参考书籍：

* 1. 《0day安全：软件漏洞分析技术》2011，基本是windows平台，基础全面技术也相对新颖。
  2. 《网络渗透技术》2005，老旧，很多环境已经不一样，包含很多平台（win32、linux x86/x64、AIX PowerPC、Solaris SPARC）。
  3. 《Exploit编写系列教程》<http://bbs.pediy.com/showthread.php?t=123602>具体的编写流程，其中一些极限shellcode编写可能会在CTF中涉及。

工具：

Pwntools(python第三方库)和peda(peda@github)

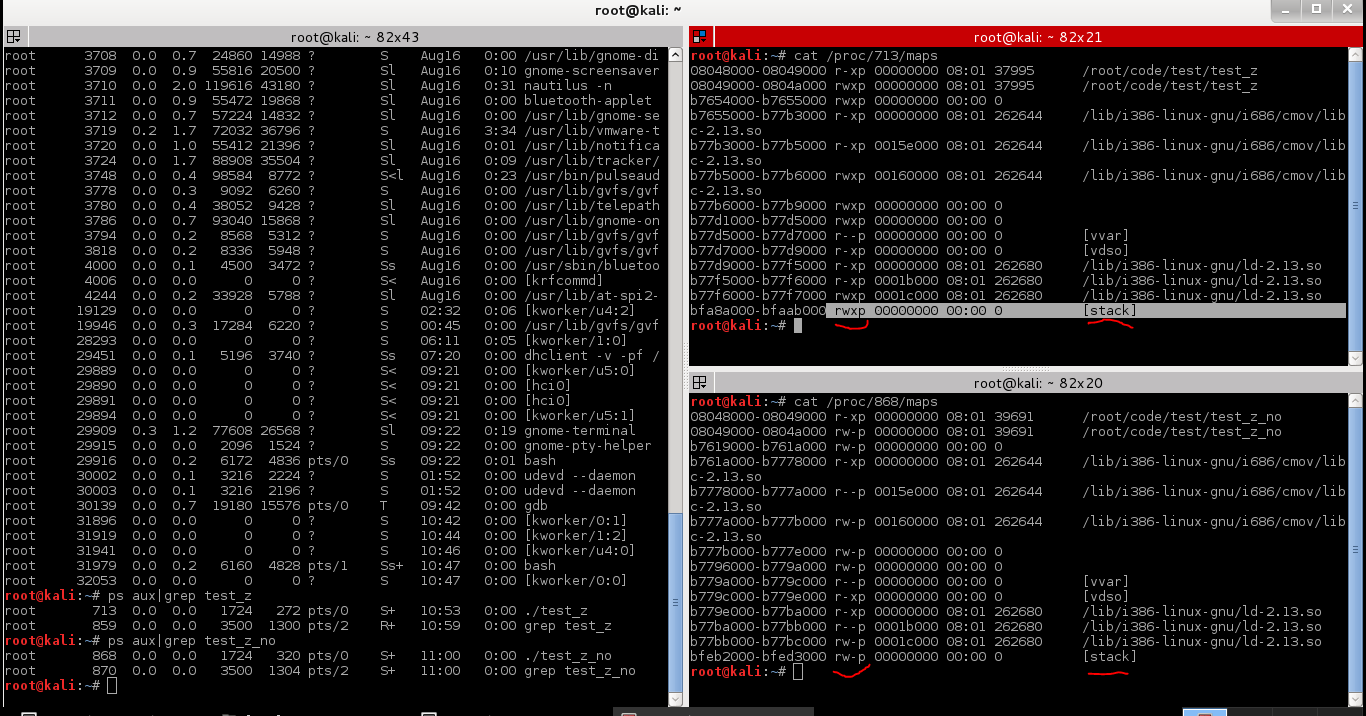
编译选项：

-fno-stack-protector 关闭stack protector。

-fstack-protector 启动堆栈保护，不过只为局部变量中含有char数组的函数插入保护代码。

-fstack-protector-all启动堆栈保护，为所有函数插入保护代码。

-z execstack 关闭NX，使堆栈可执行。Cat /proc/pid/maps查看线程（包括线程）的一些信息。可以看到关闭NX时，堆栈可执行，正常编译不可执行。



$strip test\_strip 去除文件中函数名的相关信息。

系统配置ASLR(Address space layout randomization)

查看：$sysctl –n kernel.randomize\_va\_space

设置：$sysctl –w kernel.randomize\_va\_space+0/1/2

或：

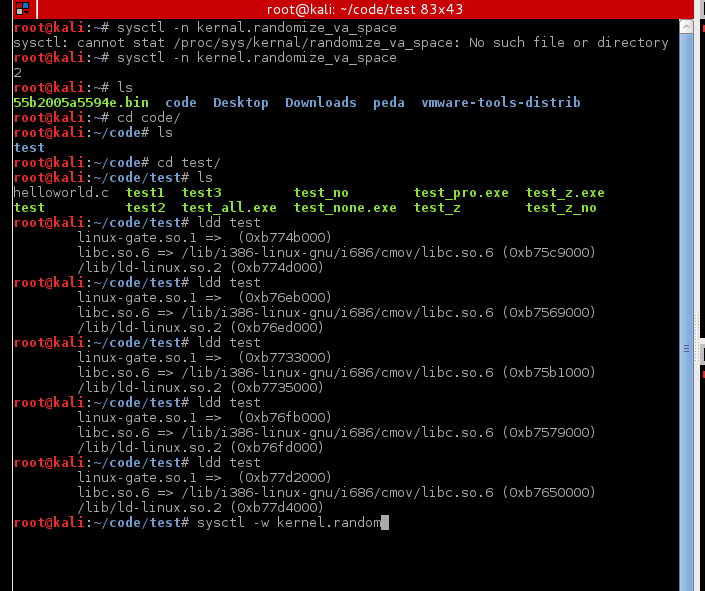
$echo 0/1/2 > /proc/sys/kernel/randomize\_va\_space

0 关闭aslr

1 mmap base、stack、vdso page将随机化

2 在1的基础上heap随机化

$ldd <elf> 多次，查看是否开启aslr。



<elf

Socat

监听、接受port上传来的参数，然后运行后面的命令。

socat TCP4-LISTEN:<port>,fork EXEC:<elf>

edb

类似于OD的Linux下的工具。Linux 和Windows下代码段地址一般相同。

* 1. 调试流程：
     1. 加载可执行程序
     2. 设置symbol directory;
     3. 检查breakpoint manager；
     4. 按F9跳到main入口。
  2. 添加argument参数，先在options->application arguments中添加参数，然后重新打开<elf>
  3. Symbol Directory:Options->Directories->Symbol Directory->设置为源文件位置。
  4. 寻找跳板指令，因为各平台中地址可能会随机变化（如开启aslr），所以可以利用跳板指令软着陆，跳到shellcode。

Plugins->OpcodeSearcher->寻找需要的跳转

pattern.py

黑盒测试工具，生成溢出输入的辅助工具。

pattern.py create <length>，生成指定长度的输入字符串。

pattern.py offset <ox xxxxx>，根据地址的值返回在pattern中的偏移量。

<A~Z><a~z><0~9>

x86中操作EIP的指令

* 1. Jxx etc 比较转移和跳转指令
  2. Call指令的分解：
     1. Push <eip+5>
     2. Jmp <eip+5>+<xxxxxxx>

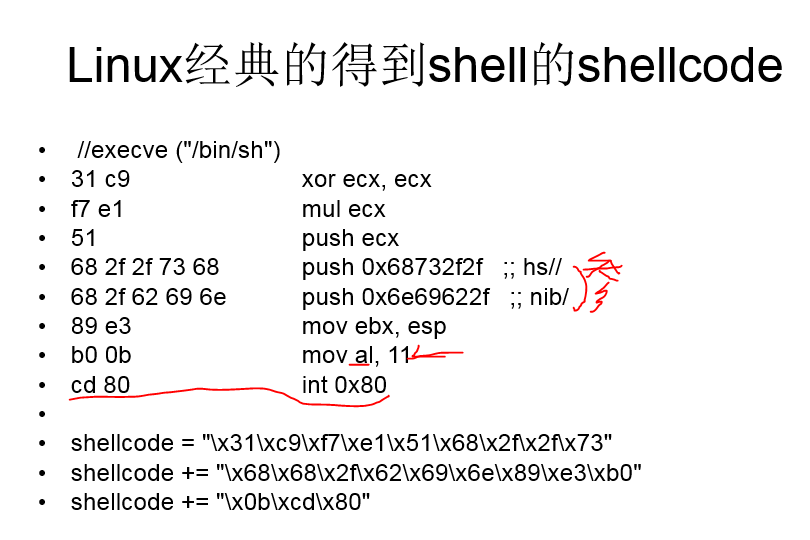
Gdb，Linux下命令行汇编调试工具。

* 1. 工作流程：
     1. 启动程序，如gdb ./exp2
     2. 设置断点：两种方式
        1. 在symbol没去掉时，b main
        2. 去掉symbol，b 0x地址
     3. Run
     4. 其他调试命令
  2. 找jmpcall，寻找适当的跳转语句，为跳转shellcode做准备

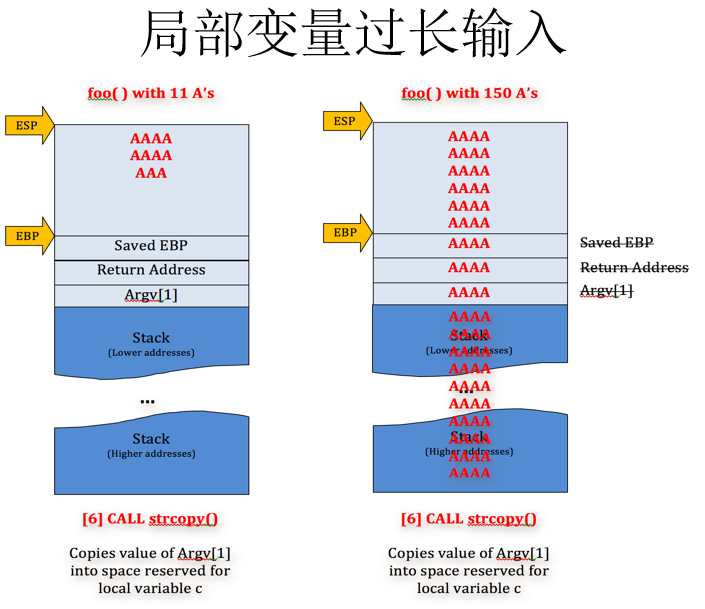
int 80

int 80是Linux平台下的终端向量表，比windows下终端调用简单的多。先push进中断所需参数，然后al设置为中断号，最后调用int 80中断即可。

Linux中断向量表：http://asm.sourceforge.net/syscall.html



溢出攻击原理图



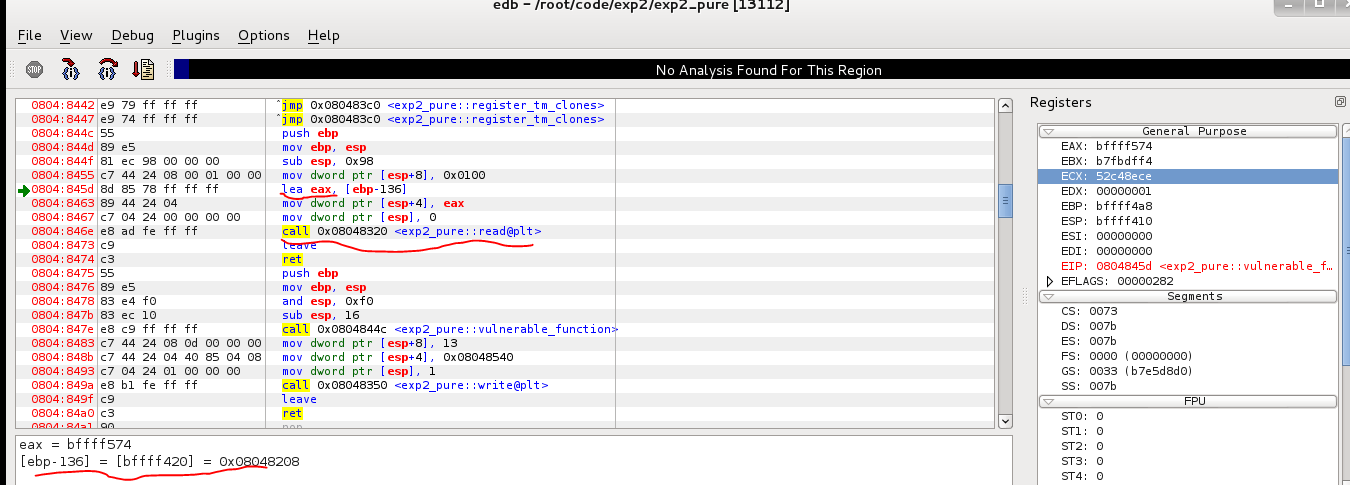
溢出实验基本步骤

1. 基本原理：

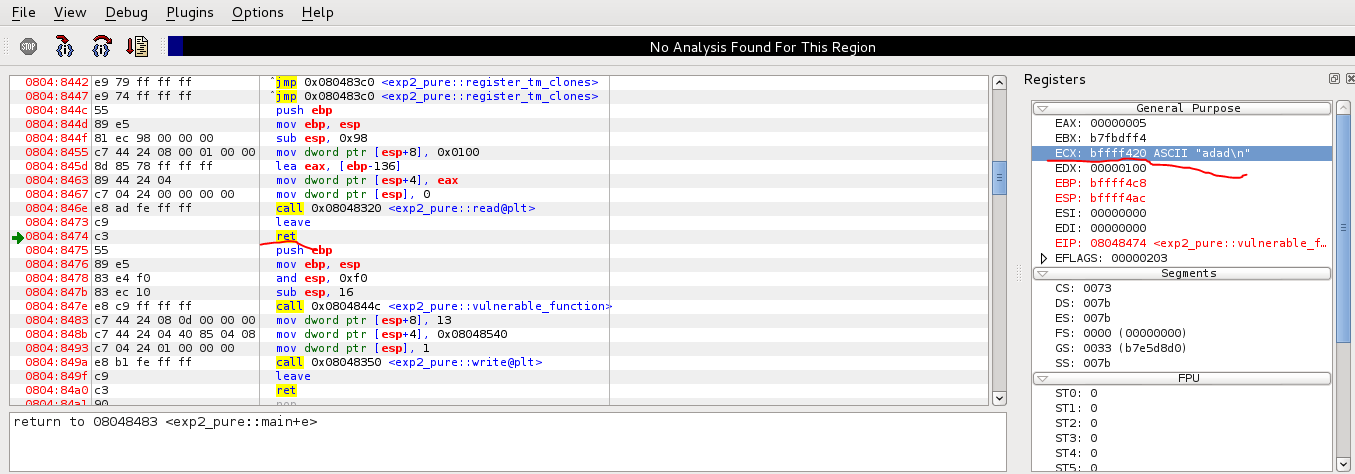
在函数调用的return处设置一个跳板地址，通过跳板地址跳到shellcode处执行shellcode。所以要完成以下主要工作：

* + 1. 确定偏移量，找到输入数据的内存地址，以及和return语句的偏移。
    2. 寻找合适的寄存器，寄存器内地址值要求和输入数据尽量靠近。
    3. 根据寄存器寻找合适的跳板地址。

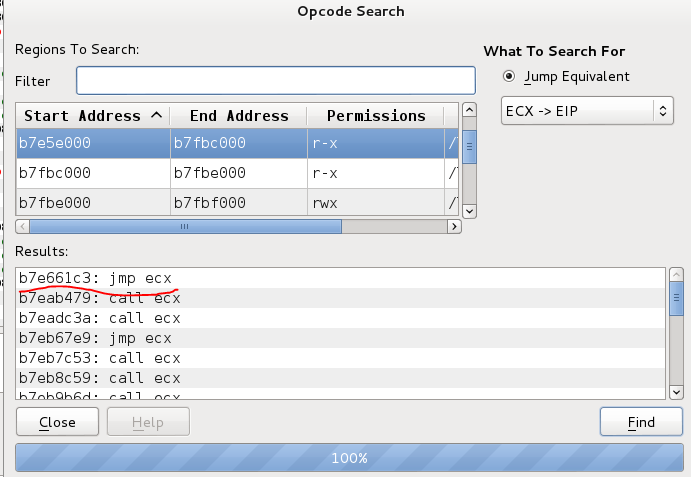
1. 流程
   * 1. 在edb上运行最终版本的exp2\_pure，定位到输入位置，找到输入数据的内存地址。输入地址和ebp相差136，return语句是ebp下一句，所以输入地址和return语句的偏移量是140。



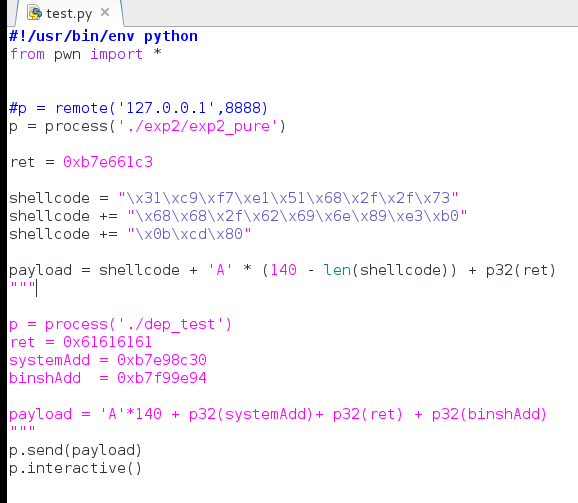
* + 1. 运行到return语句，寻找合适的寄存器，要求寄存器中的内存地址和输入数据地址尽量接近。发现正好ecx正好指向输入数据的起始地址。



* + 1. 寻找合适的跳板地址。寻找jmp ecx的命令，尽量在导入的库中寻找，因为地址相对固定。



* + 1. 修改python溢出输入程序代码，设置ret，偏移和shellcode位置。



* + 1. Server端运行socat TCP4-LISTEN: <port>,fork EXEC:<ELF>
    2. 运行python溢出输入程序。成功获得shell。

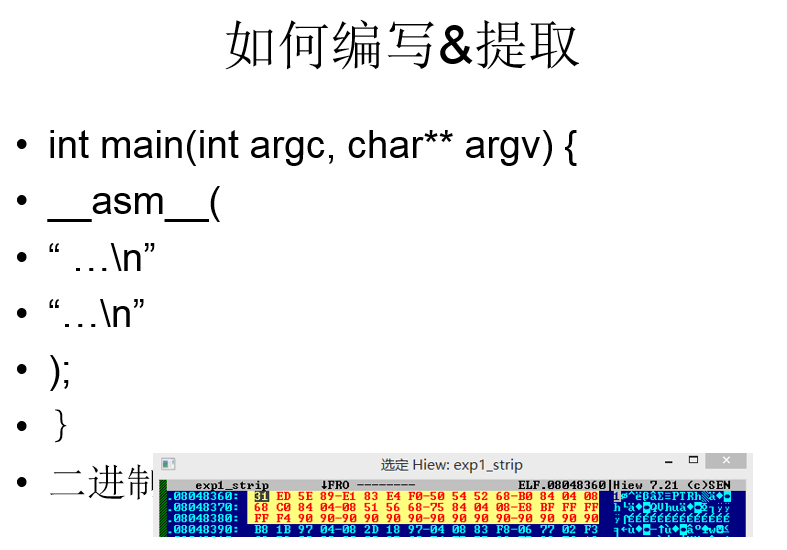
1. 其他
   * 1. 因为本程序中使用read()函数获得输入，所以ecx寄存器指向输入数据内存地址。是否有寄存器可用，哪个寄存器可用和输入函数有关，scanf()基本寄存器都不可用，需要esp或ebp。Scanf()和pwnkiller.py配合send(payload)有问题。
     2. 前期调试时，可以先运行python溢出输入程序，在send()数据前设置暂停，然后将最新的程序在edb中打开，file->attach到python程序，这样将以python的输入为输入数据，传入edb中程序运行。

调试方法：

* + - 1. test.py脚本的 process函数后 叫如 raw\_input("xxxx")
      2. $python test.py 启动脚本
      3. 在EDB中attach test.py中启动的进程
      4. 在程序溢出函数输入函数（read）后一条指令 下断点
      5. 在test.py运行窗口 输入任意字符
      6. 在edb中F9运行到设置的断点
      7. 就可以调试具体的溢出流程了~
  1. 利用esp等溢出
  2. 利用system(‘/bin/sh’)函数溢出

将shellcode中的int 80调用改为system(‘/bin/sh’)。

* 1. 源码中增加汇编代码

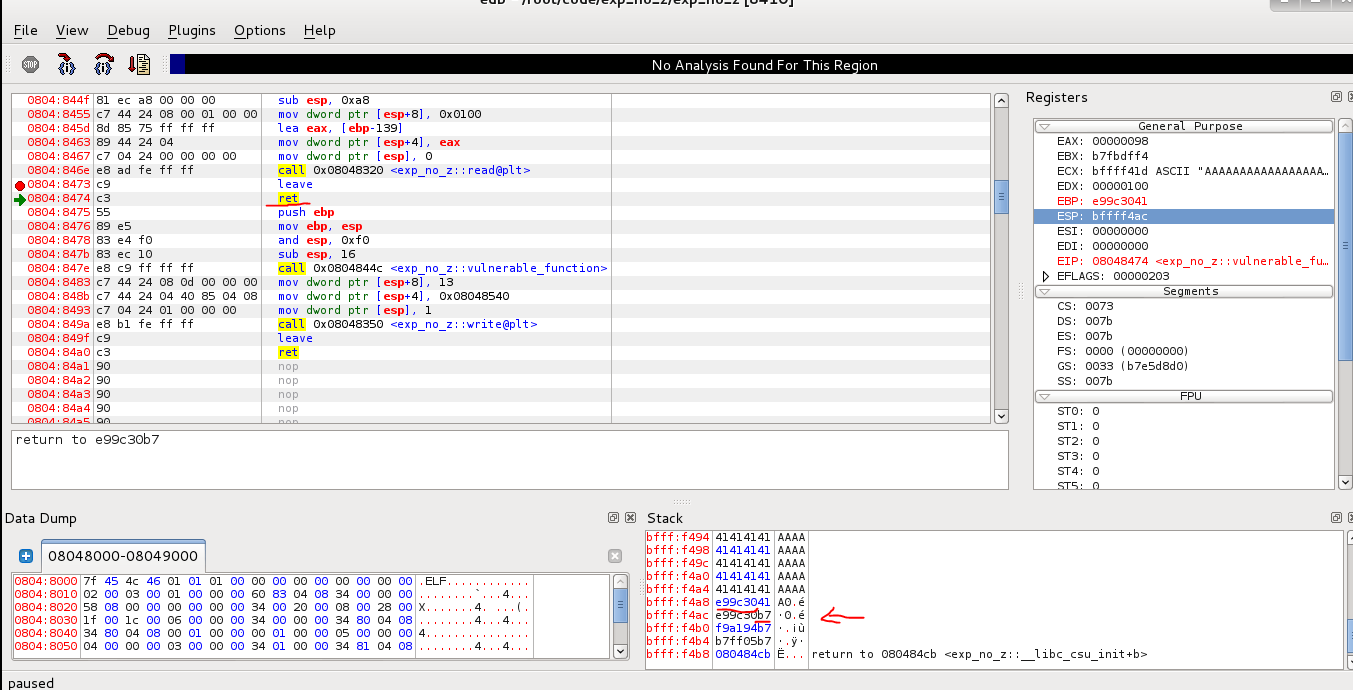


开启DEP(Data Execution Prevention)关闭aslr（实验环境在exp\_no\_z）

利用system(“/bin/sh”)溢出获得shell，利用edb的attach方式进行调试。

（关闭aslr，sysctl -w kernel.randomize\_va\_space=0）

* 1. 利用attach方式，在read()函数之后断点。寻找到输入数据到return语句的偏移量。



* 1. 用gdb调试exp\_no\_z，寻找到system()函数和’/bin/sh’字符串在动态库libc中的地址
     1. Gdb /path/exp\_no\_z
     2. 需要运行起来才会动态载入libc，所以：

B main

Run

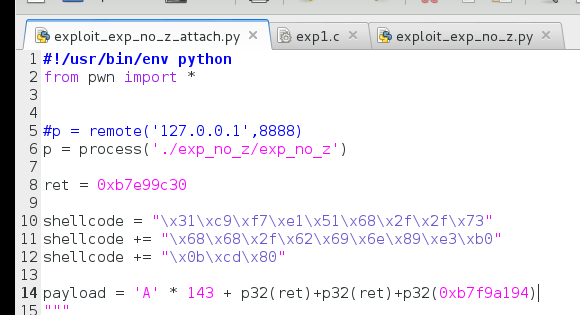
* + 1. 寻找system()函数和’/bin/sh’字符串地址：

Print system

Searchmem “/bin/sh” libc

* 1. 构造溢出的输入

首先保证return语句的地址是system()函数的地址（即ret语句时esp所指的地址）。要构造对system()函数的调用，还要构造其实参（即“/bin/sh”），所以system()函数地址后面再加上”/bin/sh”字符串的地址。但是，实际上return语句是pop eip，那么system()函数地址就出栈了。而真实的system()函数调用时，栈顶是函数返回地址，esp+4是输入参数。因此还需要在system()函数地址和”/bin/sh”字符串地址中间再插入任意4个字节数据。构造后正确的输入如下：



* 1. 成功溢出。

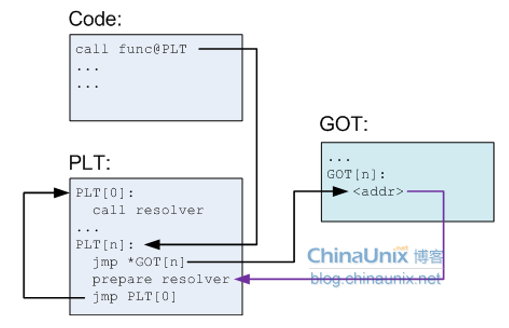
Linux下plt和got的动态链接库链接机制（实验环境在exp\_plt中）

全局偏移表 (Global Offset Table，GOT)

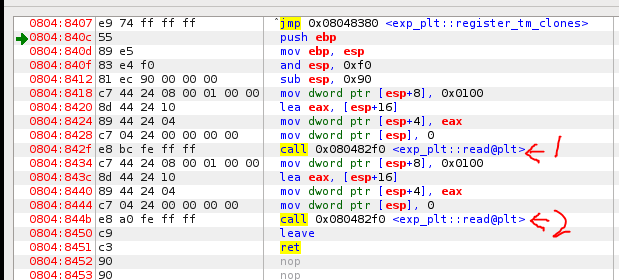
过程连接表(Procedure Linkage Table，PLT)

Plt和got是linux中非常重要的动态库加载机制。每一个外部定义的符号在got中会有相应的（可理解为map）条目，如果符号是函数，则在plt中也存在相应（可理解为map）的条目。

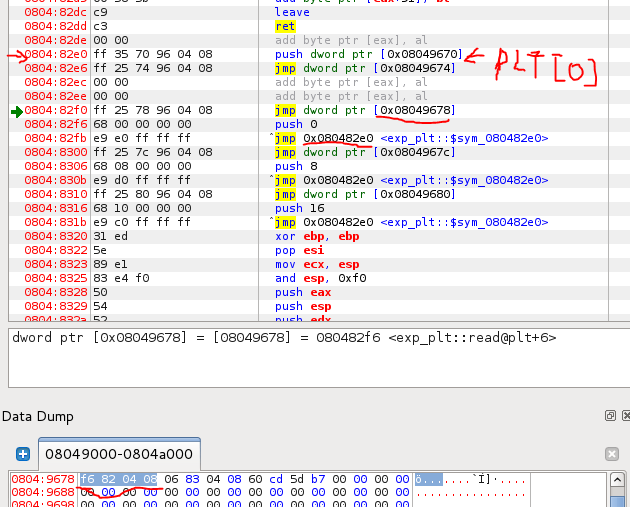
在代码中引用的函数地址就是plt中对应表项的地址，该表项会指向（其实是通过jmp实现）got中相应的表项。而got中表项的值就是该外部函数的真实地址。当第一次调用函数，got表项的值实际是plt表项的下一条指令，最终会跳转到plt[0]表项，这其实是一段解析动态加载（外部）函数真实地址的方法，该方法会获得动态加载（外部）函数的真实的地址，并且写回到got相应的表项。并且该方法运行结束后会直接跳到动态加载的函数中（不需要任何call指令，而是利用ret指令）。以后再调用外部函数时，got表项已经设置为正确的地址（而不是plt表项的下一条指令），所以会直接跳转到外部函数起始位置。



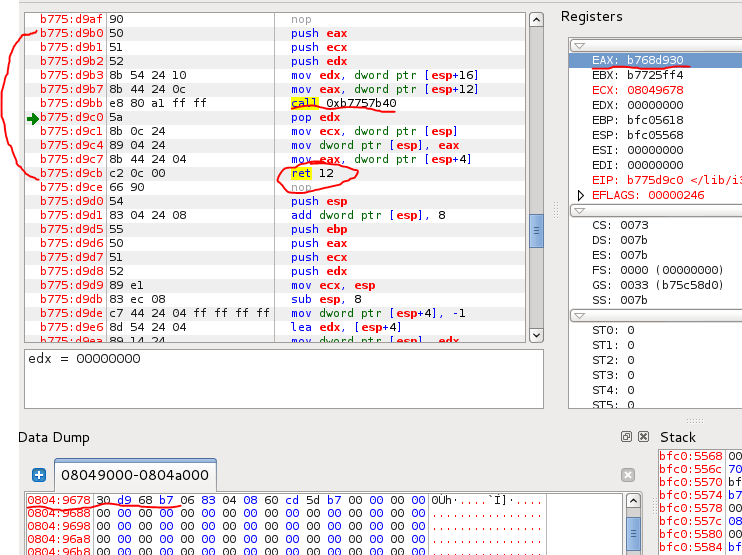
* 1. **实验过程分析：**
     1. 在edb中启动exp\_plt，执行到main入口



* + 1. 运行到第一次call 0x80482f0指令，立刻跳转到plt表中



* + 1. Plt表指向了got表中的0x8049678位置，而该got表的值是0x80482f6，就是Plt表的下一条指令（push 0）。
    2. 之后程序会跳转到plt[0]的位置（push和jmp两条指令组合形成），jmp指令便会跳转到动态加载函数方法处。
    3. 函数0xb7757b40便是具体的动态加载函数方法，它的返回值(eax)是动态加载函数的地址。此时got表项也设置为正确的外部函数地址。后面的ret指令直接跳转到调用的外部函数的起始位置，而不是简单的函数返回（注意看ret前对eax和esp的操作）。



* + 1. 第二次调用该外部函数时，got表项已经设置为正确的外部函数地址，通过got表直接跳转到外部函数，不需要再做动态加载函数。

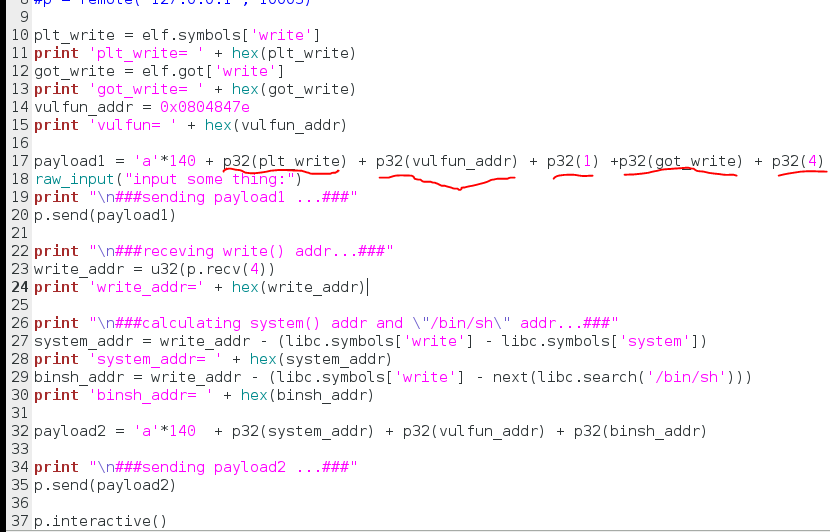
开启DEP和aslr（实验环境在exp\_aslr，exploit\_known.py）

（开启aslr，sysctl -w kernel.randomize\_va\_space=2）

* 1. 虽然开启了aslr，但是通过cat /proc/pid/maps发现用户代码段地址不发生改变，加载的libc在不断变化。
  2. 虽然开启了aslr，绝对地址会不断发生变化，但是函数间的相对地址保持不变。如libc中的read/write函数和system函数的相对地址偏移。
  3. 溢出原理：

因为一次加载中，libc中函数的相对偏移不发生改变，所以可以想办法先获得一个常用函数的地址（如write()函数，利用write函数正好可以将地址显示出来）。所以至少要溢出两次，第一次溢出获得write()函数地址，write()函数的返回地址应该为第二次溢出的函数地址；第二次溢出获得shell。

* 1. **操作步骤：**
     1. 获得write()函数在libc中的plt和got中的地址。
     2. 构造输入参数，使得正常运行函数返回时调用write()函数，并且write()函数的返回地址是第二次溢出的函数地址，同时write()的参数为write()函数动态加载后got表中的函数地址。



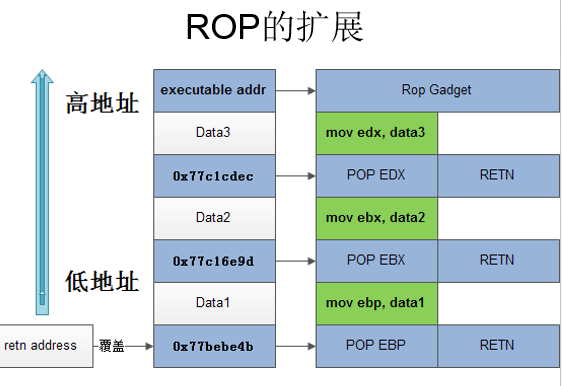
* 1. 实验后说明
     1. 只有在源程序中使用的函数elf.plt中才会存在，因此ELF.symbols['system']是不存在的。否则可以直接用plt调用system。

Exploit\_unknown

先通过搜索，获得system的函数地址。然后利用read函数读取”/bin/sh”字符串地址。

ROP

Rop就是通过调整读取指令的地址（如从指令的第2个字节开始读取解析），将指令拆解为两条指令，并且第2条指令是ret。从而通过第1条指令实现目的意图，通过第2条指令跳转目的地址。



X64

1. 和x86的区别：
   * 1. ROP中填入的地址不能大于0x00007fffffffffff，否则会抛出异常。Length计算方法，有补充。
     2. 参数传递发送变化，影响ROP中system函数参数的构造
     3. 前六个参数依次保存在RDI, RSI, RDX, RCX, R8和 R9中，如果还有更多的参数的话才会保存在栈上。
2. AT&T和Intel汇编语言的区别：
   * 1. Intel指令是 xx 目的寄存器，源寄存器

AT&T指令是xx 源寄存器，目的寄存器

在c源码中嵌入汇编代码一般使用AT&T指令。

1. 关闭dep和aslr的实验（实验环境在exp1）

实验环境换成了x64，最大的变化就是在栈中是8位对齐，而不是4位了。因此在计算栈内的偏移时，移动的最小单位是8。

1. 开启dep，关闭aslr实验（实验环境在exp\_dep，解法1是exploit\_exp\_dep，解法2是test64）

实验环境换为x64，实参的传递应该放入rdi中，而不是简单的压入栈中。所以只能利用rop将数据传入rdi并跳转。

* + 1. 方法1，利用rop: pop rdi

在跳转到system()函数地址前，要先将”/bin/sh”字符串地址传入rdi。所以在跳转system()函数前可以用pop rdi将数据传入rdi寄存器，同时将rsp+8。所以用”pop rdi”的rop指令可以实现。在gdb中用命令：

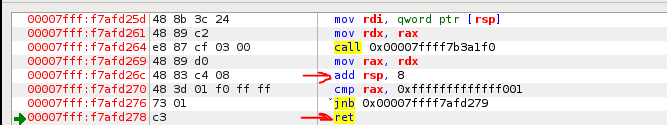
Ropsearch “pop rdi” libc

在libc中查找该rop指令，然后将溢出输入构造为：

‘a’\*n+rop\_addr+sh\_addr+system\_addr

* + 1. 方法2，利用asmsearch: mov rdi,QWORD PTR [rsp]

该方法和方法1类似，只是在rop指令寻找时没有使用ropsearch，而是使用基本的asmsearch，查找普通汇编指令：mov rdi, QWORD PTR [rsp]。该指令也是将构造好的栈值[rsp]复制到rdi中。但是，该命令可以成功必须满足两个必要条件：1.mov指令后必须自带一个ret指令，；2.mov指令到ret指令必须将rsp置为正确的值；3.在mov和ret指令之间的其他指令必须对rdi寄存器无影响。



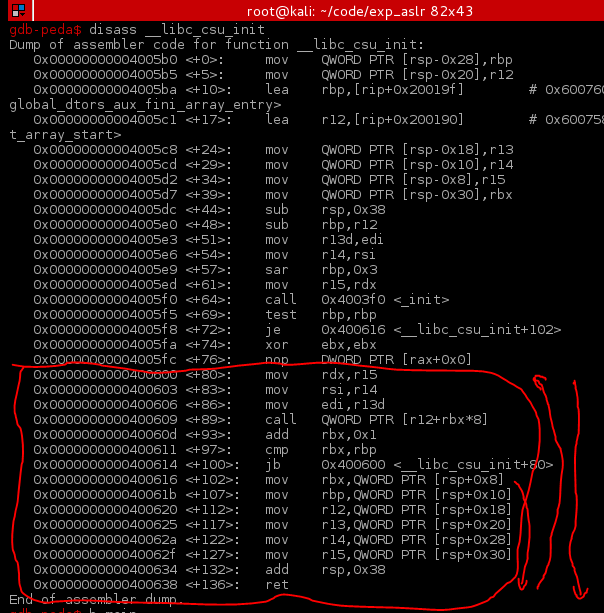
1. 开启dep和aslr实验（实验环境在exp\_aslr）

整体思路还是和x86环境下的一样，先通过调用write()函数获得其动态加载的地址（即got表项的值），并且将该地址写回。然后利用libc中偏移量固定，计算出system()函数、”/bin/sh”字符串和”pop rdi”指令的地址。

* + 1. 构造write()函数的参数并调用

因为x64通过寄存器传实参，所以构造write()函数的参数时，相对复杂。正好libc中有个\_\_libc\_csu\_init方法，该方法的后半部分代码是可利用的完美代码（如下图）。

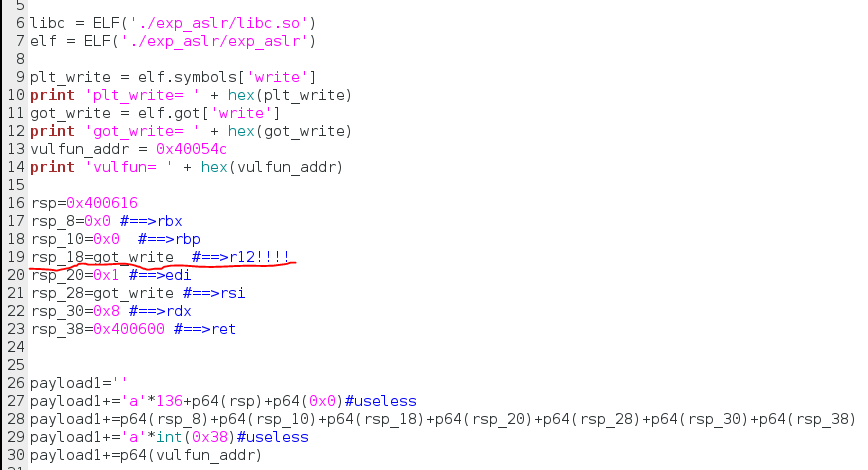
命令：disass \_\_libc\_csu\_init



通过命令 disass \_\_libc\_csu\_init命令可以查看该段代码。

* + - 1. 从0x400616开始执行，将栈中一连串数据传入寄存器。传入后正好又调整rsp。
      2. 构造返回地址为0x400600，然后程序跳回到0x400600，再次执行这段代码。
      3. 0x400600-0x400606部分的指令正好是第1步中的数据传到储存实参的3个寄存器中。通过构造合适的r12+rbx\*8就可以调用目标函数。注意这里的指令是 call QWORD PTR [r12+rbx\*8]，所以[r12+rbx\*8]应当是目标函数的地址，r12+rbx\*8就是指向目标函数的指针的地址（就是二级指针，got表的地址就是二级指针）。
      4. 最后因为rsp又下移了0x38，所以又可以控制最终的跳转地址。

根据上面的思路就可以构造溢出的输入（如下图）。首先，构造第一次溢出的地址是0x400616，使其进入\_\_libc\_csu\_init函数预先设定好的地址执行。接着，加塞一个无用的函数返回地址（第一个useless）。然后根据用处，分别在栈的rsp+8~rsp+0x30位置填充write()函数的相关参数。其中rsp+0x18是write()函数在got中表项的地址（二级指针），rsp+0x28是write()函数返回内容的地址。最后在rsp+0x38处构造ret地址为0x400600，使其回到上面重新执行。第二次执行到rsp+0x8~rsp+0x30时，随意填充（第二个useless）。最终的ret地址设为vulfunc。

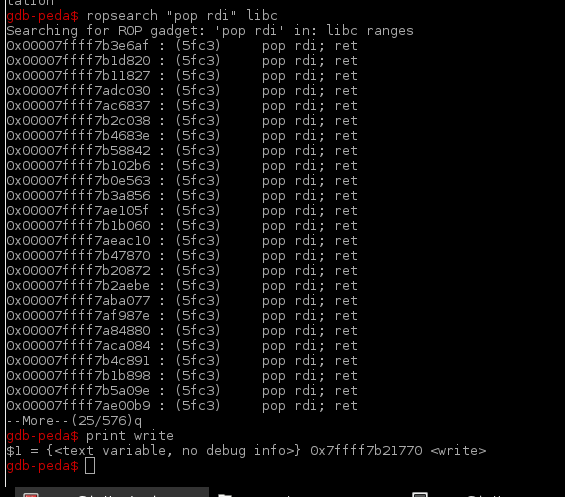


* + 1. 构造第二轮溢出的输入

构造第二轮溢出就是根据第一轮返回获得的write()函数地址，计算rop指令、system()函数和”/bin/sh”字符串的实际地址，然后进行调用。溢出输入的构造方式和上文开启dep关闭aslr的一样。

* + - 1. 计算rop指令实际地址

在gdb中运行程序后，用ropsearch名称查找到合适的”pop rdi;ret”指令，获得在Libc中的静态地址。用print write命令获得write()函数在libc中的静态地址。然后计算rop指令实际地址



* + - 1. 计算system()函数和”/bin/sh”字符串实际地址。可以和上面一样，或者和x86环境下一样。
      2. 构造第二次溢出的输入。和开启dep关闭aslr一样。

Print溢出

Printf()函数对传入参数的栈平衡（或保护）不够，容易造成溢出问题。

1. **Printf %p**

输入参数默认为地址形式进行打印，输入参数在x86环境下是esp上面的值。

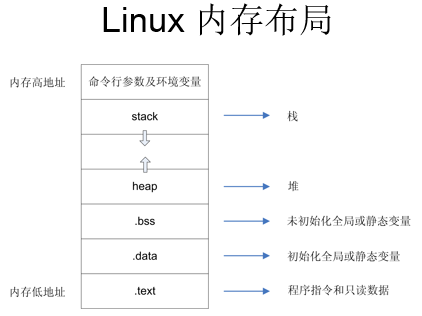
1. **Printf %n**

将该printf打印出的字符(char)数目写回到某变量中。

堆溢出

1. Linux下内存基本结构

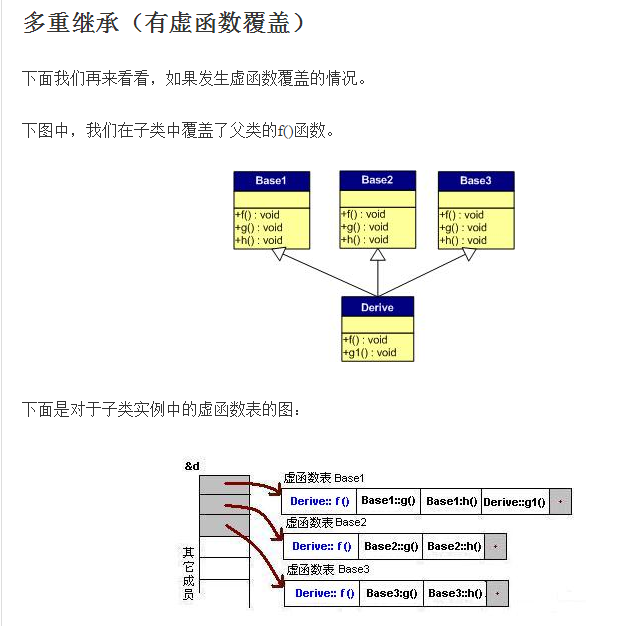
Linux下内存基本结构如下，其中栈是由高地址起始，不断向低地址填充。而堆是从低地址起始，不断向高地址填充。



1. C++虚函数和虚表

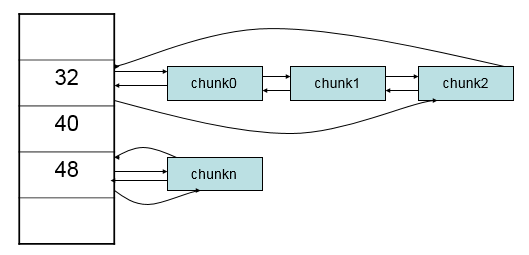
虚函数是C++中实现多态的机制，定义为虚函数的函数可以被子类重载，虚函数本身可以有实现（visual function()）。纯虚函数必须被子类重载，纯虚函数本身不可以有实现（visual function()=0）。

虚表是虚函数实现多态机制的底层设施。每个对象实例的开头位置保存了指向虚表的指针，存在一个继承就有一个虚表。没有重载的虚函数在虚表中依次排列，重载的虚函数拍在虚表的最前方。



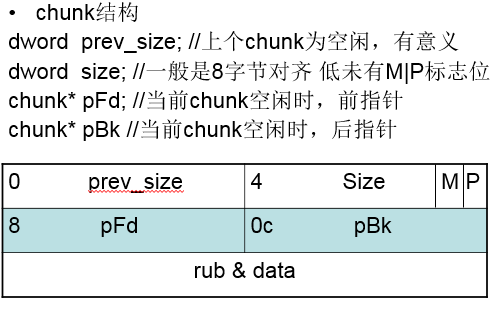
1. Linux堆结构管理

堆是用于存放类实例对象等数据的空间。当用户申请堆内存，系统会根据用户请求内存的大小划分一块内存。如果堆内存空间不够，那么系统会自动扩大堆的大小，扩张的单位是一个虚页(page)，新分配的虚页加入到堆管理中（一般是加到chunk的最后）。



堆管理结构如上图所示，每一个在堆空间受管控的内存块叫做一个chunk（可能已经分配给实例，也可能还未被分配给任何对象实例）。用户申请分配一块内存，系统依次遍历每个chunk，直到发现空闲（未分配）并且能够满足用户请求大小的chunk。该chunk会根据用户请求大小进行切分，形成一个用户需求大小的chunk和一块未分配的chunk（大小大概是源chunk-用户需求，因为每个chunk\_head还需要消耗一定的空间）。

Chunk的基本结构如下图所示，



当用户请求释放空间时，则这块chunk空间被标记位未分配，并且检查前后的chunk邻居，尽量合并为更大的空闲chunk。当有空间的chunk块大于虚页，则系统将虚页为大小的单位将堆空间返还给内存。

Shellcode

1. X86

X86环境下经典shellcode如下：

31 c9 xor ecx, ecx

f7 e1 mul ecx

51 push ecx

68 2f 2f 73 68 push 0x68732f2f   ;; hs//

68 2f 62 69 6e push 0x6e69622f   ;; nib/

89 e3 mov ebx, esp

b0 0b mov al, 11

cd 80 int 0x80

"\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80"

但是当遇到scanf()函数时，\x09(ascii解释为\t横向制表符)，\x0a(ascii解释为\n换行)，\x0b(ascii解释为\v纵向制表符)，\x0c(asicc解释为\f换页)，\x0d(ascii解释为\r回车)，都会被当做ascii来解释，从而导致scanf()函数认为输入结束！

31 c9 xor ecx, ecx

f7 e1 mul ecx

51 push ecx

68 2f 2f 73 68 push 0x68732f2f   ;; hs//

68 2f 62 69 6e push 0x6e69622f   ;; nib/

89 e3 mov ebx, esp

b0 1b mov al, 0x1b

2c 10 sub al, 0x10

cd 80 int 0x80

"\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x1b\x2c\x10\xcd\x80"

1. X64

X64下经典shellcode如下：

"\x48\x31\xd2" // xor %rdx, %rdx

"\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68" // mov $0x68732f6e69622f2f, %rbx

"\x48\xc1\xeb\x08" // shr $0x8, %rbx /bin/sh

"\x53" // push %rbx

"\x48\x89\xe7" // mov %rsp, %rdi

"\x48\x31\xc0" // xor %rax,%rax

"\x50" // push %rax

"\x57" // push %rdi

"\x48\x89\xe6" // mov %rsp, %rsi

"\xb0\x3b" // mov $0x3b, %al

"\x0f\x05"; // syscall

\x48\x31\xd2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89\xe7\x48\x31\xc0\x50\x57\x48\x89\xe6\xb0\x3b\x0f\x05

1. Eb jmp

当输入空间有限时，可能需要分割shellcode，并分布于两块相隔不远的内存。此时可以利用EB指令（short jmp）实现跳转。

EB后跟一个字节表示跳转的偏移量（有符号数），从EB指令的下一条指令开始计算。

1. 利用metasploit生成shellcode

* Show payloads，找到linux/x64/exec，并且use
* Set cmd ‘/bin/sh’，设置命令内容
* Generate –t py –b ‘\x00’，生成shellcode，-t申明shellcode类型为python，-b申明需避免的坏字符

