**超详细题解**

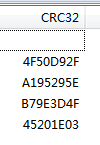
Misc:

签到题(50):

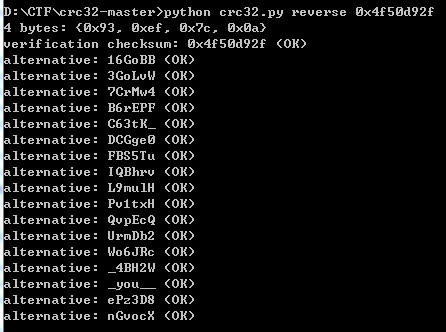


Have a try(100)：

题目疯狂提示了碰撞，这题其实是CRC32碰撞，用winrar查看可得文件的CRC32的值



使用工具碰撞



可以看出某个很像flag的字串，以此类推，加上题目格式，得到flag：zsctf{\_you\_\_\_are\_\_\_\_the\_\_best\_}

。

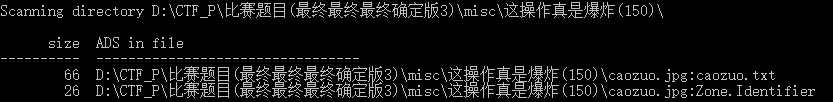
I wanna be a dalao(100):

基础隐写，有很多方法可以提取隐写的rar，最简单的方式直接把后缀修改成rar，解压后看见一个exe文件，打开可以看到是个I wanna游戏(手残党福利)，通关后得到flag



这操作真是爆炸(150):

Ntfs流隐写，这里选择使用工具lads.exe，可看到存在txt文件



使用notepad caozuo.jpg:caozuo.txt可看到以下内容:

6D6F76206561782C64776F726420707472205B6573705D2C616464206573702034

尝试16进制转ascii码

得到字串mov eax,dword ptr [esp],add esp 4，这里需要一点汇编基础，题目要求化简指令，mov eax,dword ptr [esp],add esp 4指令其实等于pop eax，加上flag格式，可得flag为

zsctf{pop eax}

少年你经历过绝望吗(200):

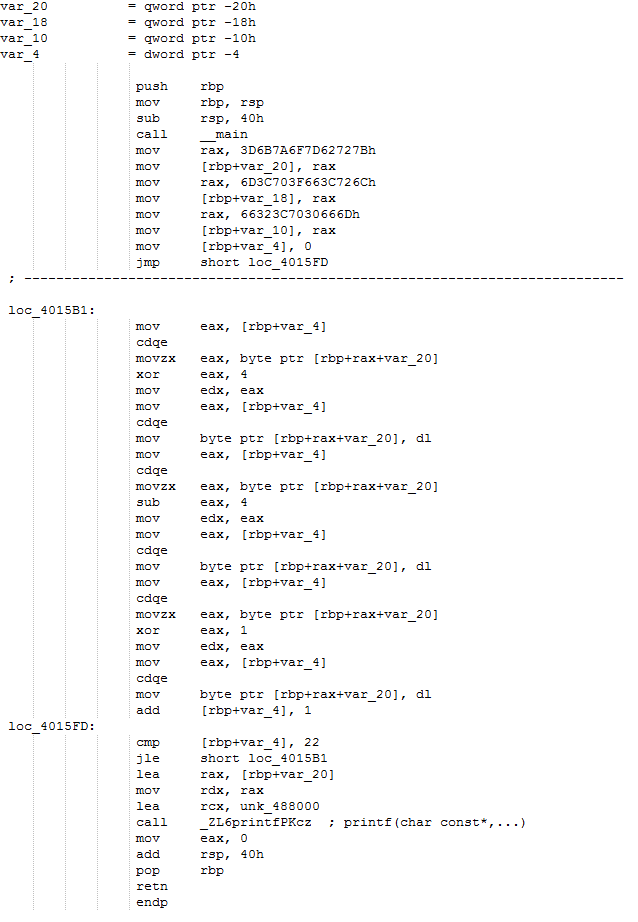
这题恶意就有点重了.....步骤挺多的，但都是老套路，有经验的可以秒了。

首先两个压缩包是有密码加密的，但有一个py文件，打开看能推测出压缩包密码的线索，py文件中d变量是重点，果断输出d，看到一个链接？



打开链接，直接就能看到解压密码是什么了。接下来解压第一个压缩包，在flag.txt中看到压缩包2解压密码是男主角中文名，通过提供的两个图片和音乐，老司机一下就知道密码是什么了，实在不认识也可以使用百度的搜图功能，得知游戏是塞尔达传说，可以查到主角名字叫林克。

得到密码打开第二个压缩包，看到3个音乐格式文件，从提示.txt中可以得知flag分三段隐写在三个文件中，txt打开第一个文件，拉到最后可以看到一段汇编。



分析可得程序是将

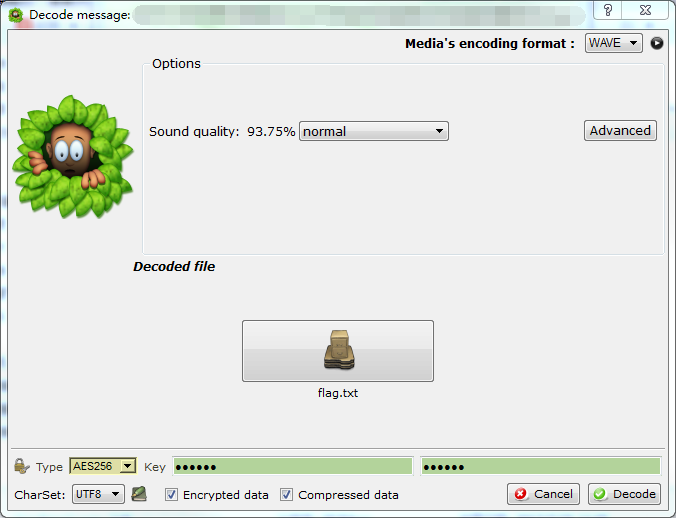
0x7b,0x72,0x62,0x7d,0x6f,0x7a,0x6b,0x3d,0x6c,0x72,0x3c,0x66,0x3f,0x70,0x3c,0x6d,0x6d,0x66,0x30,0x70,0x3c,0x32,0x66

这段字符每个进行异或4再减4再异或1，然后输出，按照汇编逻辑写出程序即可得

成功得到第一段flag。

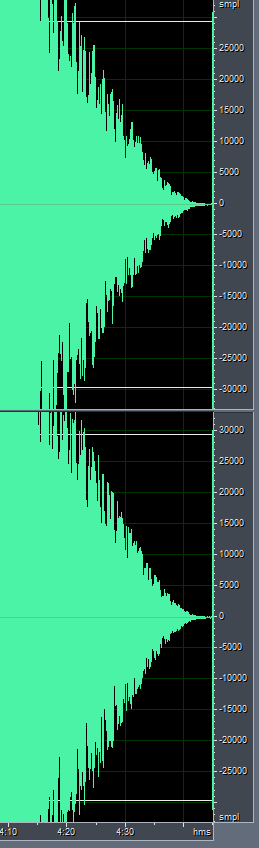
接下来分析第二个文件，照样尝试文本打开，拉到最后看到



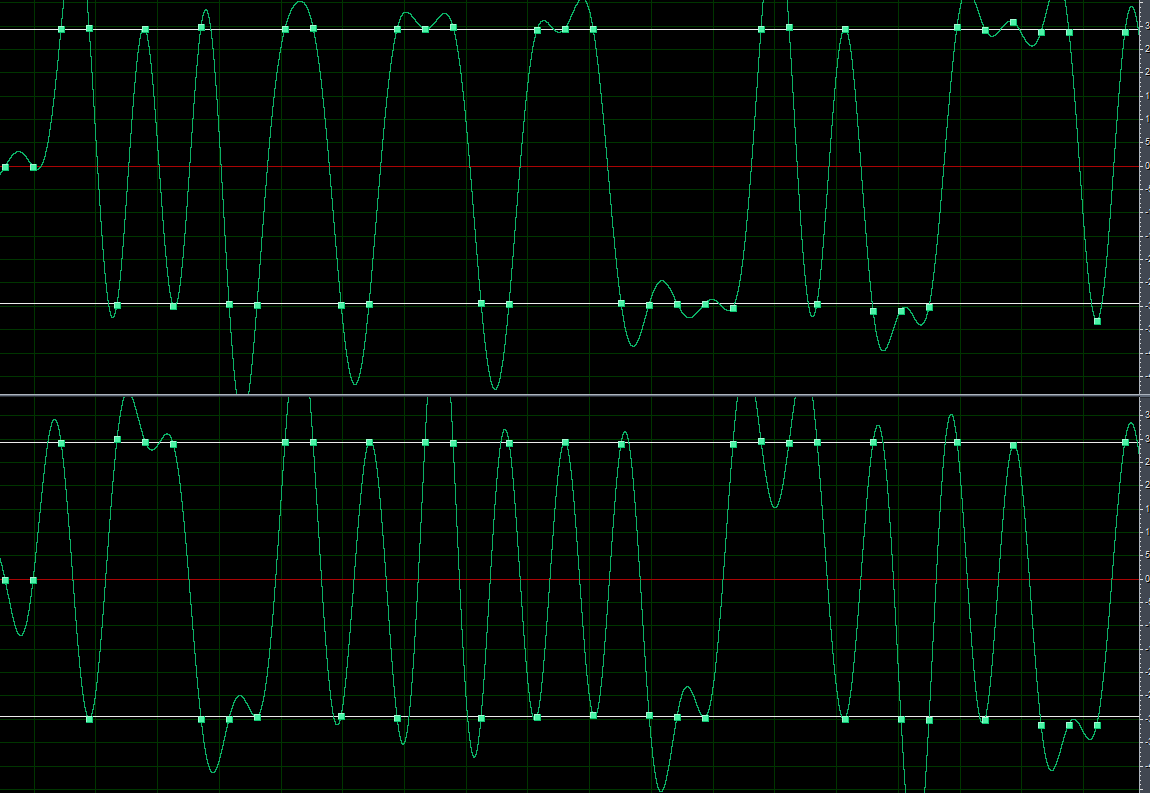
结合文件格式是wav，想到可能是silenteye工具的隐写，将文件放进工具，进行解密，可得

打开flag.txt可得第二段flag。

接下来就是最后一个文件了，最后一个文件的隐写在波形上，用任意一款可看波形的音乐编辑软件，看到



音乐文件结尾波形存在异常，放大观察

可以看到有上下起伏的波形图，在上的点代表二进制的1，在下的点代表二进制的0

上音轨和下音轨存在不同的结果(下音轨用于混淆)，将提取的二进制串转ascii，最后尝试

得上音轨为正确结果，拼合三段flag，得zsctf{j4es5\_6q5dd\_1q53\_0x61\_jg84}

Misc题的wp到此结束。

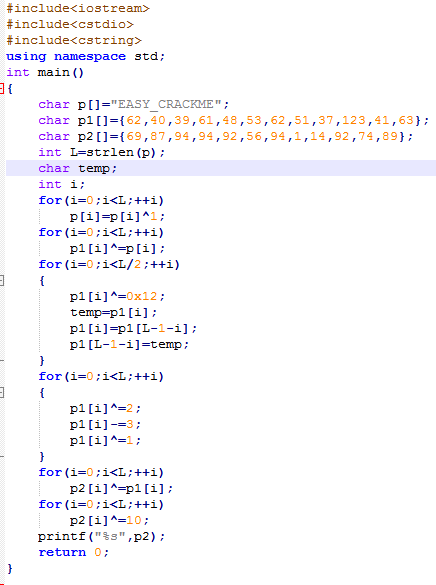
Crypto:

不算加密的加密(50):

瞎写的加密......

打开cpp文件查看代码逻辑，得到的线索有：key在某个地方，string1和flag加密后的值，

写解密脚本解密即可，根据代码的逻辑反推，脚本如下:

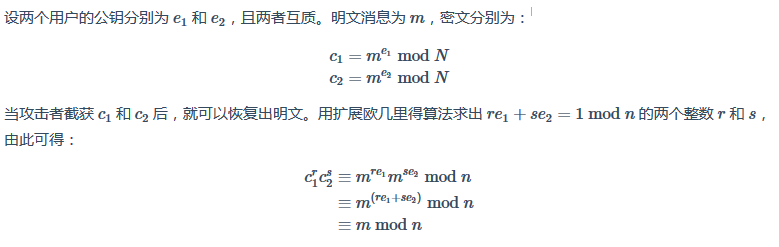


最后得到flag：zsctf{88e79\_1qw545}

加密对话(100):

一开始nc上去能看到一段对话(劝退开始)，一开始是一个猪圈密码，搞定后看到一段东西，能猜想RSA，因为主流的公钥加密也没多少，题目给出公钥{n,e}，根据题目的描述，可以知道两段message的明文是一样的，而且可以看到给出的公钥{n,e}只有e不同，且e互质，n是相同的，所以可以使用共模攻击来算出明文，

原理为：



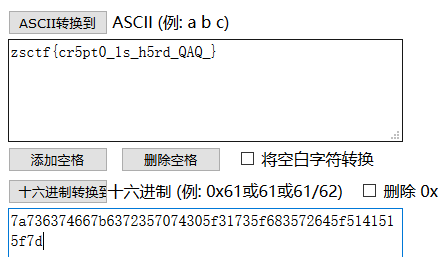
有兴趣的大佬可以研究一下，利用公式可以写出脚本如下：



运行后输出：



将16进制转换成ascii码即是flag:



nukami is pong(150)：

write up by nukami

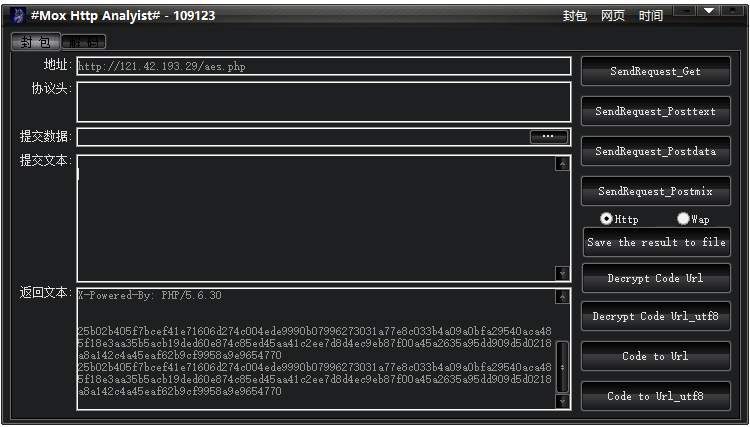


从PHP代码中我们可以看到我们POST给接口的文本会被按行切分后在每一行末尾加上

$flag\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_$surprise 后进行加密，加密参数中的rijndael-192即是192位的AES加密，而ecb则指明AES模式为ECB。可以猜测这道题目是要我们通过控制提交的文本，利用AES-ECB安全问题来获取flag。

我们已知加密位数是192位，也就是对应着一个区块有24字节，而它的十六进制表示则会有48个字符。

我们首先提交个”\n”，从php代码可以看出这样得到的加密结果是单纯的我们需要爆破的数据$flag\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_$surprise的加密结果



将加密结果按照48位hex字符分割

25b02b405f7bcef41e71606d274c004ede9990b079962730

31a77e8c033b4a09a0bfa29540aca485f18e3aa35b5acb19

ded60e874c85ed45aa41c2ee7d8d4ec9eb87f00a45a2635a

95dd909d5d0218a8a142c4a45eaf62b9cf9958a9e9654770

可以知道待破解的文本 $flag\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_$surprise 占了4个区块，也就是说它的长度在(72,96]的区间内，然后我们再逐渐添加提交的文本长度，当我提交到” #######################”(len =23 )时，加密结果变成了5个区块，这意味着当我们填充了22个字符时，待加密的明文恰好填满了四个区块，因此我们待破解的文本的长度就是96 – 22 = 74

这个待破解长度的计算方法是为了自动化脚本准备的，而如果我们要节省脚本编写时间直接手动破解的话，就可以直接这样

先提交 #\*48(这代表着48个重复的’#’)，这会恰好占用两个区块，这两个区块返回的密文是

78fcc4055e1c1fd88bff72500e16e046193cbbd2547f4cba

78fcc4055e1c1fd88bff72500e16e046193cbbd2547f4cba

也就是说

#\*48 对应的密文就是 78fcc4055e1c1fd88bff72500e16e046193cbbd2547f4cba

接下来我们缩减一个#，提交#\*47,这时候第二个区块的密文发生了改变，变成了

93b506b316c30811cde4fba5bde987a81cfefa11c13c82a7

也就是第二个区块中，这个由我们已知的#\*23和一个未知的字符构成的文本，加密后的结果是

93b506b316c30811cde4fba5bde987a81cfefa11c13c82a7

这个时候，我们修改第24个’#’成任何一个可能的字符，并提交获取它的密文，这个时候我们提交的内容看起来就像

#...#a#...#? (?用来占位，代表未知的字符)

#...#b#...#?

#...#c#...#?

…

#...#f#...#?

我们将可能的字符逐个替换到第24位中，直到替换了字符’f’时发现，返回的密文

93b506b316c30811cde4fba5bde987a81cfefa11c13c82a7

93b506b316c30811cde4fba5bde987a81cfefa11c13c82a7

71aab001f8769ddfc923f8e3a028838388d1984a5901c7a2

4fdff6422e59d368cdd97f0278e8303d604d326e4557ce47

486daa5b4ead9746023d7e19bf0dbab029468221dfe9ba53

b681de07e44b8e09a9774e65ae6693f76b5f569bcecf7be0

第一个区块和第二个区块又变得一样，这意味着这两个区块的明文相同，因此我们可以知道第二个区块中最后的一位，那个我们待破解的未知字符是’f’，依次原理逐个往下爆破

#...#fa#...#f?

#...#fb#...#f?

…

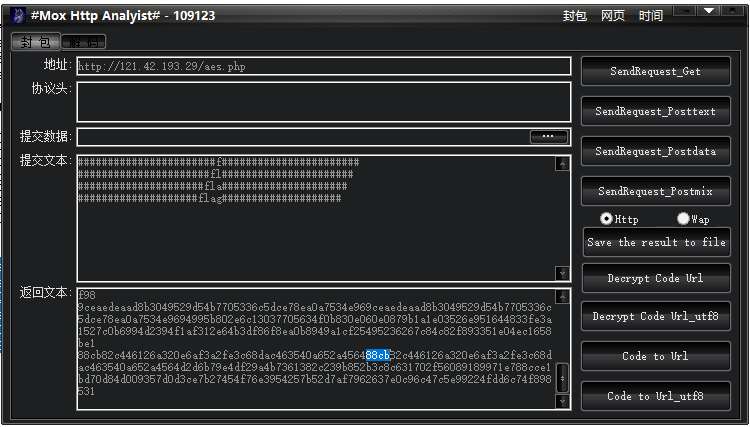
#...#fl#...#f?

#...#fla#...#fl?

…

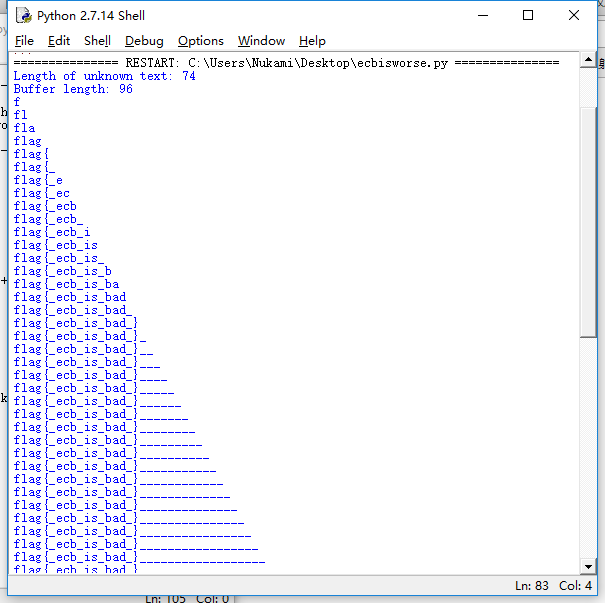
这样逐位爆破下去，我们就能得出 $flag\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_$surprise 的完整内容

flag{\_ecb\_is\_bad\_}\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_surprise{ASDSD-ASDAS-DSAADS}



这个方法理论上可以碰撞出插入点之后的所有字符

附上自动化脚本



import urllib2

#flag{\_ecb\_is\_bad\_}\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_surprise{ASDSD-ASDAS-DSAADS}

\_blockBits = 192

\_hexBlockLength = \_blockBits / 4

\_blockLength = \_blockBits / 8

\_url = "http://121.42.193.29/aes.php"

def Post(content):

try:

request = urllib2.Request(\_url,content)

connect = urllib2.urlopen(request)

response = connect.read()

except:

response = ""

return response

def CaculateHexBlockCount(cipherHex):

return len(cipherHex) / \_hexBlockLength

def TestUnknownBlockCount():

testResponse = Post("\n")

array = testResponse.split("\n")

return CaculateHexBlockCount(array[0])

def TestUnknownTextLength():

unknownBlockCount = TestUnknownBlockCount()

payload = ""

for i in range(\_blockLength):

payload = payload + " " \* i + "\n"

response = Post(payload)

array = response.split("\n")

for i in range(len(array)):

if(CaculateHexBlockCount(array[i]) > unknownBlockCount):

return \_blockLength \* unknownBlockCount - i + 1

return 0

def GenerateBuffer(unknownTextLength):

i = \_blockLength

while i < unknownTextLength:

i += \_blockLength

return "#" \* i

def GetDictionary():

return "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ\_-{}"

def PaddingPayload(text):

str = GetDictionary()

payloads = [text + "#"]

for c in str:

payloads.append(text + c)

return payloads

def BuildPayload(buffer,cracked):

postBufferLength = len(buffer) - len(cracked)

basePayload = buffer[-postBufferLength + 1:]

paddingPayload = PaddingPayload(("#"\*\_blockLength+cracked)[-\_blockLength + 1:])

for i in range(len(paddingPayload)):

paddingPayload[i] += basePayload

payload = "\n".join(paddingPayload)

return payload

def Pong233(payloads,ciphers,workBlock):

str = GetDictionary()

payloads = payloads.split("\n")

ciphers = ciphers.split("\n")

if(ciphers[-1] == ""):

ciphers =ciphers[:-1]

workBlockIndex = \_hexBlockLength \* (workBlock - 1)

for i in range(0,len(ciphers)):

cipherBlockA = ciphers[i][0:\_hexBlockLength]

cipherBlockB = ciphers[i][workBlockIndex:workBlockIndex+\_hexBlockLength]

if(cipherBlockA == cipherBlockB):

return payloads[i][\_blockLength - 1]

return ""

def Crack():

unknownLength = TestUnknownTextLength()

print "Length of unknown text:",unknownLength

buffer = GenerateBuffer(unknownLength)

bufferLength = len(buffer)

workBlock = bufferLength / \_blockLength + 1

print "Buffer length:",bufferLength

cracked = ""

while unknownLength > 0:

payload = BuildPayload(buffer,cracked)

response = Post(payload);

if(response == ""):

print "Error: Empty Response!"

break

crackResult = Pong233(payload,response,workBlock)

if (crackResult == ""):

print "Crack failed!"

break

cracked += crackResult

print cracked

unknownLength -= 1

return cracked

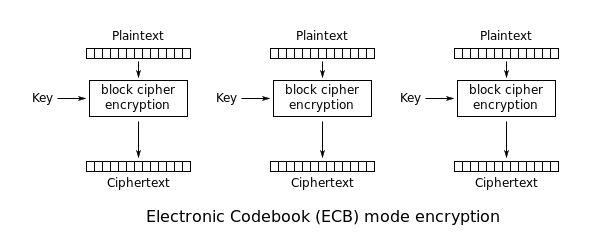
def main():

Crack()

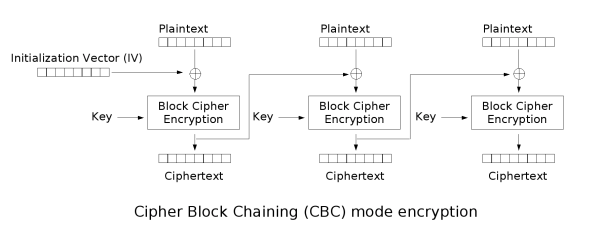
if \_\_name\_\_ == "\_\_main\_\_":

main()

原理：



上图显示了ECB模式的加密流程，明文Plaintext被切分为多个等长区块，用相同的密钥Key通过加密算法进行加密得到各个区块对应的密文Ciphertext，它们多个区块间相互独立，这么设计给ECB带来一个优势就是可以并行加密，可以多线程并行加密多个区块，然后再把各个区块密文连结起来，提高加密效率。



上图显示了AES的另一种模式 CBC mode的加密流程可以看到CBC与ECB对比，除了多了一个初始向量IV外，就是它各区块之间多了个箭头连线，这连线表示一个区块的加密结果会如IV之于第一个区块一般，作用于下一个区块的加密过程。CBC与ECB之间的这个区别体现在加密结果上的差异就是：ECB因为区块独立，如果有两个区块它们的Plaintext内容相同，那它们的Ciphertext也会一模一样。而CBC则会因为前一个区块的加密结果（或IV）参与了加密运算，即使两个区块的Plaintext内容相同，它们的Ciphertext也会不同。

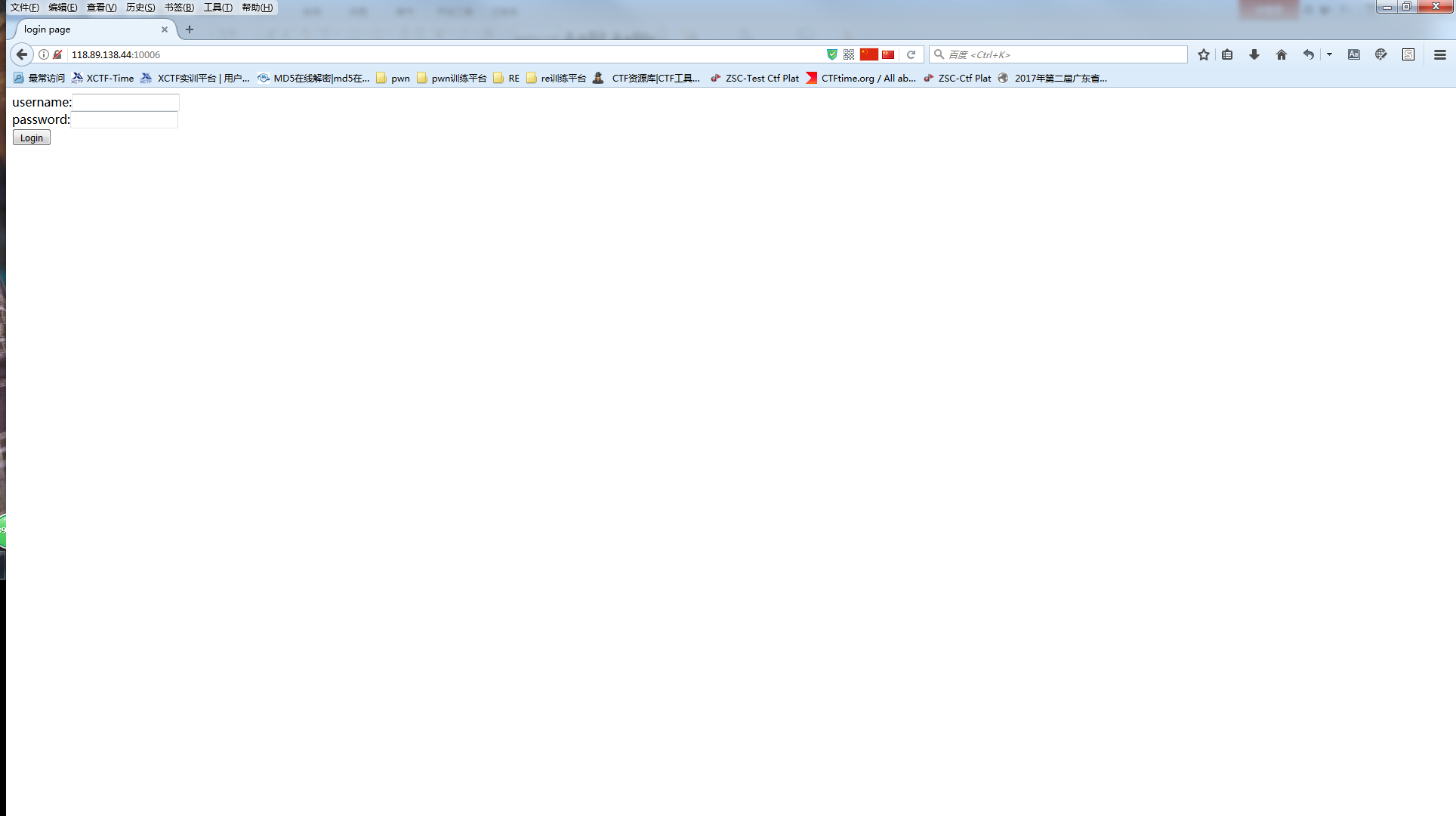
正是由于ECB模式区块独立的特点，当我们可以往一待加密的数据中插入数据（存在插入点），并能获取到对应的加密结果时，我们就有可能碰撞出插入点之后存在的未知数据。如题目中碰撞出服务器php代码中，被插在提交文本后面的flag。

Web:

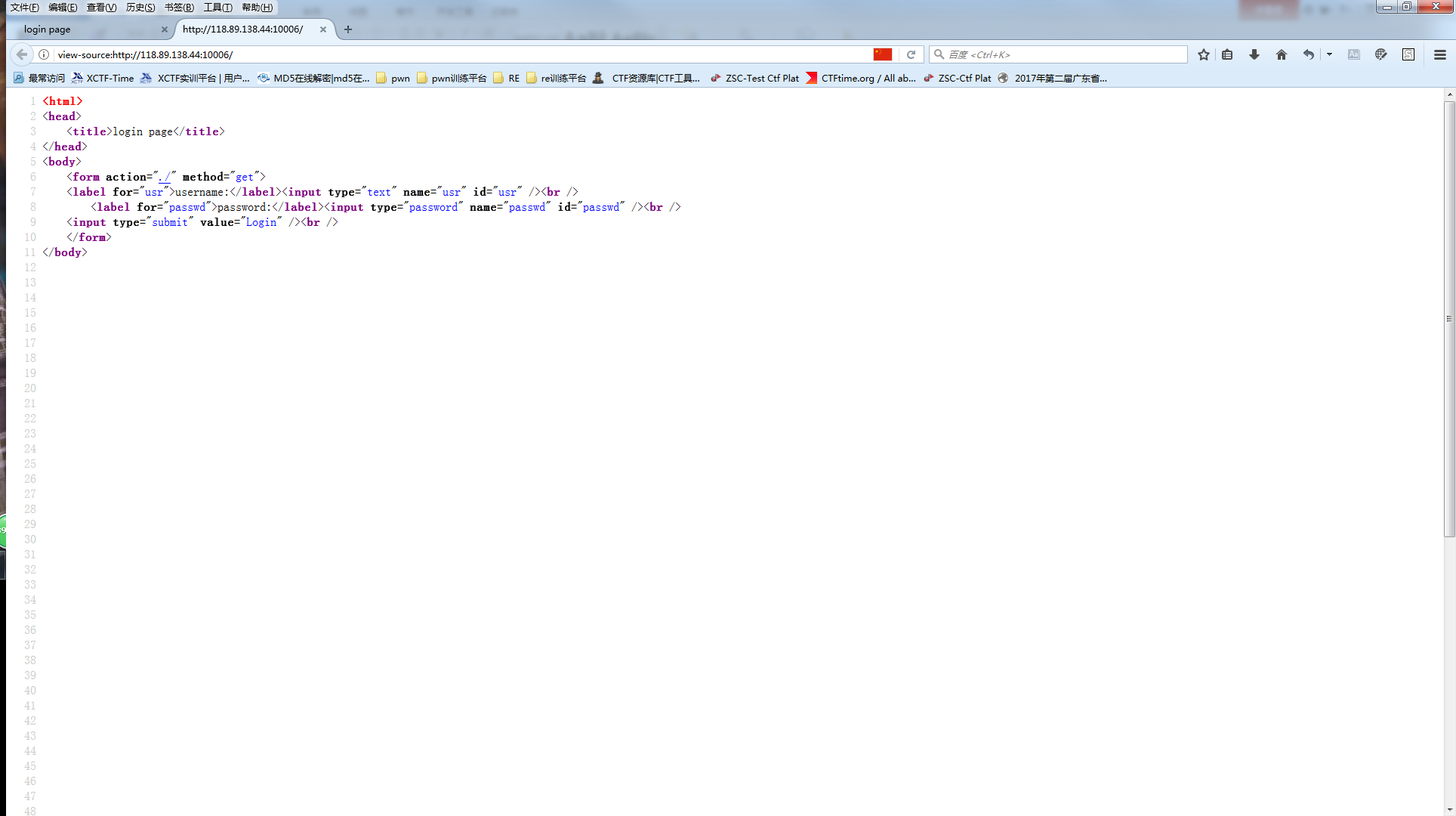
Login(100):

这题考察的是php的特征。

打开页面可以看到界面



老规矩，右键查看源码

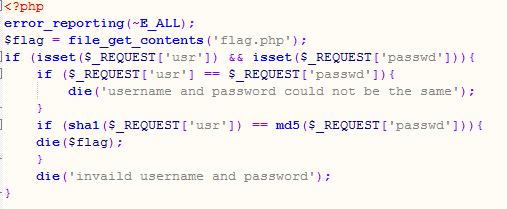


发现下面还有内容



直接访问index.src

可以下载到src文件，打开可以看到index.php的源码



可以知道当usr的sha1值等于passwd的md5值的时候，就会输出flag

我们选择利用Php的特性。



得到flag



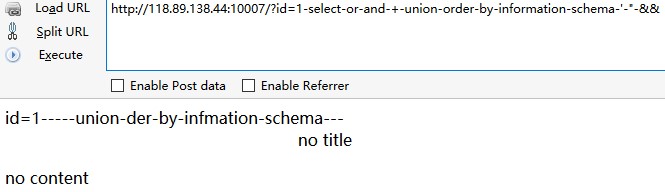
至于为什么，请自行百度

sqli1(150)：

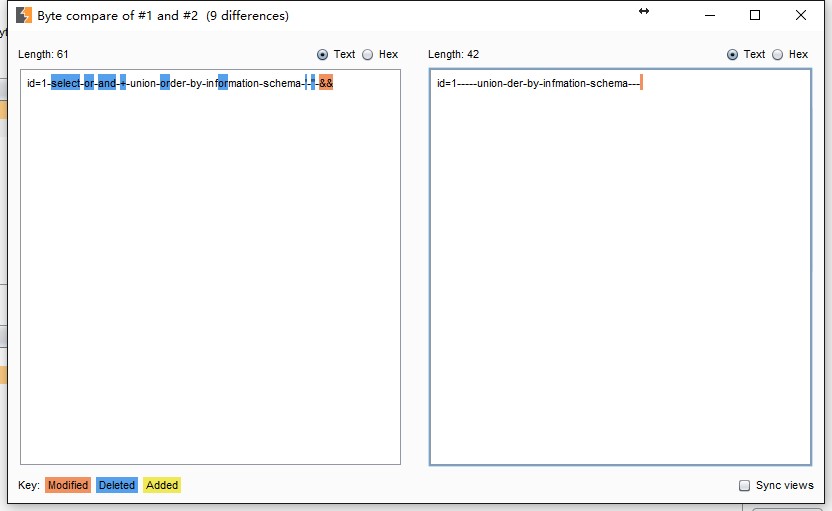
by 网络科技谐会=>hakira

拿到题目很明显是一道sql注入题

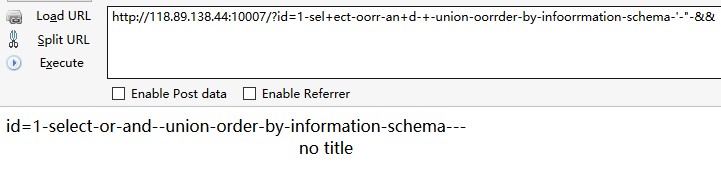
在页面 上方给出了注入语句的过滤后的样子，方便 进行一些基础关键词的测试之后发现过滤了 空格，等号=，单引号’，双引号”，加号+，select， or，and等很多关键词。



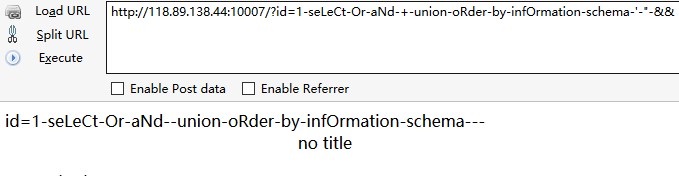
也可以用burp来对比一下那些被过滤



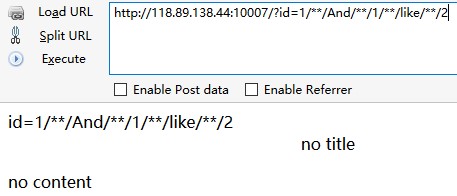
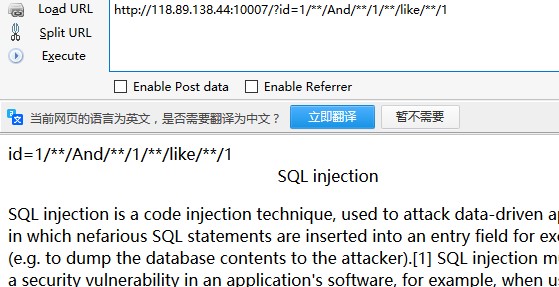
知道被过滤了什么东西之后，我们可以尝试进行绕过，测试发现，题目的防护并不是递归过滤的，我们可以在关键词中加入会被过滤的字符。如下：



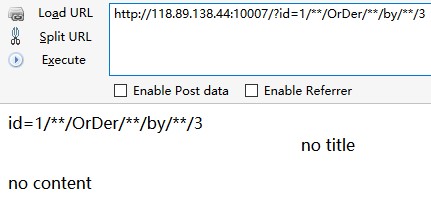
在此题目中也可以直接使用大小写进行绕过



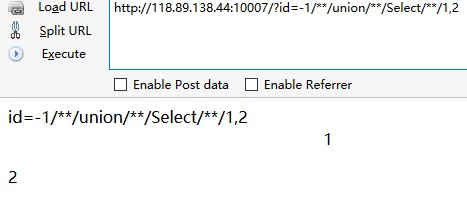
空格的过滤可以通过%0a（换行符）和注释/\*\*/来绕过 等号=可以的用like和in进行替换



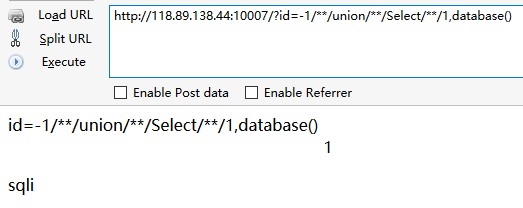
确定可以进行测试之后，进行手工联合注入查询 通过order by来判断字段数



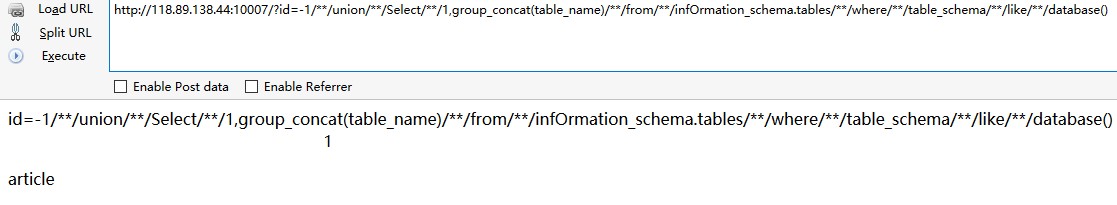
确定为2个字段，接着爆出显示位



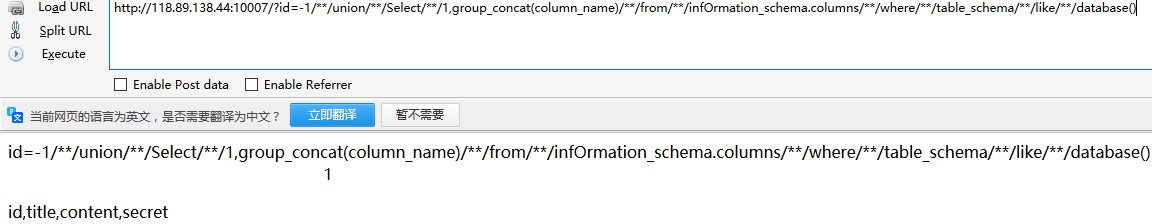
通过database()获取当前数据库



查询对应的表有哪些



然后就是查字段：）



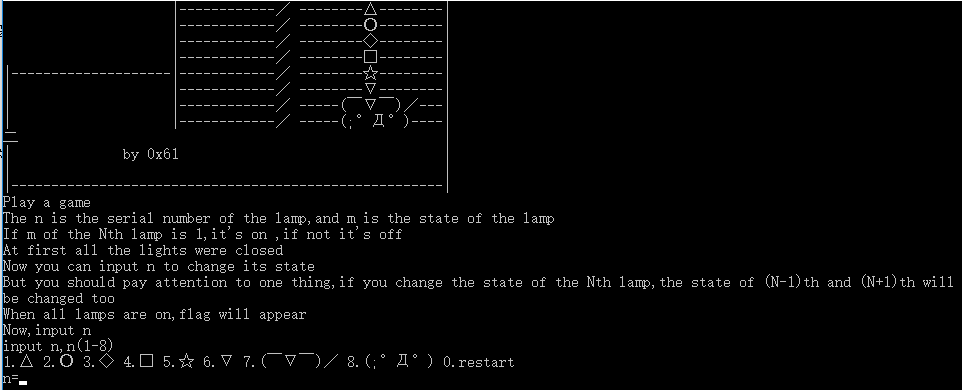
接下来就是getflag了XD（emmm他这是防了一波用limit的君子）



Reverse:

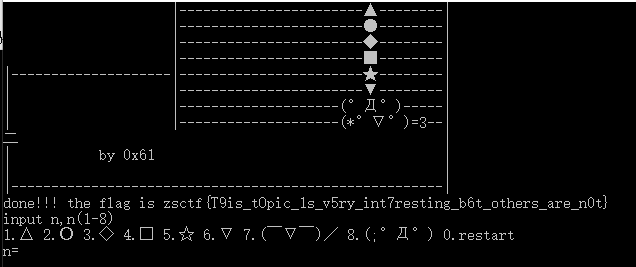
A game(50):

这题是某天脑抽写出来的23333，是经典的逆向不如通关系列，双击打开发现确实是个游戏，题目简介并没有骗你们。但是还是像crypto一样，劝退第一关是要看比较撇脚的英语。

，打开看到的是一个图形(其实是一个电路图。。。)，根据介绍可以

知道游戏就是要把八个开关都打开，flag就会出现，这类游戏有一个简单的解决办法，就是从1到n的开关都操作一遍，就可以得到答案，当然也可以选择逆向，把生成flag的代码抄下来。照样可以得到flag。

每个开关打开的关闭的状态都不一样的，233333，我是觉得还蛮有意思的，大家可以去玩玩，通关之后得到flag



something exciting(75)：

这题国赛的re200原题，但是换了个数独棋盘，读懂判定的代码就可能知道这个其实是个数独游戏，填入的正确的数就是flag。老规矩IDA伺候



通过字串可以定位到核心函数，该函数的主要功能其实是 棋盘生成-->检验输入合法性--->检验输入正确性

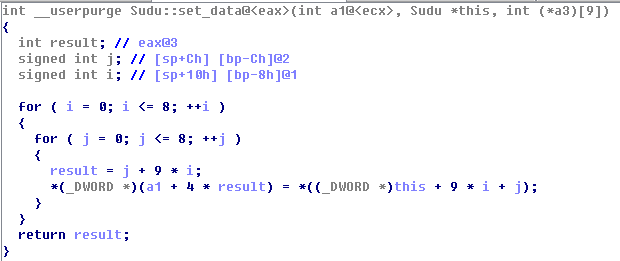
其中分别对应的函数是---->棋盘生成

中的

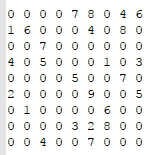
---->检验输入合法性

--->检验输入的正确性

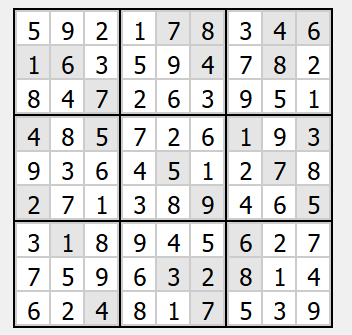
接下来详细分析每个函数的判断过程，首先是set\_data的棋盘生成



通过this传入的数据对a1指向的内存空间进行初始化，需要注意的是指针的类型是DWORD，也就是DD，4字节。查看对应内存空间的数据可以得到数独棋盘为



网上随便可以找到解题器(当然你也可以自己玩)，得到结果



然后分析函数set\_sudu里的set\_number，可以知道填数的规则是棋盘有的填0，其他正常

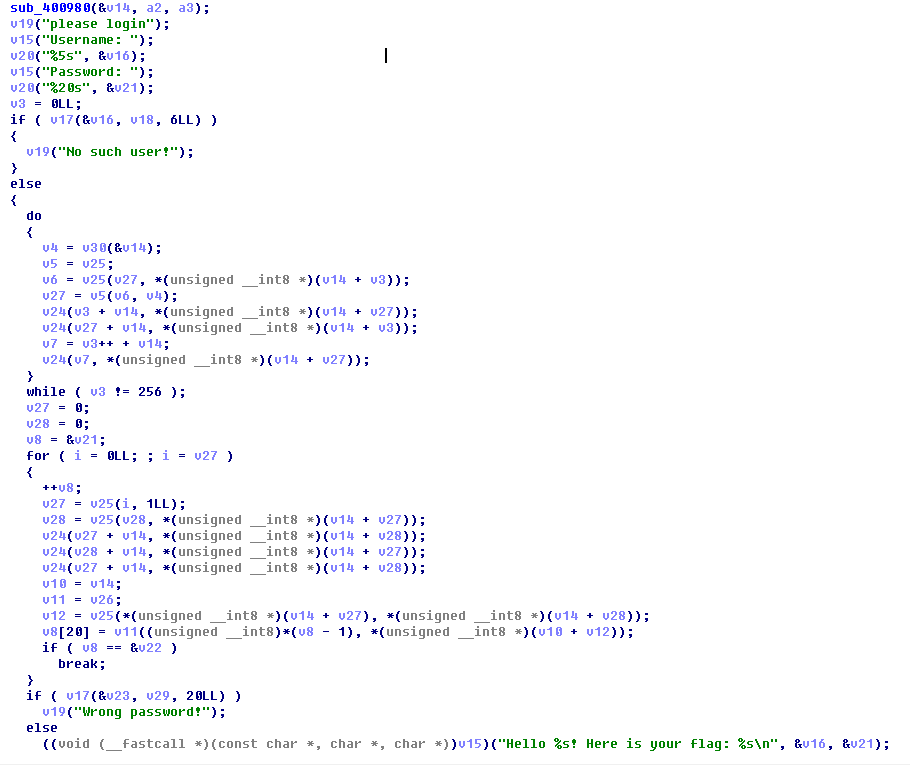
而check函数很显然就是检验填入的数独是否符合规则。

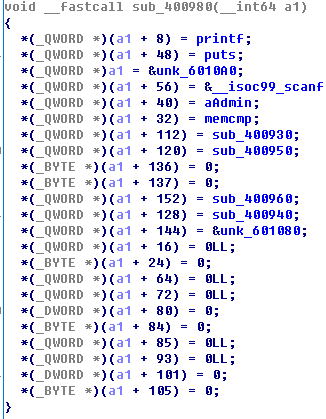
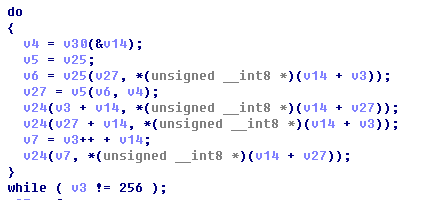
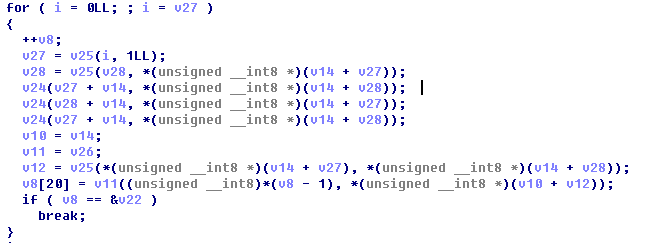
最后可以得到

flag:zsctf{592100300003590702840263951080726090936401208071380460308945027759600014620810539}

login(100):

这题是个64位的elf文件，解题可选的方法有很多，这次还是祭出IDA，方便快捷，定位到主函数得到

萌新到这里一般都卡一下，因为看到函数是变量来的，其实在sub\_400980里已经把各个函数的地址赋给了这些变量，

如图，这里可以选择纯静态，把对应的东西都记下来，再在主函数里进行审计，但是有一个更加简单的方法，就是用动态，可以直接查看变量对应的函数而且最重要的还是可以跳过程序段，这个程序段是一段环境的初始化，如果要纯静态，会增加一些复杂的过程，最后分析出来程序段对字串进行的处理，

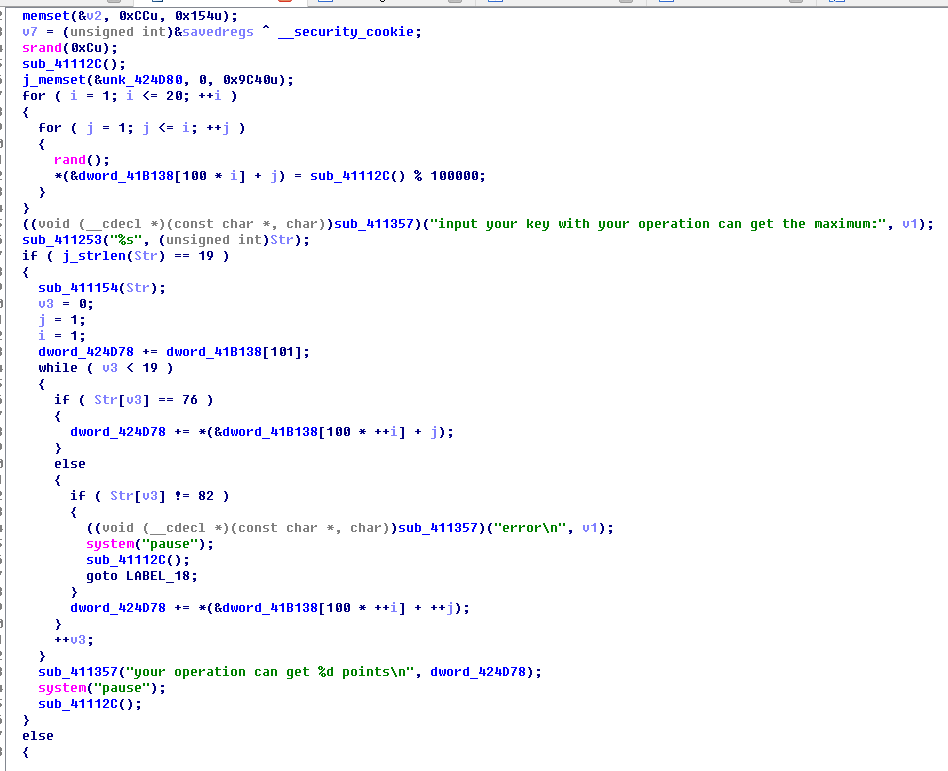
得出flag：zsctf{Ea4y\_A1l3ged\_RC4\_rE~}

Mountain climbing(150)：

这题加了UPX壳，我还把UPX的UPX!标识符给删了，用网上的一般脱壳机是不能识别的，但可以修复标识符后再用工具脱，或者OD手脱。



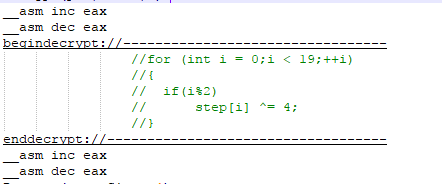
脱掉之后直接IDA。



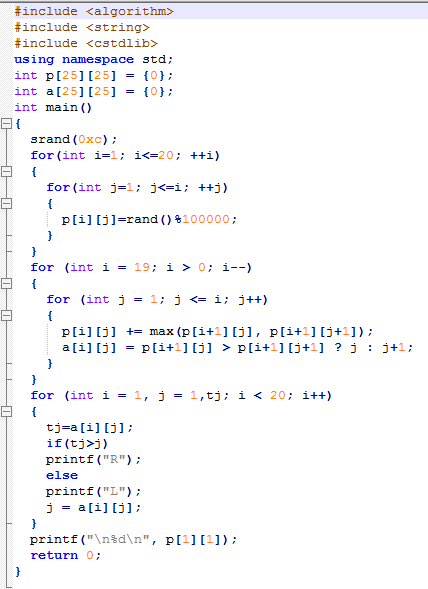
这题需要一定经验，因为分析出来的函数很乱，甚至能看到rand()函数单独在一行，但实际上是在sub\_41112C()这个位置的，还有一些printf和scanf都乱了。

程序的流程是先生成一个地图，是一个数塔，要求用户输入指令，在数塔上模拟移动，每一个目标点都有这对应的积分，要求移动经过的路径的积分最大，需要用DP解决，网上就能找到代码。

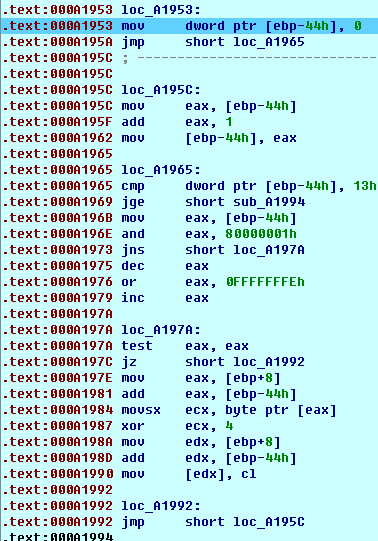
除了杂乱的伪代码外，另一个可能会卡的地方是sub\_411154((int)Str)函数，这个函数的主要功能是动态解码程序部分代码，然后再执行，函数功能如下



其中step是用户输入字串，进行处理之后再传进下面的判断程序，主判断程序的解决脚本如下：

(程序出来的结果需要删除一个字符，因为并没有优化)

出来的字串再按照动态解码的函数内容



对字串进行异或就是答案了

Flag:zsctf{RVRVRHLVRVLVLVRVLVL}

Take the maze(200):

一开始不打算出压轴。。。但是看到题库的大佬扫题的战绩，想了想还是出道压轴的比较好。

这题的核心是VM+自定义检验，检验是以一个迷宫的形式检验字串能否按对应的操作到达终点。

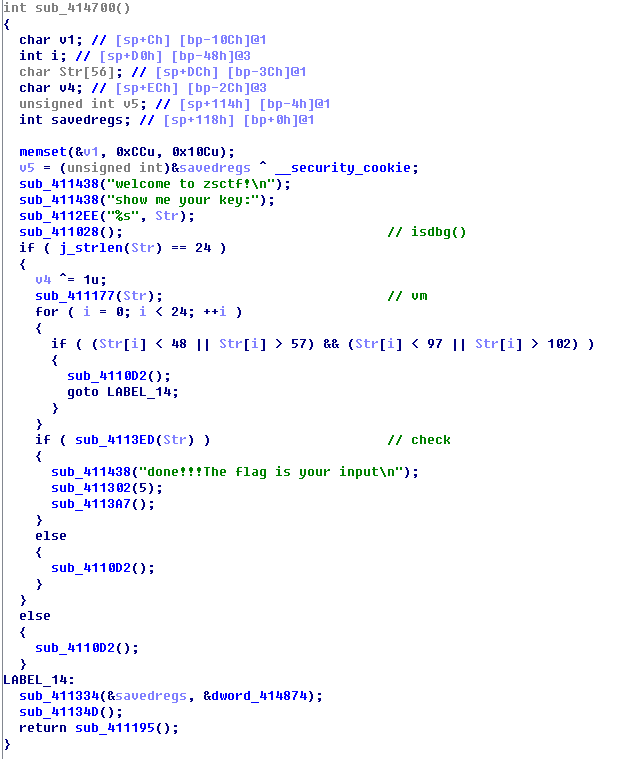
首先在程序某些地方加了”慵懒”的debug检测，防一手猛男们调试，也存在栈不平衡来了一手防F5，需要patch C:\Users\Administrator\AppData\Roaming\Tencent\TIM\Temp\6PZT)JB{JDQ[FE~$M)}6GW0.gif



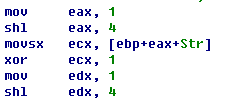
首先要将程序patch后才能F5，搞定之后进入主题。

一开始是一个入门级vm，可以尝试翻译字节码。实在翻译不出来也可以用黑盒的方式测试，也为了方便调试，写了32位的。

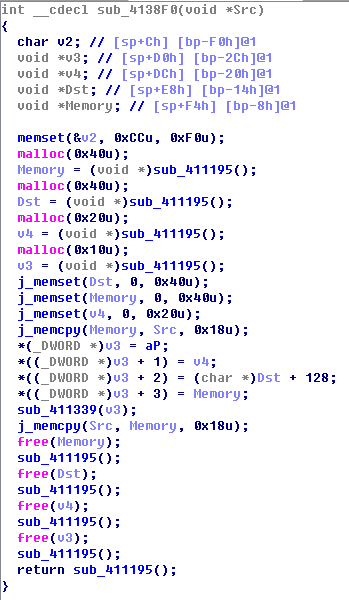
IDA打开，用字串定位主函数。



这里有个问题就是出现了一个v4^1u，在伪代码中并说明v4得指向，查看汇编

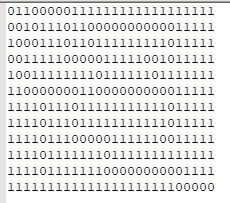
可得是指向字串的第17位(str[16])。

而从程序可以确定输入字串的长度为24，输入之后就在vm中进行处理。

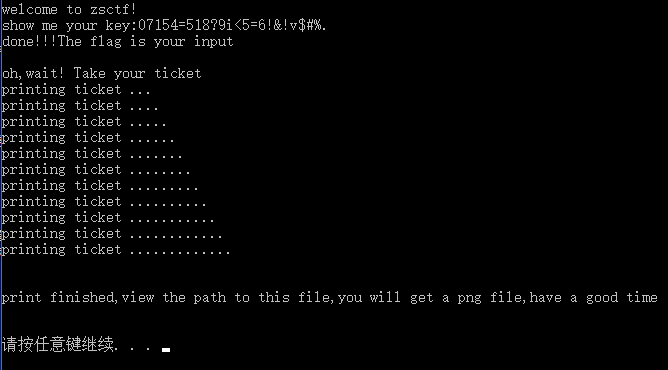


其中aP就是要翻译的字节码，通过翻译或者黑盒就可以得出VM的行为是将字串的每一位异或i++，i从0开始，即可推测出处理前的字串，然后返回主函数，进入判定函数

判定函数有点长，就不截图了，核心是一个迷宫，通过输入字串作为索引，将预设好的被索引字串作为移动标识，分析代码可得到地图为



而且程序中的移动存在循环，而且看到对输入字串的处理后可以知道实际上移动方向后面是跟着次数，即可得出最后正确的字串，然后再逆推过vm部分，就可以得到正确的输入字串，输入后得到



可以看到文件目录下出现了一个png文件，是一个二维码，



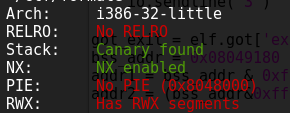
扫描可得

Flag:zsctf{07154=518?9i<5=6!&!v$#%.Docupa}

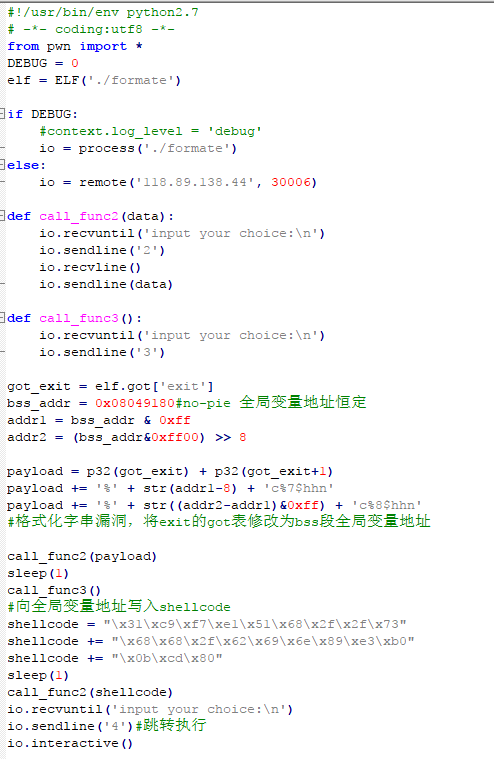
Pwn:

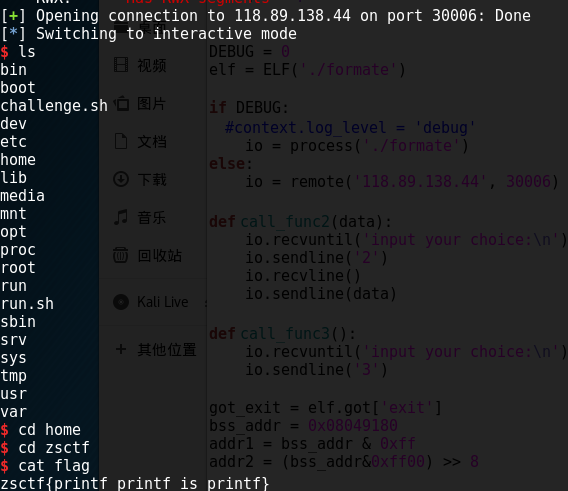
printf(75):

这题利用的是格式化漏洞进行任意地址读写，漏洞处于中的printf(src)，shecksec有

没有pie，思路就很容易构造了，因为src其实是一个全局变量，既地址固定

先找到src的固定地址，再构造格式化字串改写exit的got表为src地址，再将shellcode写入src处，最后引用exit即可getshell，下面直接给出exp

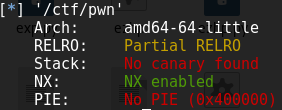




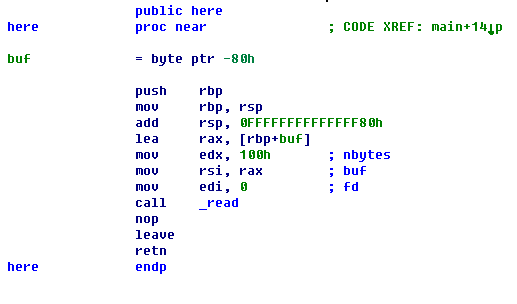
运行可得flag

WorX(100):

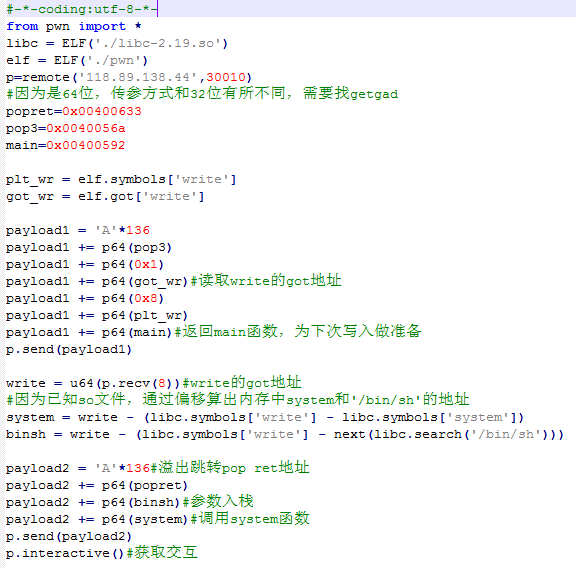
这题是一道栈溢出，checksec可以看到nx开启了，这里选择ret2libc



依然IDA，看见有read函数，

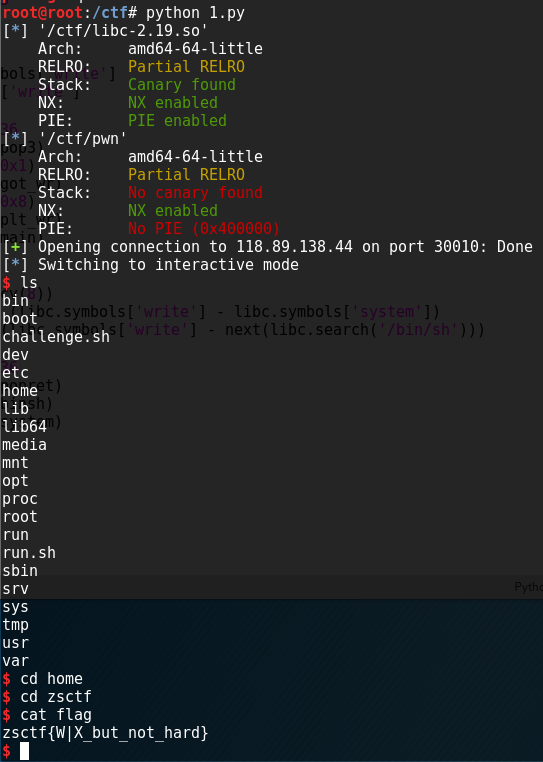


解决方法有很多，因为给出了lib文件，这里选择直接调用system(‘/bin/sh’)，exp如下:



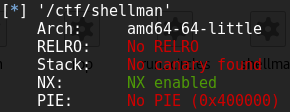
获取交互后

即可获得flag



Other-shellcode(150):

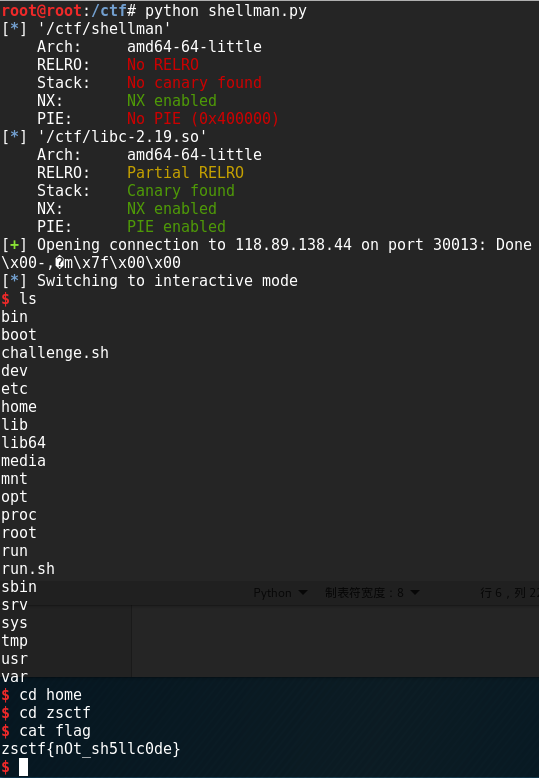
这题还是一道堆溢出，但调用的malloc，free等函数都是系统的，所以存在双链表冲突检验，先checksec



这里选择的方法是通过free伪造chunk之后控制一段可写区域， 利用可写区域泄露free函数的Got地址，通过给出来的lib文件计算system的地址,将system的地址写进之前控制的指向free的got，即可把free的got覆盖成system，然后直接调用的可以，具体exp如下：

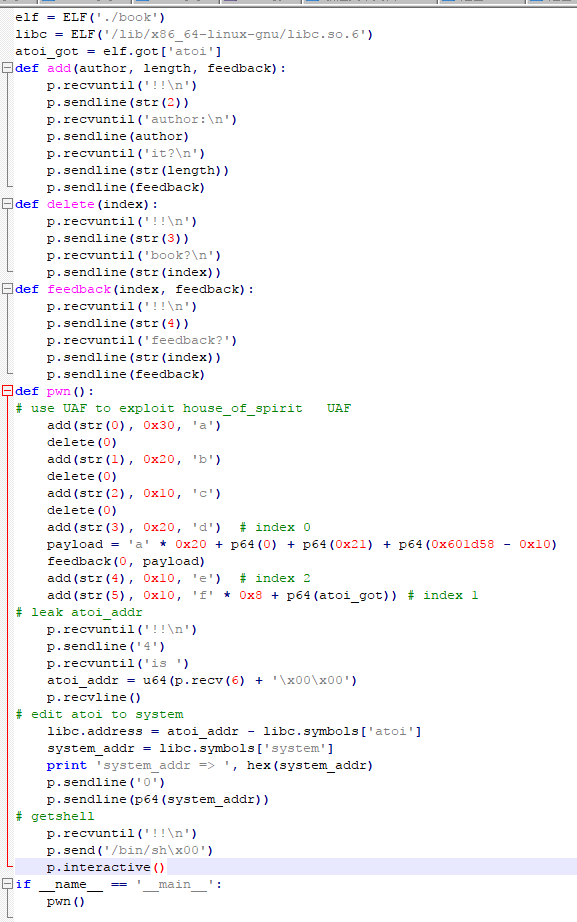


执行之后获取交互，即可获得flag

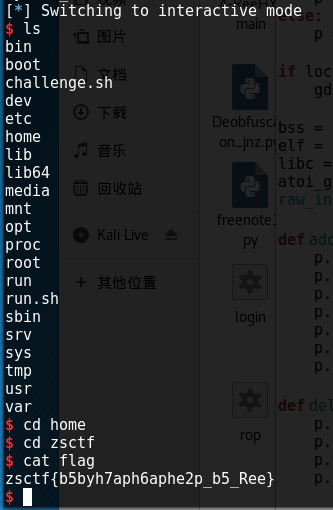


babyheap(125):

这题太多解法了。。。随便大佬秀操作，这里选择fastbin attack，IDA打开文件，可以了解程序的基本流程。通过fastbin attack可以达到任意地址堆申请，从而做到改写got表，这里选择将atoi函数got表覆盖成system，exp如下



运行后获得shell，得到flag



题解也到此结束了，感谢大家的支持(写题解比做题难受多了。。。。)