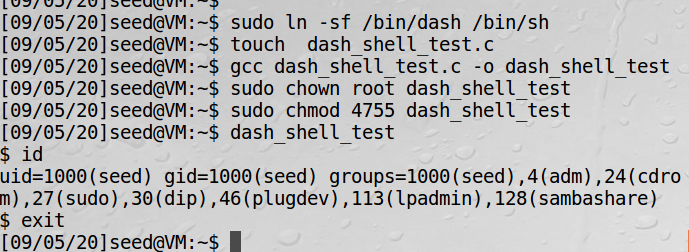
$ sudo chown root dash\_shell\_test

$ sudo chmod 4755 dash\_shell\_test

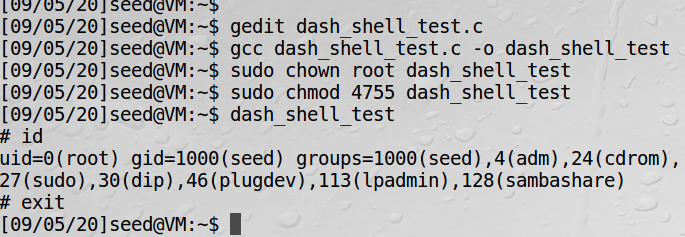
从上面的实验中，将看到seuid（0）有所作为。让我们在调用execve（）之前，在shellcode的开头添加用于调用此系统调用的汇编代码。

更新的shellcode添加了4条指令：（1）在第2行中将ebx设置为零，（2）通过第1行和第3行将eax设置为0xd5（0xd5是setuid（）的系统调用号码），以及（3）执行系统 在第4行中调用。使用此shellcode，当/bin/sh链接到/bin/dash时，我们可以尝试对易受攻击的程序进行攻击。使用上面的shellcode修改exploit.c或exploit.py; 再次尝试执行任务2的攻击，看看是否可以获得根shell。请描述并解释结果。

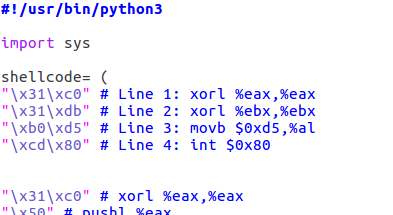
更改/bin/sh符号链接，使其指向/bin/dash，运行该程序，并该执行文件改成root的Set-UID程序，发现不能获得root权限：



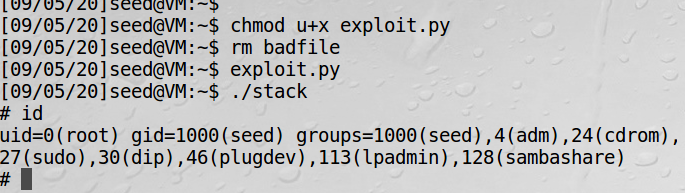
把setuid(0)去掉注释，重新编译代码dash\_shell\_test.c，执行该程序，发现可以获得root权限：



把代码加入exploit.py：



再次运行攻击，可以获得root shell：



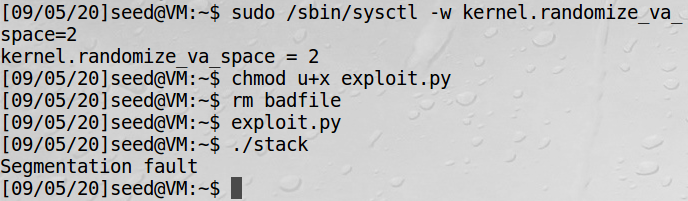
**6、Task 4: Defeating Address Randomization（击败地址随机化）**

在32位Linux机器上，堆栈仅具有19位熵，这意味着堆栈基地址可以具有219=524288种可能性。这个数字不是很高，并且可以用蛮力方法轻松耗尽。在此任务中，使用这种方法来克服32位VM上的地址随机化对策。首先，使用以下命令打开Ubuntu的地址随机化。执行任务2中开发的相同攻击。请描述并解释观察。

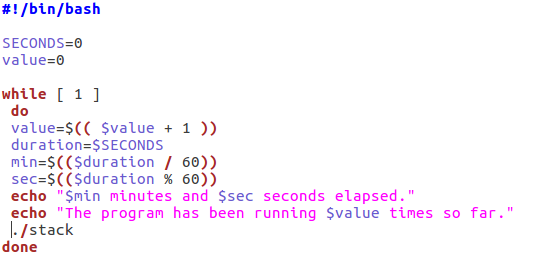
$ sudo /sbin/sysctl -w kernel.randomize\_va\_space=2

然后使用蛮力方法反复攻击易受攻击的程序，希望放入badfile中的地址最终可以正确。使用以下shell脚本在无限循环中运行易受攻击的程序。如果攻击成功，脚本将停止；否则，它将继续运行。请耐心等待，因为这可能需要一段时间。请描述观察。

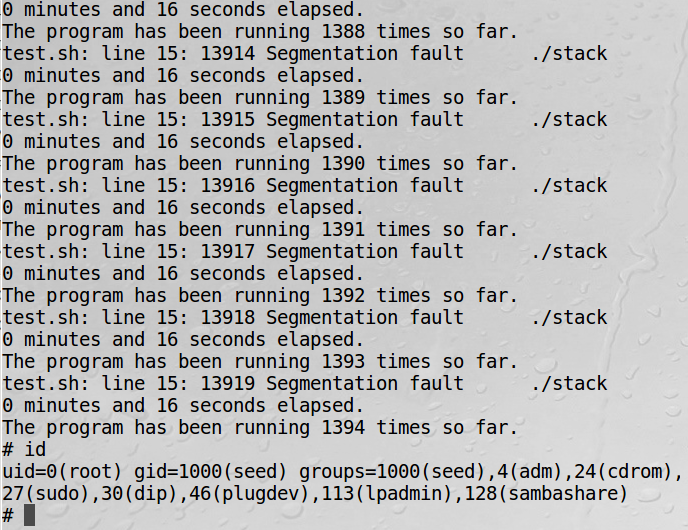
打开地址随机化，重新运行exploit.py，发现攻击失败：



新建一个脚本文件把脚本放进：



脚本停止运行，攻击成功，获得了root权限：



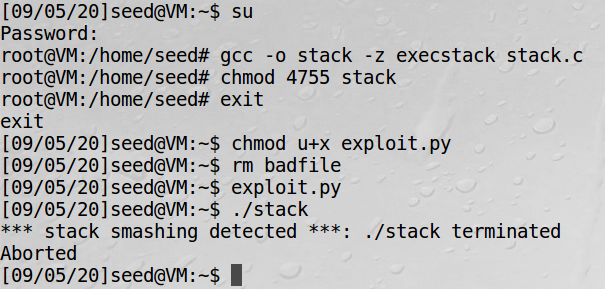
**7、Task 5: Turn on the StackGuard Protection（打开StackGuard保护）**

在执行此任务之前，请记住首先关闭地址随机化，否则将不知道哪种保护有助于实现保护。

在之前的任务中，在编译程序时禁用了GCC中的StackGuard保护机制。在此任务中，可以考虑在StackGuard存在的情况下重复执行任务2。为此，应在不使用-fno-stack-protector选项的情况下编译程序。对于此任务，将重新编译易受攻击的程序stack.c以使用GCC StackGuard，再次执行任务1，并报告观察结果，可能会报告发现的任何错误消息。

在GCC版本4.3.3和更高版本中，默认情况下启用StackGuard。因此，必须使用前面提到的开关禁用StackGuard。在早期版本中，默认情况下禁用它。如果使用较旧的GCC版本，则可能不必禁用StackGuard。

如图所示，使用StackGuard保护编译stack.c，攻击失败：



**8、Task 6: Turn on the Non-executable Stack Protection（开启不可执行的堆栈保护）**

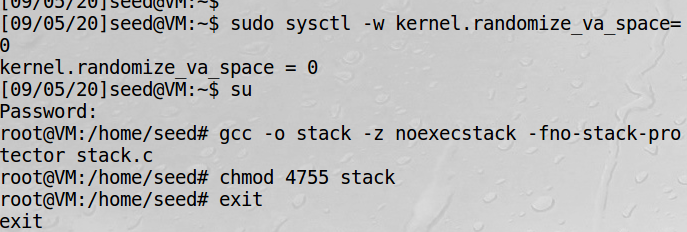
在执行此任务之前，请记住首先关闭地址随机化，否则不知道哪种保护有助于实现保护。

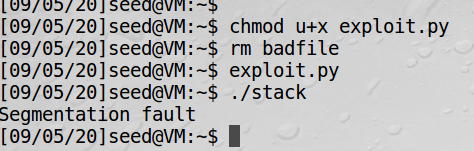
在之前的任务中，有意使堆栈可执行。在此任务中使用noexecstack选项重新编译易受攻击的程序，并重复执行任务2中的攻击。可以获得shell程序吗？如果没有，那是什么问题？这种保护方案如何使攻击变得困难。在实验报告中描述观察和解释。可以按照以下说明打开不可执行的堆栈保护。

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

应当注意，不可执行的堆栈仅使不可能在堆栈上运行shellcode，但不能防止缓冲区溢出攻击，因为在利用缓冲区溢出漏洞之后，还有其他方法可以运行恶意代码。返回libc攻击就是一个例子。针对该攻击设计了一个单独的实验。

如图所示，关闭地址随机化，用noexecstack命令编译，攻击失败没有root权限：





**二、Return-to-libc Attack Lab**

**The BUF SIZE value for this lab is: 50**

**1、Turning Off Countermeasures**

（同一、Buffer Overflow Vulnerability Lab）

**2、The Vulnerable Program（漏洞程序）**

上面的程序有一个缓冲区溢出漏洞。它首先从一个名为badfile的文件中读取一个大小为300字节的输入到BUF SIZE大小小于300的缓冲区中。由于函数fread（）不会检查缓冲区边界，因此会发生缓冲区溢出。该程序是拥有的root Set-UID程序，因此，如果普通用户可以利用此缓冲区溢出漏洞，则该用户可能能够获得root shell。应当注意，该程序从用户提供的名为badfile的文件中获取输入。因此，可以以一种方式构造文件，以便当易受攻击的程序将文件内容复制到其缓冲区中时，可以生成root shell。

编译代码，然后将其变成根拥有的Set-UID程序。不要忘了包括-fno-stack-protector选项（用于关闭StackGuard保护）和“-z noexecstack”选项（用于打开不可执行的堆栈保护）。还应注意的是，更改所有权必须在打开Set-UID位之前完成，因为所有权更改会导致Set-UID位关闭。

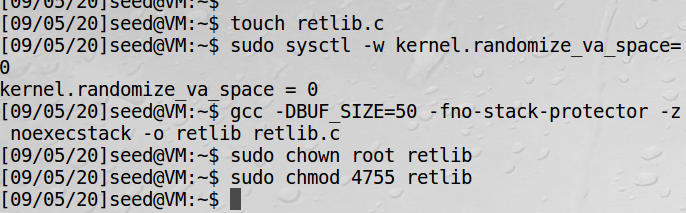
// Note: N should be replaced by the value set by the instructor

$ gcc -DBUF\_SIZE=N -fno-stack-protector -z noexecstack -o retlib retlib.c

$ sudo chown root retlib

$ sudo chmod 4755 retlib

编译漏洞代码并将它设置为root所有的Set-UID程序：



**3、Task 1: Finding out the addresses of libc functions（找出libc函数的地址）**

在Linux中，当程序运行时，libc库将被加载到内存中。当关闭内存地址随机化时，对于同一程序，该库始终加载在同一内存地址中（对于不同的程序，libc库的内存地址可能不同）。因此，可以使用调试工具（例如gdb）轻松找到system（）的地址，即可以调试目标程序retlib。即使该程序是root拥有的Set-UID程序，仍然可以对其进行调试，只不过该特权将被删除（即有效用户ID与真实用户ID相同）。在gdb中，键入run命令以一次执行目标程序，否则，将不会加载库代码。使用p命令（或打印）来打印出system（）和exit（）函数的地址（稍后将需要exit（））。

$ touch badfile

$ gdb -q retlib ➙Use "Quiet" mode

Reading symbols from stack...(no debugging symbols found)...done.

gdb-peda$ run

......

gdb-peda$ p system

$1 = {<text variable, no debug info>} 0xb7e42da0 <\_\_libc\_system>

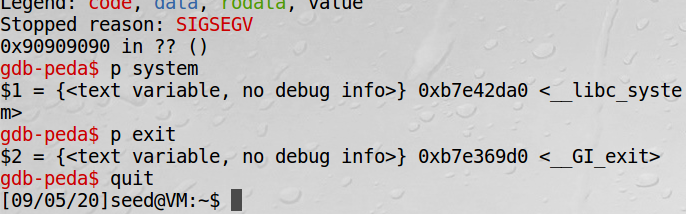
gdb-peda$ p exit

$2 = {<text variable, no debug info>} 0xb7e369d0 <\_\_GI\_exit>

gdb-peda$ quit

应当注意，即使对于同一程序，如果将其从Set-UID程序更改为非Set-UID程序，libc库也可能不会加载到同一位置。因此，调试程序时需要调试目标Set-UID程序；否则得到的地址可能不正确。

使用gdb找到system和exit的地址：



**4、Task 2: Putting the shell string in the memory**

攻击策略是跳转到system（）函数，并使其执行任意命令。由于希望得到一个shell提示，因此希望system（）函数执行“/bin/sh”程序。因此，必须先将命令字符串“/bin/sh”放入内存中，并且必须知道其地址（该地址需要传递给system（）函数）。有许多方法可以实现这些目标，可以选择一种使用环境变量的方法。

当从shell提示符下执行程序时，shell实际上会产生一个子进程来执行该程序，并且所有导出的shell变量都将成为该子进程的环境变量。这提供了一种简单的方法，可以在子进程的内存中放入任意字符串。定义一个新的外壳变量MYSHELL，并使其包含字符串“/bin/sh”。通过以下命令，可以验证字符串是否进入子进程，并由运行在子进程内部的env命令将其打印出来。

$ export MYSHELL=/bin/sh

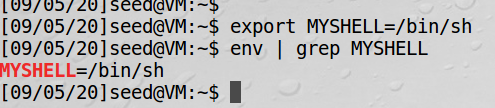
$ env | grep MYSHELL

MYSHELL=/bin/sh

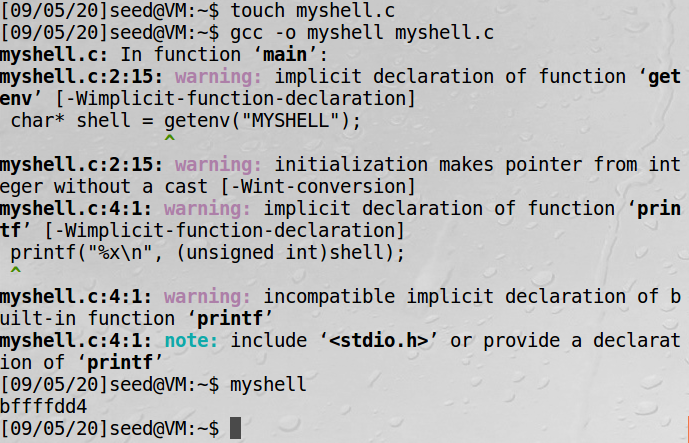
将使用此变量的地址作为system（）调用的参数。使用以下程序可以轻松找到此变量在内存中的位置。

如果关闭了地址随机化，则会发现打印出了相同的地址。但是，当运行易受攻击的程序retlib时，环境变量的地址可能与通过运行上述程序所获得的地址不完全相同。当更改程序名称时，这样的地址甚至可以更改（文件名中的字符数有所不同）。好消息是，外壳程序的地址将与您使用上述程序打印出的内容非常接近。因此，可能需要尝试几次才能成功。

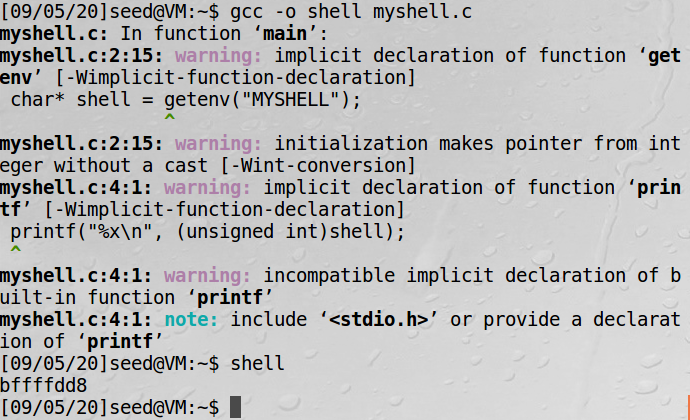
设置环境变量：



运行代码，选择生成的文件为myshell时的地址：



重新运行一次，生成文件名为shell时的地址：



结论：环境变量的地址可能与通过运行上述程序所获得的地址不完全相同。

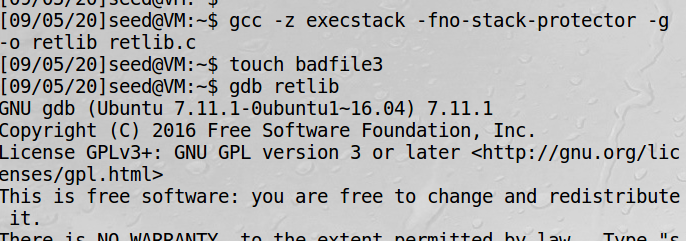
**5、Task 3: Exploiting the buffer-overflow vulnerability（利用缓冲区溢出漏洞）**

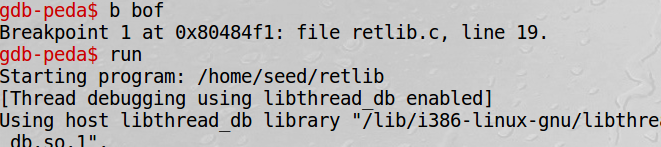
创建badfile的内容。由于内容涉及一些二进制数据（例如libc函数的地址），因此可以使用C或Python进行构造。

攻击变化1：是否确实需要exit（）函数？请尝试攻击，但不要在badfile中包含此函数的地址。再次发起攻击，报告并解释观察结果。

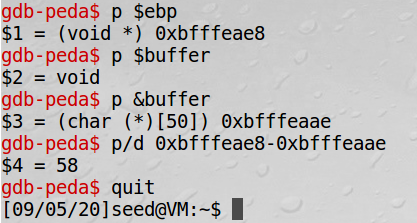
攻击变型2：攻击成功后，将retlib的文件名更改为其他名称，并确保新文件名的长度不同。例如，可以将其更改为newretlib。重复攻击（不更改badfile的内容）。您的攻击成功与否？ 如果没有成功，请说明原因。

新建badfile文件，运行gdb命令：

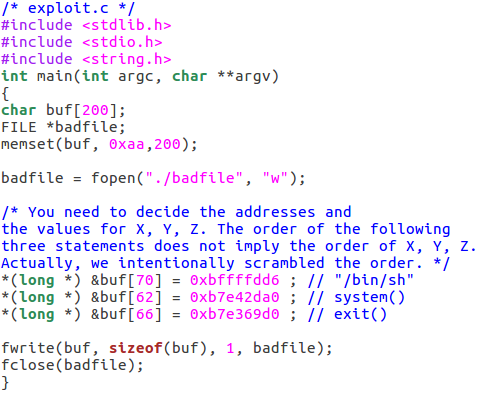




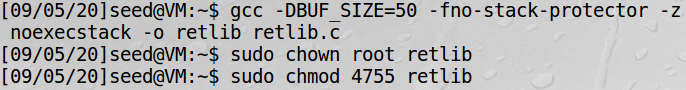
从图中可以看出offset的值为58+4，所以创建badfile的代码中X，Y，Z应该分别为：62，66，70。



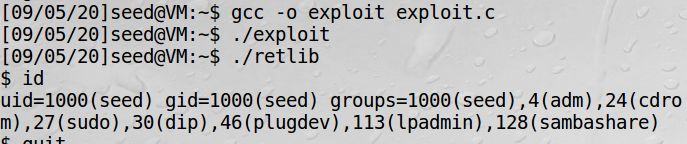
根据offset的值和最开始的system地址、exit地址，修改exploit.c文件：



（前面有小失误所以重新运行了retlib）

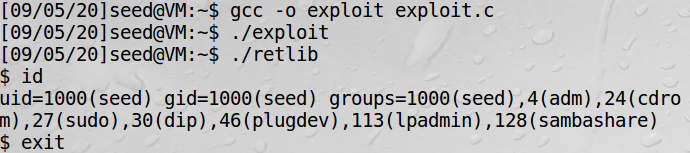


编译代码并执行，可以看到攻击成功：



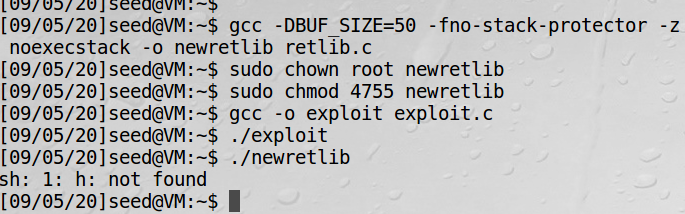
攻击变化1：是否确实需要exit（）函数？请尝试攻击，但不要在badfile中包含此函数的地址。再次发起攻击，报告并解释观察结果。

如图所示，注释掉exit的地址后仍然能成功攻击（但我并不清楚badfile中是否包含此函数的地址）



攻击变型2：攻击成功后，将retlib的文件名更改为其他名称，并确保新文件名的长度不同。例如，可以将其更改为newretlib。重复攻击（不更改badfile的内容）。您的攻击成功与否？ 如果没有成功，请说明原因。

如图，重新生成文件名为newretlib的文件，攻击失败，可能是因为文件名变换导致的地址变化。



**6、Task 4: Turning on address randomization（打开地址随机化）**

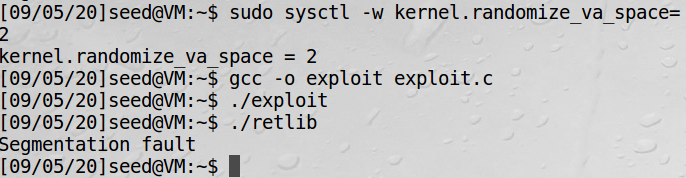
在此任务中，打开Ubuntu的地址随机保护，并查看该保护是否对“返回libc”攻击有效。首先，打开地址随机化：

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

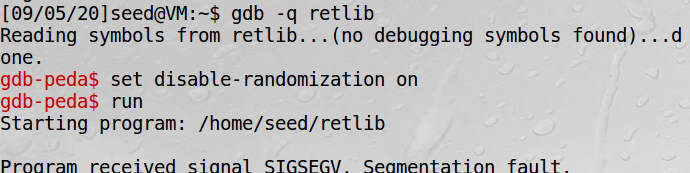
请运行上一个任务中使用的相同攻击。能成功吗？请描述观察并提出假设。在构造badfile的漏洞利用程序中，需要提供三个地址以及X，Y和Z的值。如果打开了地址随机化，则这六个值中的哪个是不正确的，请在报告中提供证据。

如果打算使用gdb进行调查，则应注意，默认情况下gdb禁用调试进程的地址空间随机化，无论是否在基础操作系统中打开了地址随机化。在gdb调试器中，可以运行“show disable-randomization”以查看随机化是否已关闭。可以使用“set disable-randomization on”和“set disable-randomization off”来更改设置。

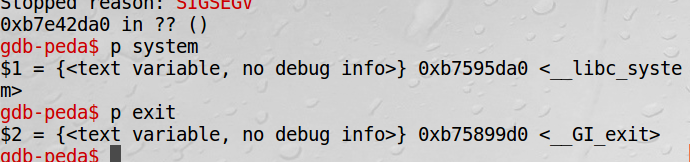
打开地址随机化，重新编译exploit，发现攻击失败：



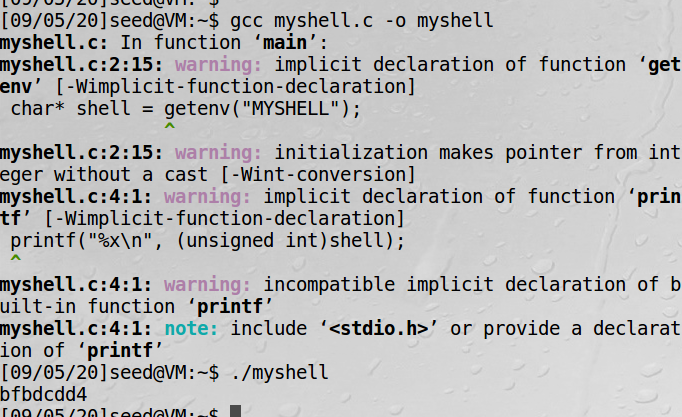
用gdp查询system和exit的地址：



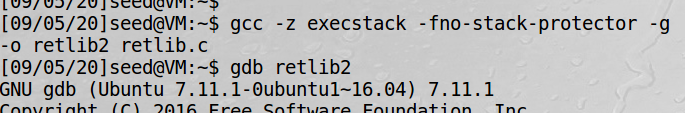
发现都发生了变换：

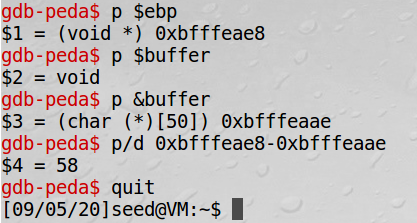


重新运行myshell，发现环境变量的地址未发生变化：



同样重新验证retlib，offset的值仍然是58，X，Y，Z的值也并未发生变化：





结论：system和exit的地址发生了变化，其他数据都没有变，攻击失败的原因应该在此。

**7、Task 5: Defeat Shell’s countermeasure**

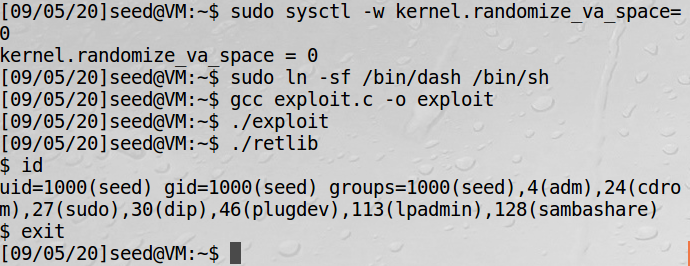
此任务的目的是在启用Shell的对策后发起“返回libc”攻击。在执行任务1至4的攻击之前，将/bin/sh重新链接到/bin/zsh，而不是/bin/dash（原始设置）。这是因为某些shell程序（例如dash和bash）具有一种对策，当它们在Set-UID进程中执行时会自动放弃特权。在此任务中，想克服这种对策，即，即使/bin/sh仍指向/bin/dash，也希望获得root shell。首先将符号链接更改回：

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

当/bin/sh指向/bin/dash时，不能直接返回system（）函数，因为system（）实际上使用/bin/sh来执行命令，并且/bin/dash将放弃特权。有很多方法可以解决此问题。一种方法是返回不依赖于/bin/sh的其他函数。另一种方法是在调用system（）之前调用setuid（0）。setuid（0）调用将真实用户ID和有效用户ID都设置为0，从而将进程变成非Set-UID one（它仍然具有root特权）。事实证明，使用返回libc技术很难做到这一点。

存在两个主要挑战：（1）如何将多个函数（带有参数）链接在一起，以及（2）如何将零作为参数传递而在恶意输入中不包含零？在这项任务中，专注应对拳头挑战。可以忽略第二个挑战，并在输入中输入零。在易受攻击的程序中，有意使用了fread（），它不同于strcpy（），不受零影响。

如图所示，关闭地址随机化，/bin/sh指向/bin/dash，再运行exploit.c，发现攻击成功可以拿到root权限：



**8、Task 6: Defeat Shell’s countermeasure without putting zeros in input**

在此任务中，解决任务5中的第二个挑战，即不允许在输入（坏文件）中输入任何零（二进制零）。在现实世界的攻击中，将数据复制到缓冲区通常使用诸如strcpy（）之类的函数，当遇到零时，它将终止复制。为了模拟现实情况，添加了此约束。

规避此限制的主要思想是首先在setuid（）函数获取其参数的位置放置一个非零值，但是在调用setuid（）之前，调用一系列函数，例如sprintf（）来将非零值更改为零。之后，我们调用setuid（），但是现在它的参数已经为零。基本上，我们首先将有效负载放在堆栈上（不带零），然后使用return-to-libc技术自我修改放置在堆栈上的数据。

为了实现此目标，需要能够将一系列功能链接在一起，其中一些功能具有多个参数。任务3和5中使用的基本的“返回libc”技术对函数及其参数的数量有限制。需要一种更通用的技术，称为返回定向编程（ROP），它允许将任意数量的函数（带有或不带有参数）链接在一起。在任务3和5中进行的返回libc攻击只是ROP的特例。

**三、总结**

这两章主要进行的是栈溢出和返回lib的实验，都有一定难度，在实现攻击的过程中被卡了很多次，地址值经常算不对又不知道哪里出了错。有时候还经常粗心大意把命令和文件名输入错误。好在还是完成了，对攻击的配置环境也有了更多的了解，对栈溢出和返回lib的原理也了解得更详细了。希望下一次的实验有进步。