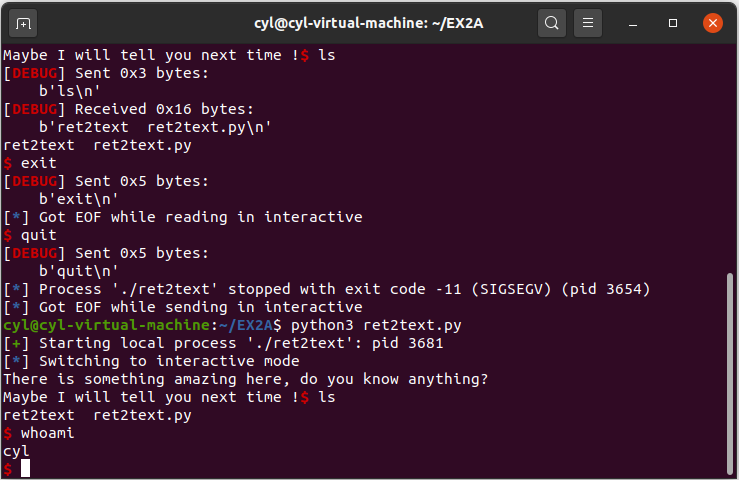
**《基于栈溢出的ROP利用》实验报告**

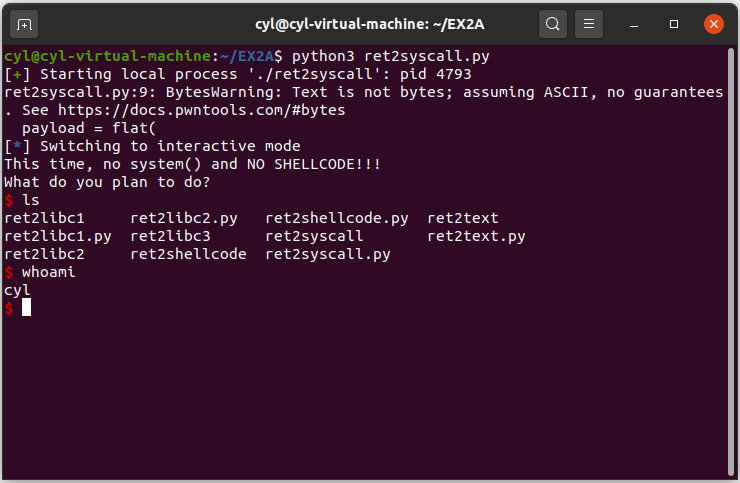
蔡雨良

1. 基本ROP复现
2. ret2text

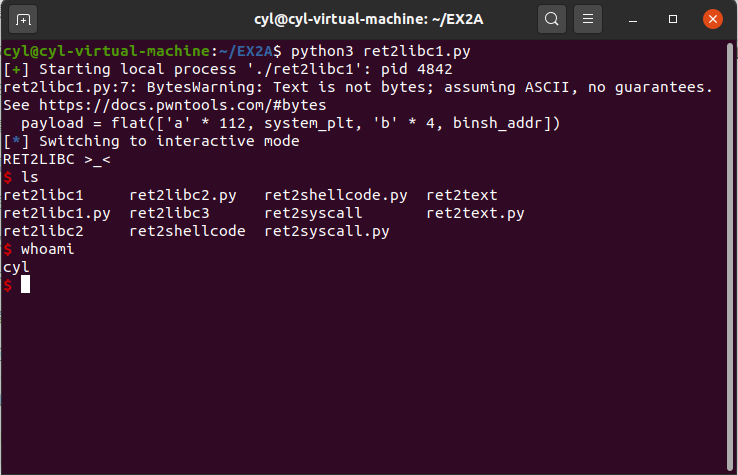
直接覆盖返回地址到原system(‘/bin/sh/)代码段即可。

1. ret2syscall

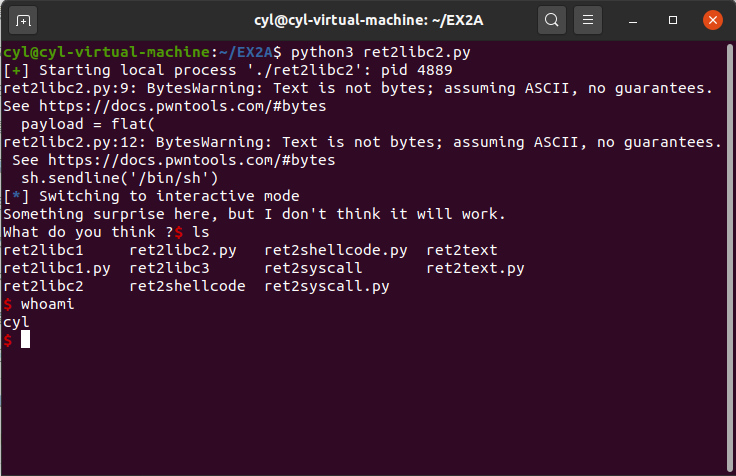
将寄存器的值覆盖为符合system系统调用的值，结合代码段中/bin/sh构造system指令。

1. ret2libc1

构造system的栈帧，使得其参数地址为’/bin/sh’的地址，直接完成system指令。

4.ret2libc2

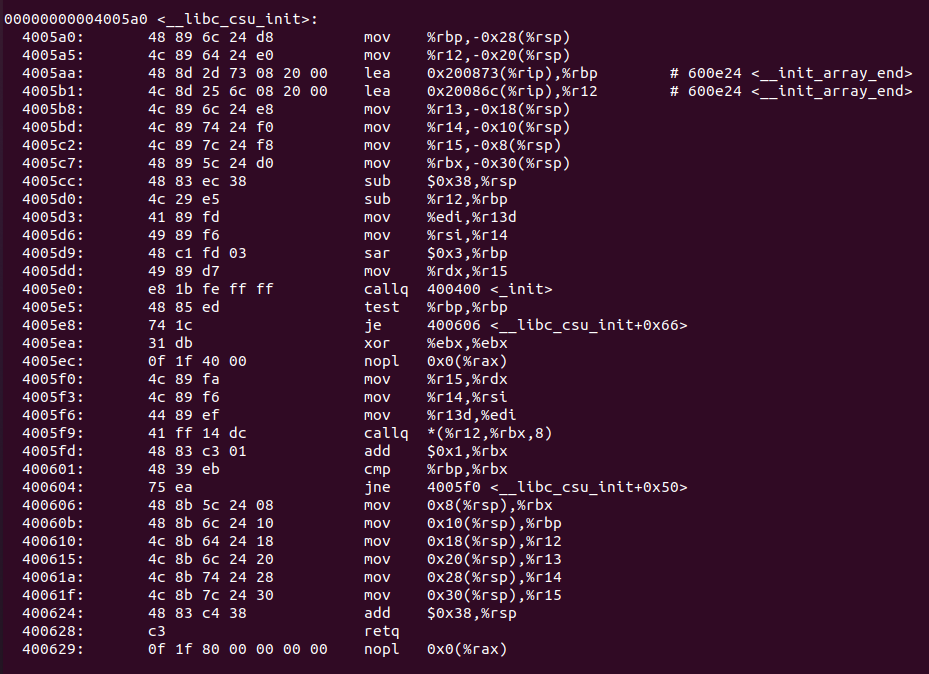
通过gets函数将’bin/sh/’写入程序内存空间并且保持栈平衡，再执行system()



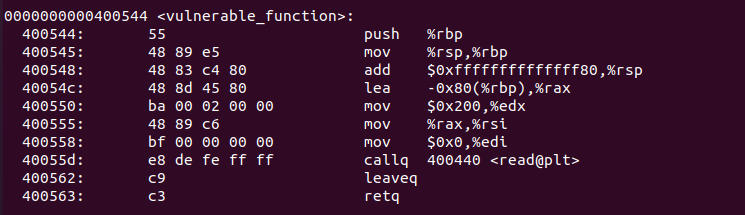
1. 中级ROP：ret2csu
2. 思路

在 64 位程序中，函数的前 6 个参数是通过寄存器传递的，但是大多数时候，我们很难找到每一个寄存器对应的 gadgets。 这时候，我们可以利用 x64 下的\_\_libc\_csu\_init中的gadgets。这个函数是用来对 libc 进行初始化操作的，而一般的程序都会调用 libc 函数，所以这个函数一定会存在。

以下是我的环境中\_\_libc\_csu\_init函数对应的汇编代码:



其中，使用的gadgets段为4005f0开始的mov %r15,%rdx到400628结束的retq。

然后是程序的代码段：

看起来并没有与system(‘bin/sh’)相关的代码段供我们利用，所以我们需要手动构造之。

所以我们需要得知系统中write()函数的地址以及system的地址以便写入system与bin/sh。

使用execve函数也可得到相同的效果。

所以：大致思路如下：

1. 先得知write与execve的地址
2. 将execve的地址与bin/sh写入程序内存。
3. 执行execve(bin/sh)

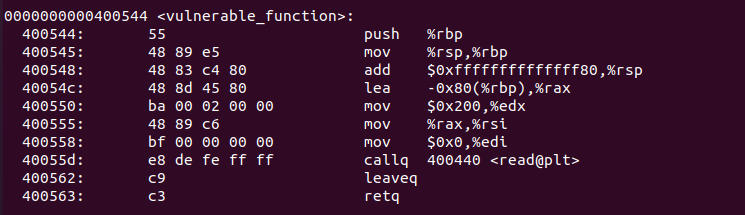
所以我们需要构造3次payload以使程序执行我们写入的函数。

1. payload构造（解题的关键之处）

前128个字符无关紧要，随意填充即可。

关键是我们在之后（也就是程序的rbp开始）写入的数据。我们需要利用\_\_libc\_csu\_init中的gadgets将寄存器赋予对应的值，同时结合原函数（vulnerable function）使得当该函数中retq指令执行时返回到我们想要的地址，这里由于我们要重复利用该函数，所以返回地址为main函数的基址。

首先观察原函数，发现read()调用之后分别执行的leaveq以及retq指令。



二者作用如下：

而pop (出栈)指令将栈顶的元素弹出栈，然后栈指针rsp + 8。

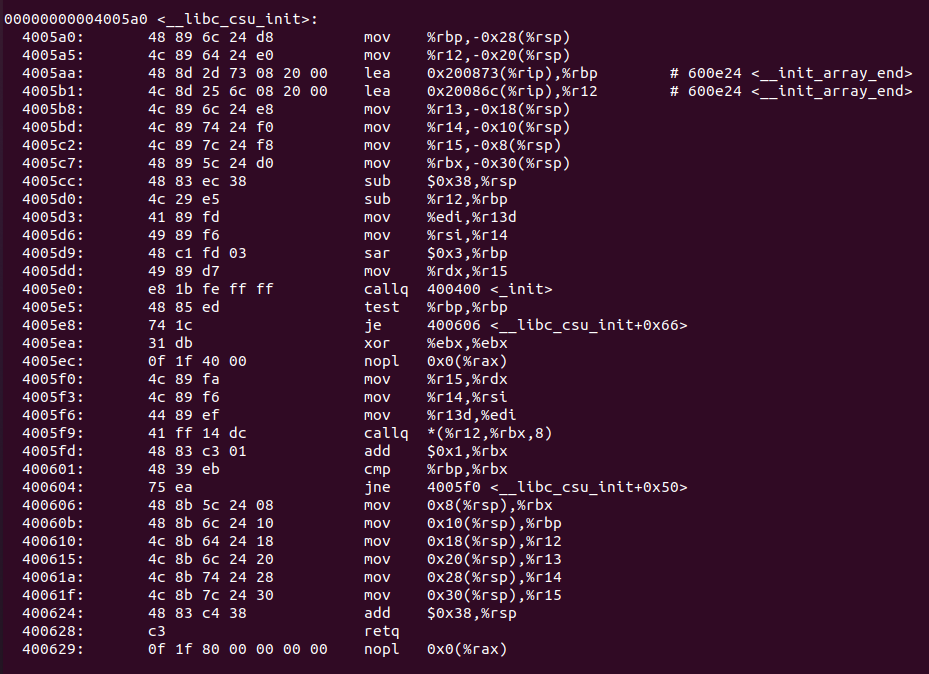
所以在read()执行完毕直到retq函数执行后共有两次pop指令：

第一次pop指令无关紧要，所以第一次pop的字节也可以随意填充（这也是我直接填充了128+8=136个无效字符的原因）

而第二次pop会将弹出的元素传到rip里，这时我们就应该控制rip里的内容使得程序转到\_\_libc\_csu\_init中的gadgets执行。

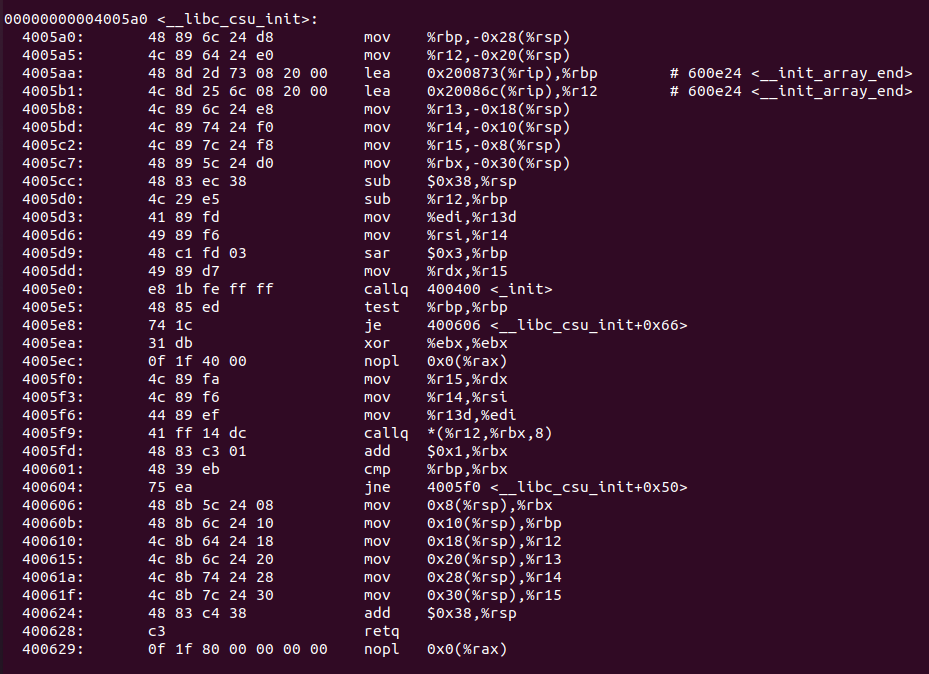
应该转移到哪里呢？通过对gadegets分析，我们可以首先转移到400606（mov 0x8(%rsp),%rbx）

指令并改变6个寄存器的值。接下来，我们跳转至4005f0（mov %r15,%rdx），这使得我们可以改变更多寄存器的值，并且能够执行call()系统调用，之后指令继续按顺序执行（注意我们不让jne跳转）至retq指令，然后转移到main函数使得我们可以继续构造payload，反复按照以上操作让程序执行我们想执行的指令进行攻击。如下图所示：



**第一次**

**第一次转移**



**第二次**

**转移至main()函数的基址**

**JNE不能跳转(因为我们需要RETQ)**

**重要指令**

而对各寄存器的赋值，我们需要结合gadgets与自身需求：

rbx:0。只有这样call系统调用才会有效。

rbp:1(rbx+1)。注意到callq后面的add、cmp与jne指令，只有让rbp与rbx+1相等，jne指令才不会跳转。

r12:我们想要执行的指令的基址。

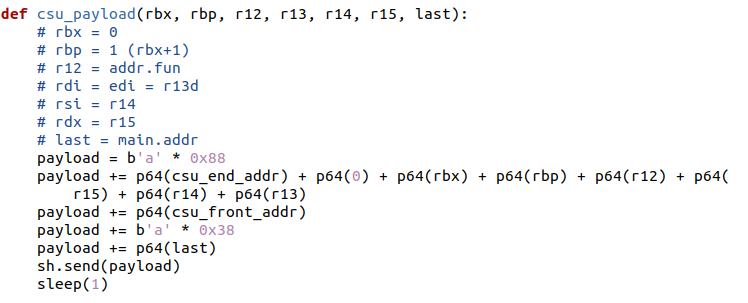
r13:rdi,edi。根据我们想执行的指令传对应参数。

r14:rsi。根据我们想执行的指令传对应参数。

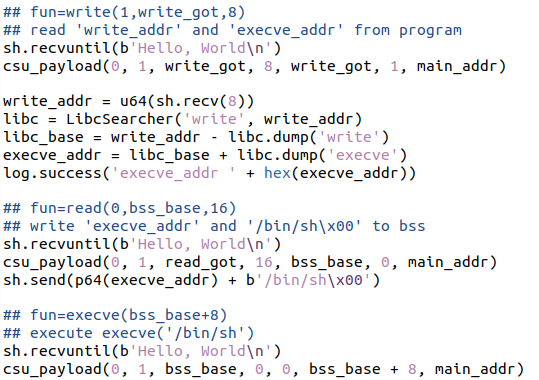
r15:rdx。根据我们想执行的指令传对应参数。

之后，如何继续填充数据以至于能让程序正确返回到main函数里呢？注意到400624（add 0x38,rsp）指令在retq指令前，所以我们应该填充0x38(56)位的无效字符，之后再填充main函数的基址，这样到执行retq指令之后，栈顶指向的要弹出给rip元素就是main函数的基址，使得程序再一次转到main函数且自身没有发觉异常。

Payload代码构造如下：

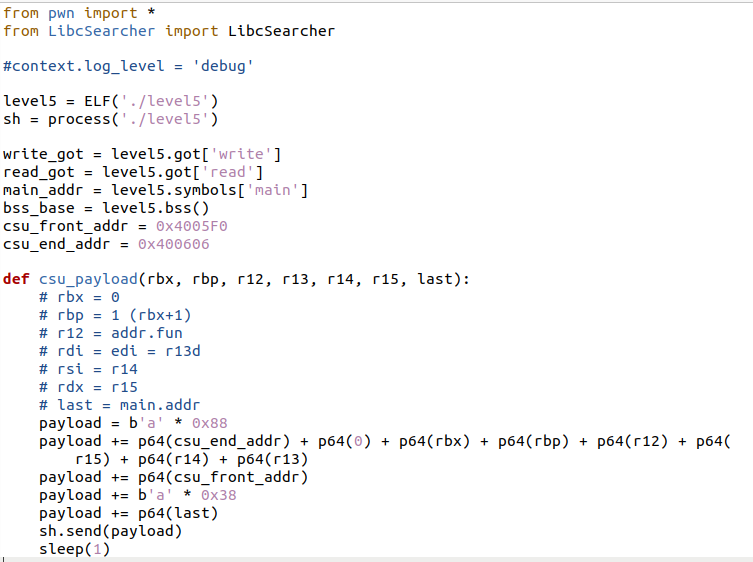


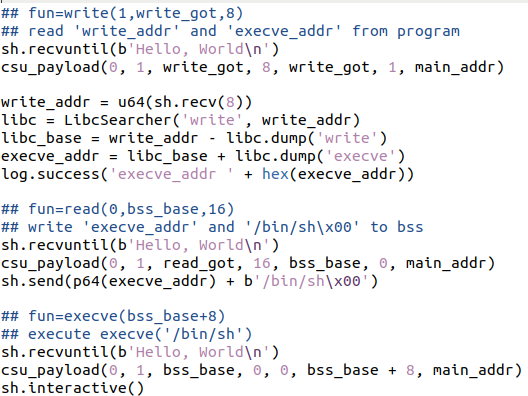
上述payload我们需要构造3次，每一次都大同小异：



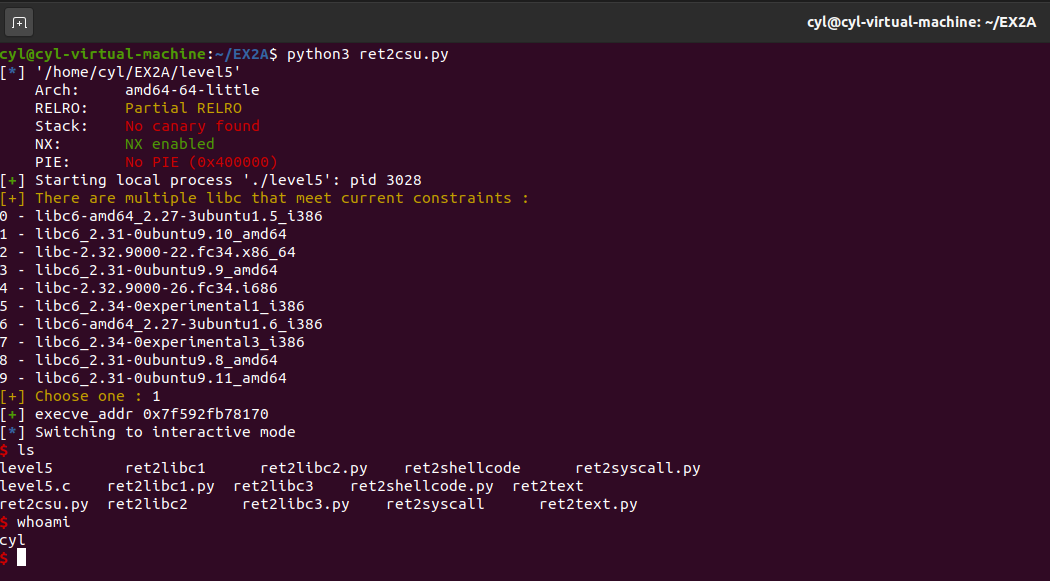
最后，执行sh.interactive()检查是否成功夺取控制权。

总的代码如下：





实际利用结果如下：（play就是该过程（ret2csu）的演示）：



1. 总结与感想

通过这次实验，我对于缓冲区溢出的攻击有了更深刻地理解：从保护方案到使用ROP利用程序中的gadgets绕过NX防护展开攻击，再到汇编知识的运用以解决ret2csu问题，这其中不乏难以下手的挑战性，以及精巧设计数据以达到目的的趣味性和成就感。再次感谢老师及助教提供了这个学习的机会，让我能在网络攻防方面的理解更进一步，体会到网络攻防的趣味！

1. GitHub

链接：https://github.com/cyl9000/EX2A