CTF密码学总结（二）

原创

沐一 · 林 于 2021-11-03 22:39:40 发布 2007 收藏 18
分类专栏： 笔记 文章标签： ctf

本文链接：https://blog.csdn.net/xiao__1bai/article/details/121132381


笔记 专栏收录该内容
18 篇文章 5 订阅
订阅专栏

目录

攻防世界之工业协议分析2：（16进制转字符、）

加密逻辑平铺类型

传统密码加密逻辑平铺：

2021年9月绿城杯，CRYPTO的[warmup]加密算法：（暴力破解、仿射密码、下标对应解密）

复杂加密类型

RSA直接给参数类型：

攻防世界之cr3-what-is-this-encryption：（RSA通用脚本解密）

攻防世界之OldDriver：（低加密指数广播攻击）

RSA明文密钥文件类型：

攻防世界之best_rsa：（明文密钥文件提取参数、RSA共模攻击、CTF-RSA-tool脚本修改）

攻防世界之RSA256：（）

RSA脚本逻辑加密类型：

2021年9月广州羊城杯，CRYPTO的BigRsa：（RSA多层模n加密、CTF-RSA-tool脚本修改、RSA模不互素）

2021年9月绿城杯，CRYPTO的RSA-1：（CTF-RSA-tool脚本修改、RSA通用脚本解密）

2021年9月绿城杯，CRYPTO的RSA-2：（费马分解算法）

2021年10月广东强网杯，CRYPTO的RSA AND BASE?：（排列组合算法、下标对应解密）

ECC椭圆曲线加密：

攻防世界之easy_ECC：（ECC加密）

LFSR反馈移位寄存器类型：

攻防世界之streamgame1：（CTF中的LFSR考点(一)、文件读取对齐的二进制）

攻防世界之streamgame2：（CTF中的LFSR考点(一)、文件读取对齐的二进制）

js类型加密

js逻辑平铺类型：

攻防世界之flag_in_your_hand1:（字符串中文含义暗示、冗余中锁定关键代码）

python类型逻辑加密

pyc文件反编译：

攻防世界之easychallenge：（源代码修改逻辑解密、）

传统密码类型解析

凯撒密码(24个字母)：

摩斯密码(只有01(无规则)或.-，空格或/做分隔符)：

CTF 密码学总结

出人意料的flag：

指在题目中获取到了flag，但是这个flag可能长得不像flag，或者flag还要经过进一步的脑洞处理，而不是常规的解密处理。

非预期行为：

指解题中出现与预想结果不符合的一系列非预期行为，这基本说明了在中间或前面存在其他自己还没分析的操作。

冗余中锁定关键代码：

从后往前看，就是确定比较关键对象，从该对象开始排除其他无关变量，一步步找出与该对象有关的其它变量，最后串起找到的所有相关变量，然后开始逆向分析。

题目类型总结：

题目描述暗示：

指题目给出的描述中有解题的大方向思路，以及对解题过程中出现的一些疑惑点的解释。

字符串中文含义暗示：

指解题中遇到带有中文暗示的字符串，这通常是出题者给的提示，抓住暗示重新梳理思路往往是正确的选择。

简单密码类型：

摩斯密码：

指解题中的密文涉及摩斯密码，摩斯密码的特征是以.-或01组成的，分隔符有空格或斜杠/。

url编码：

指解题中的密文涉及url编码，url编码的特征是使用 "%" 其后跟随两位的十六进制数来替换非 ASCII 字符。

传统base64解密：

指题目中密文是涉及base64加密，密文通常是4的倍数，基本元素是ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/和补充的'='

unicode解密：

指解题中密文涉及 unicode 解密，unicode现在已知的特征是ASCII（英文也是ASCII码）转unicode码时是 &# 前缀 +2 个 16 进制数，并用 ; 分隔。中文转unicode时是 \u + 4 个 16 进制数，中间没有间隔。

举例：1234 ----->1234 牛逼-------->\u725b\u903c

ASCII解密：

指解题中密文涉及ASCII码，需要转成字符。ASCII共128个可显示字符，范围是0~127或0~7F。不同通常每个ASCII码的间隔依据出题者来定。

栅栏密码：

指解题中密文涉及栅栏密码，因为栅栏密码有传统的矩阵型和 W 型，所以需要自己辨认。根据题目给的 key 分段来逐个尝试。

转轮机加密：

指解题中密文涉及转轮机加密，转轮机密文的特点是等长的分好组的乱序字母，原理是转齿轮把一个字母换成另一个来拼成一句话，所以会有多组密钥，但是只有一组密文。

格式举例，等长的分好组的字符串：

ZWAXJGDLUBVIQHKYPNTCRMOSFE

KPBELNACZDTRXMJQOYHGVSFUWI

键盘密码：

指解题中密文涉及键盘密码加密，键盘密码的特征是4~6个为一组，在键盘上呈包围态势，有明显的间隔。

16进制转字符：

指解题中密文是由16进制基本元素组成的，0~F或0~f，直接16进制转字符即可。

大小写字符提取密码：

指解题中题目给出的密文是一篇小说之类的，其中的内容里面并没有flag，所以内容含义没有意义。但是有一些字母的大小写形式与其他的不一样，这些不一样的字母提取出来就是一种加密类型了。

举例提取大写字符命令：

cat 1.txt | grep -o [A-Z] |tr -d '\n'

grep -o 只显示匹配到的字符串，tr -d 删除指定字符，不删除换行符的话就很长的打竖显示。

ROT13加密：

指解题中给出的密文是简单的数字或字母，但是难以判断加密类型，这时可以尝试用一下ROT13之类的字符移动来转换一下，由于是同一层面的数字和字母，所以通常可以装换出同层的具备传统密码特征的密文，就可以继续解题了。

poem codes诗歌加密：

指解题中密文涉及poem codes诗歌加密，poem codes诗歌加密的特征是诗歌 --> 关键词，原文 --> 参照顺序排列，密文 --> 按诗歌关键词对原文映射取值。

仿射密码：

指解题中密文涉及仿射密码，仿射密码加密的特征是一种替换密码，它是一个字母对一个字母的。字母系统中所有字母都藉一简单数学方程加密。

复杂密码类型：

ECC加密：

是一种建立公开密钥加密的算法，基于椭圆曲线数学的椭圆曲线密码学。

CTF-RSA-tool脚本修改：

指解题中RSA类型是CTF-RSA-tool工具中内嵌类型的一种变形，无法直接用工具生成答案，需要重CTF-RSA-tool中锁定对应的代码，抽出来自己修改才行。

factor_N.py：是对应的解密逻辑封装，解出p、q。

RSAutils.py：是对应的输出结果部分，解决精度和字节串转为字符串问题。

RSA通用脚本解密：

指简单的RSA题目中，直接给出或求出p、q、e、c的值，不需要再用N来大数分解，所有参数都有了就可以直接运行RSA解密脚本解题了。

RSA多层模n加密：

指解题中给了多个模，且都是2048bit 4096bit等无法正面分解的数。需要使用欧几里得算法求取模之间的公约数，也可以直接用p1 = gmpy2.gcd(n1, n2) 这个系统提供的欧几里得封装函数。根据欧几里德算法算出的p之后，再用n除以p即可求出q，由此可以得到的参数有p、q、n、e，再使用常规方法计算出d，即可破解密文。

注意：

这和RSA模不互素很像，但是模不互素只用一个n加密，另一个n是给你求公因子而已。

RSA模不互素：

指解题中RSA类型给了多个模n，且都是2048bit 4096bit等无法正面分解的数，与RSA多层模n加密的区别就是模不互素只用一个n加密，另一个n是给你求公因子而已。

低加密指数广播攻击：

指解题中RSA类型给了多个n、c、e，广播指我们需要将一份明文进行多份加密，但是每份使用不同的密钥，密钥中的模数n不同但指数e相同且很小，我们只要拿到多份密文和对应的n就可以利用中国剩余定理进行解密。

明文密钥文件提取参数：

指解题中题目给了RSA的明文和密钥文件，可能是多个，所以无法直接用CTF-RSA-tool解题（CTF-RSA-tool好像只能输入一对明文密钥文件）。

密钥文件和明文文件读取出对应数字的方法，提取n、e都是用Crypto.PublicKey.RSA模块，再抽取对应的n,c属性的。

提取加密密文c，则是直接二进制读取文件后用Crypto.Util.number模块的bytes_to_long函数转二进制流为数字的。

RSA共模攻击：

指解题中题目给了两个相同的n，不同的e、c，来加密密文。

文件相关类型：

16进制文件流：

指解题中给的密文或者附件中的内容是16进制的文件流，这需要16进制转字符后再base64解码后才能发现是类似于抓包的文件流。所以最开始的16进制流就是文件流，扔入winhex中保存为指定后缀即可。

文件流加密：

指解题中题目给的文件扔入winhex工具中16进制显示的内容，文件头等信息不符合题目给的后缀或不符合题目的文件类型描述。这时就要考虑是否在文件流层面上进行了加密处理，通常是简单的移位处理。

文件读取对齐的二进制：

指解题中需要从文件中读取二进制数，二进制数有时是不能直接从文件中复制粘贴转换的，因为有时候会有乱码显示。可以用前面积累的 base64 编码的代码写一串从文件中读取对齐的二进制的代码。

(在线转换都会省略最开头的0导致结果位数错误，进而导致结果错误，所以自己要注意)

```
f = open('5key','rb') #以二进制格式打开文件

content = f.read() #读取的是\xhh类型的十六进制

key=['{:0>8}'.format(str(bin(i)).replace('0b','')) for i in content] #从base64编码汲取的经验，二进制8位对齐。

print(''.join(key)[:19])
```

算法类总结：

费马分解算法：

指RSA题目类型中模n是4个p、q的混合乘积。

费马分解算法的特征就是n是4个数的乘积，分解n之后我们会得到p、p1、q、q1四组隔开的排列组合，但是我们的脚本可以把组合限定成p * q1， p1 * q,和p * q ， p1 * q这样。

然后通过欧几里得算法求公因子的封装函数gcd(pq1,pq) = p、gcd(p1q,p1q) = q 求出两组各一个数，然后就可以求出φ(n)=φ(p)·φ(p1)·φ(q)·φ(q1)=(p−1)·(q−1)·(p1−1)·(q1−1)了。

排列组合算法：

指解题中加密表单缺少了多个字母，但是这个多个字母有多个组合，因为无法确定是那个组合才能拼凑出正确的加密表单，所以需要用排列组合算法全部穷举出来。

密码学脚本类总结：

ASCII转字符脚本：

指解题中遇到长串ASCII码形式，需要转字符，但是一个个转太麻烦，又没有在线的长串ASCII码转字符网站。且给出的长串ASCII码的间隔依情况而定，如：/119/101/，这时需要自己根据对应间隔写出批量转换脚本。

源代码修改逻辑解密：

指解题中在有较完整源代码的情况下代码逻辑比较明朗，且可以逆向。这时需要充分利用有源代码的优势来在源代码中修改处逆向逻辑，不要自己从头到尾另写一份。

暴力破解：

指解题中对每个密文在ASCII的32~127中逐个正向加密对比，找到加密后与密文对应的字符后取chr(i)。

下标对应解密：

指解题中遇到单表替换类型，类似于仿射密码或base家族变形表单加密，正向字母表中每个字母的值使用一个简单的数学函数映射到对应的值。

对于这种加密映射的单表替换型我们可以反着来把正向表单全部加密得出反向表单，密文在反向表单中的下标等同与flag在正向表单的下标，就是下标等价，类似于base64表单替换。

单独的密文类型（优先使用ciphey工具）

多层传统加密混合：

Bugku的密码学的入门题/.-：（摩斯密码、url编码、出人意料的flag）

摩斯密码是以.-或01组成的，分隔符有空格或斜杠/，所以直接扔去摩斯密码在线解密即可。

网址：https://www.bejson.com/enc/morse/

答案差不多了，却被%u7b和%u7d卡住了，猜想是{ }的url编码，{的url编码是%7B，}的url编码是%7D

所以去掉u符号（这里u符号应该是什么十六进制之类的，我也不知道）

直接得到flag：

FLAG{D3FCBF17F9399504}

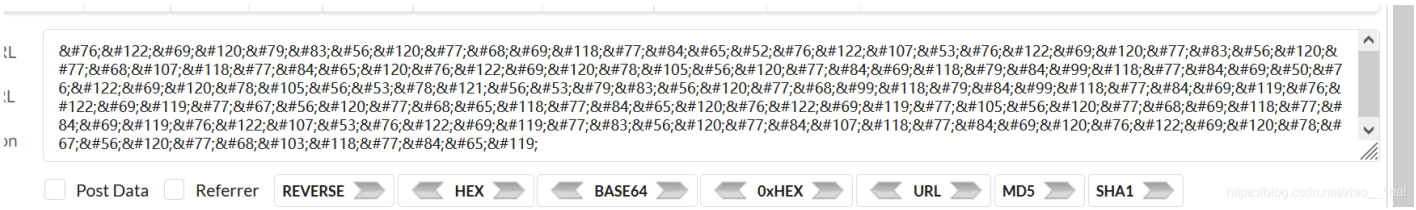
结果全部要小写：

flag{d3fcbf17f9399504}

攻防世界之混合编码：（base64解密、unicode解密、ASCII转字符脚本、传统base64解密、ASCII解密）

下载附件，打开，发现两个等号和19azA~Z的典型base64编码型，直接base64转换：

转了一堆&#出来，根据以前的做题经验，猜unicode或hex：



一开始用了十六进制来转，转了个四不像出来，后来发现转错了，unicode在线解码网址：

http://www.jsons.cn/unicode/

这三个数的看着像ASCII，因为题目暗示混合编码，直接转换看看：ASCCII表对照法：

0110 1101	0155	109	0x6D	m	小写字母m
0110 1110	0156	110	0x6E	n	小写字母n
0110 1111	0157	111	0x6F	o	小写字母o
0111 0000	0160	112	0x70	p	小写字母p
0111 0001	0161	113	0x71	q	小写字母q
0111 0010	0162	114	0x72	r	小写字母r
0111 0011	0163	115	0x73	s	小写字母s
0111 0100	0164	116	0x74	t	小写字母t
0111 0101	0165	117	0x75	u	小写字母u
0111 0110	0166	118	0x76	v	小写字母v
0111 0111	0167	119	0x77	w	小写字母w
0111 1000	0170	120	0x78	x	小写字母x
0111 1001	0171	121	0x79	y	小写字母y
0111 1010	0172	122	0x7A	z	小写字母z
0111 1011	0173	123	0x7B	{	开花括号
0111 1100	0174	124	0x7C	|	垂线
0111 1101	0175	125	0x7D	}	闭花括号

ASCII转字符脚本法：（这里因为string.split切片后返回的是列表，所以可以直接用索引获取。）

import re

```
r="/119/101/108/99/111/109/101/116/111/97/116/116/97/99/107/97/110/100/100/101/102/101/110/99/101/119/
```

```
r=re.split("/",r)
#print(r)
flag=""
for i in range(1,len(r)):
flag+=chr(int(r[i]))
print(flag)
```

单层传统加密：

Bugku crypto之聪明的小羊：（题目描述暗示、栅栏密码）

好的，传统栅栏密码，下面是我以前的笔记：

所谓栅栏密码，就是把要加密的明文分成N个一组，然后把每组的第1个字连起来，形成一段无规律的话。 不过栅栏密码本身有一个潜规则，就是组成栅栏的字母一般不会太多。（一般不超过30个，也就是一、两句话）

传统栅栏密码(矩阵行列，密钥是行数)：

假如有一个字符串：123456789

取字符串长度的因数进行分组，假如key=3

1 2 3 \\分组情况，每三个数字一组，分为三组

4 5 6

7 8 9

然后每一组依次取一个数字组成一个新字符串：147258369 \\加密完成的字符串

解题：

试一般的栅栏密码,取5为矩阵行数,得到" cyperrocaegireeol} eahfocec gnbip不正确，取5为矩阵列数,得到" cebgccfe en eohplprgecrayoii aoreg”,也不正确，除了常规的栅栏密码,还有

由题目描述可知分两组：

fa{fe13f590

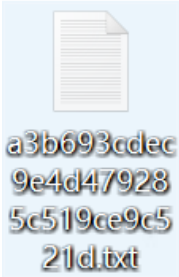
lg6d46d0d0}

那么答案很明显了，上一个下一个即可得flag：

flag{6fde4163df05d900}

攻防世界之转轮机加密：（转轮机加密、）

下载附件：



好了，记住了，以后这个内容格式的就是转轮机加密了：

```
1:  < ZWAXJGDLUBVIQHKYPNTCRMOSFE <
2:  < KPBELNACZDTRXMJQOYHGVSFUWI <
3:  < BDMAIZVRNSJUWFHTEQGYXPLOCK <
4:  < RPLNDVHGFCUKTEBSXQYIZMJWAO <
5:  < IHFRLABEUOTSGJVDKCPMNZQWXY <
6:  < AMKGHIWPNYCJBFZDRUSLOQXVET <
7:  < GWTHSPYBXIZULVKMRAFDCEONJQ <
8:  < NOZUTWDCVRJLXKISEFAPMYGHBQ <
9:  < XPLTDSRFHENYVUBMCQWAOIKZGJ <
10: < UDNAJFBOWTGVRSCZQKELMXYIHP <
11:  < MNBVCXZQWERTPOIUYALSKDJFHG <
12:  < LVNCMXZPQOWEIURYTASBKJDFHG <
13:  < JZQAWSXCDERFVBGTYHNUMKILOP <

密钥为：2,3,7,5,13,12,9,1,8,10,4,11,6
密文为：NFQKSEVOQOFNP
```

转轮机加密格式

原理就是转齿轮把一个字母换成另一个，直接上一个修改后的大佬脚本：

rotor = [#这里是要输入的转轮机原始字符串

"ZWAXJGDLUBVIQHKYPNTCRMOSFE", "KPBELNACZDTRXMJQOYHGVSFUWI",

"BDMAIZVRNSJUWFHTEQGYXPLOCK", "RPLNDVHGFCUKTEBSXQYIZMJWAO",

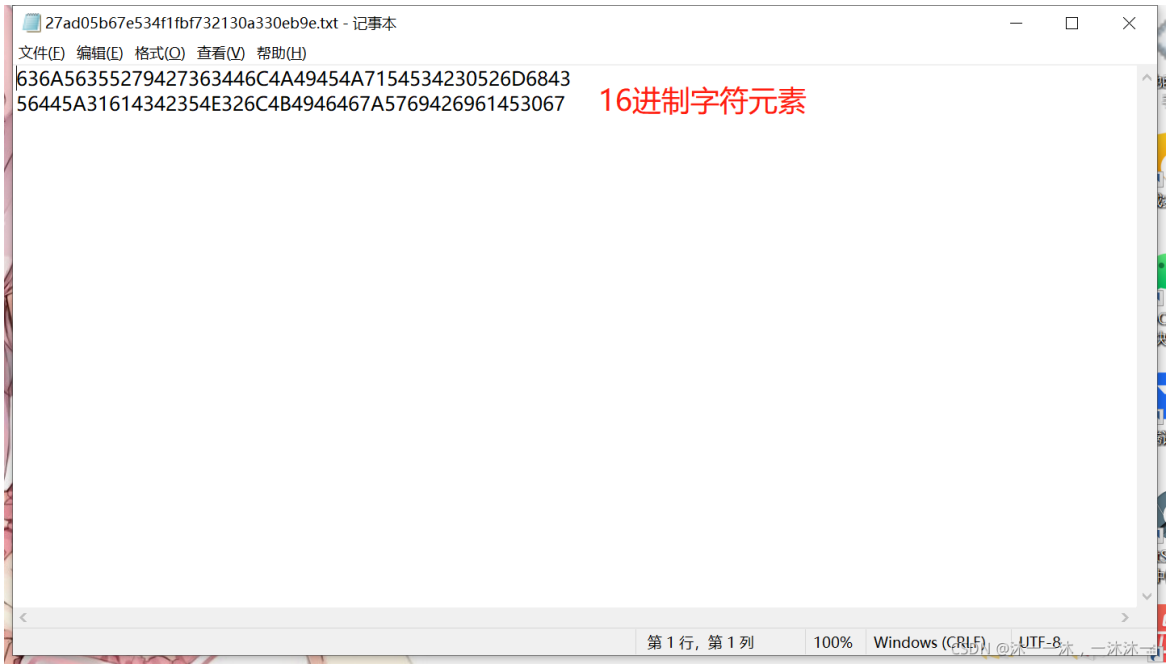
"IHFRLABEUOTSGJVDKCPMNZQWXY", "AMKGHIWPNYCJBFZDRUSLOQXVET",

```
"GWTHSPYBXIZULVKMRAFDCEONJQ", "NOZUTWDCVRJLXKISEFAPMYGHBQ",
"XPLTDSRFHENYVUBMCQWAOIKZGJ", "UDNAJFBOWTGVRSCZQKELMXYIHP",
"MNBVCXZQWERTPOIUYALSKDJFHG", "LVNCMXZPQOWEIURYTASBKJDFHG",
"JZQAWSXCDERFVBGTYHNUMKILOP"
]
cipher = "NFQKSEVOQOFNP" #这是要输入转轮机密文
key = [2,3,7,5,13,12,9,1,8,10,4,11,6] #这是要输入转轮机密钥
tmp_list=[]
for i in range(0, len(rotor)):
tmp=""
k = key[i] - 1
for j in range(0, len(rotor[k])):
if cipher[i] == rotor[k][j]:
if j == 0:
tmp=rotor[k]
break
else:
tmp=rotor[k][j:] + rotor[k][0:j]
break
tmp_list.append(tmp)
# print(tmp_list)
message_list = []
for i in range(0, len(tmp_list[i])):
tmp = ""
for j in range(0, len(tmp_list)):
tmp += tmp_list[j][i]
message_list.append(tmp)
print(message_list)
def spread_list(lst):
for item in lst:
if isinstance(item,(list,tuple)):
```

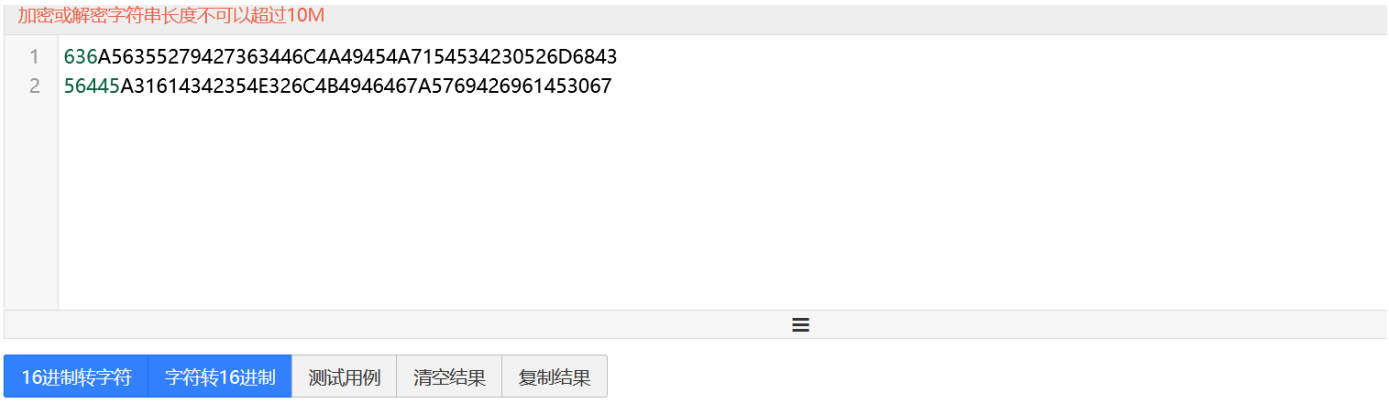
```
yield from spread_list(item)

else:

yield item

pass

if __name__ == '__main__':

for i in spread_list(message_list):

print("*"*25)

print(i) #在多个输出中查找有语义的字符串即为flag内容
```

攻防世界之告诉你个秘密：（16进制转字符、键盘密码）

下载附件，是个.txt文件，打开：



虽然没有f，但是看起来就是16进制的基本组成单位0~F，既然是十六进制就不用先十六进制转字符串，目前接触的也只有十六进制转字符串了：



转出来的字符看起来像是base64元素

转完之后是大小写字母混合和数字，虽然没有+/，但看起来也的确是base64的基本组成单位了，base64解码一下：

得出的东西是一个四不像，查了很多资料，大部分说是键盘围起来的密码，而且就叫键盘密码。

(唯一的暗示可能就是空格吧，附上别人的一句话：

看到明□张胆的空格,再瞅□下键盘,发现是键盘密码,)

r5yG lp9l BjM tFhB T6uh y7iJ QsZ bhM

最后flag就是这些字符在键盘中围起来的键，但是要大写才能提交：

TONGYUAN

攻防世界之sherlock：（大小写字符提取密码）

下载附件，是一个txt文档，内容是一篇小说。一开始我以为flag藏在关键字里，我还用百度翻译一个个看内容，现在回想起来真的太傻了，查了资料才发现字符中是有异或点的，大写字母就是要提取出来分析的地方：

title: the adventures of sherlock holmes

author: sir arthur conan doyle

release date: march, 1999 [ebook #1661]
[most recently updated: november 29, 2002]

edition: 12

language: english

character set encoding: ascii

*** start of the project gutenberg ebook, the adventures of sherlock holmes ***

(additional editing by jose menendeZ) 有不同于其它字符的大写字符

参考了别人的命令写了自己的提取大写shell命令：

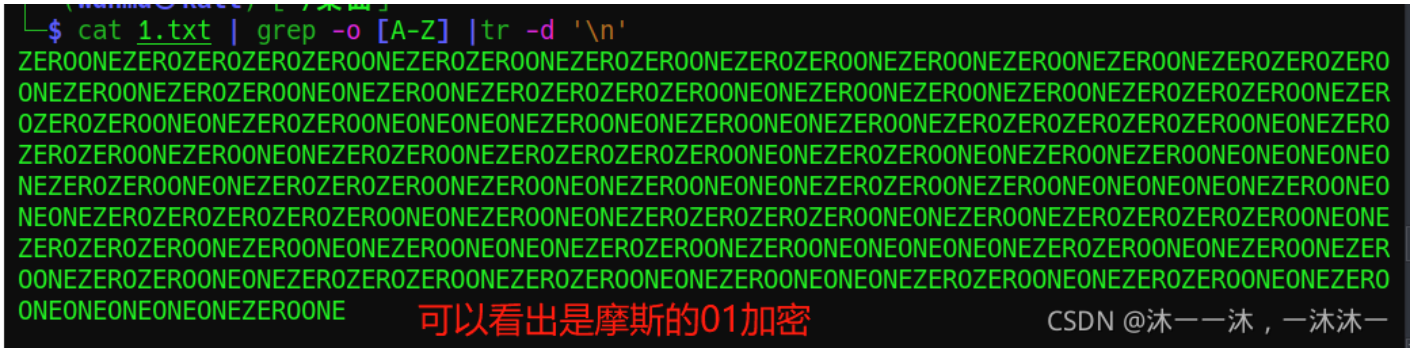
cat 1.txt | grep -o [A-Z] |tr -d '\n'

其中：

grep -o 只显示匹配到的字符串

tr -d 删除指定字符，不删除换行符的话就很长的打竖显示。

结果：



然后可以发现都是ZERO和ONE的单词，不是二进制字符串就是摩斯密码，可是摩斯密码要空格，这里没有，所以是二进制字符串。

然后就是自己写python脚本转换01率，一开始用for语句卡了一下，后来直接换while语句：

```
key1="ZEROONEZEROZEROZEROZEROONEZEROZEROONEZEROZEROONEZEROZEROONEZEROONI
flag=""
i=0
while i<len(key1):
if key1[i]=='Z'and key1[i+1]=='E'and key1[i+2]=='R'and key1[i+3]=='O':
i+=4
```

flag+='0'

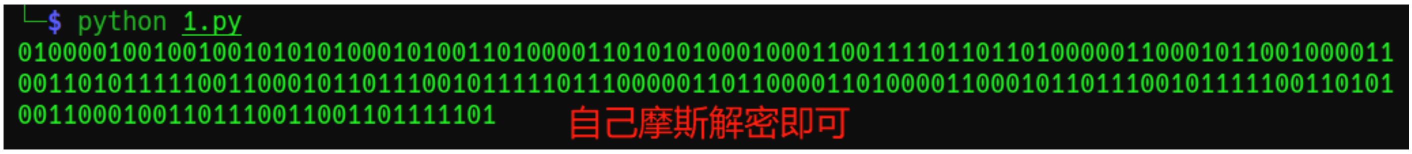
else:

flag+='1'

i+=3

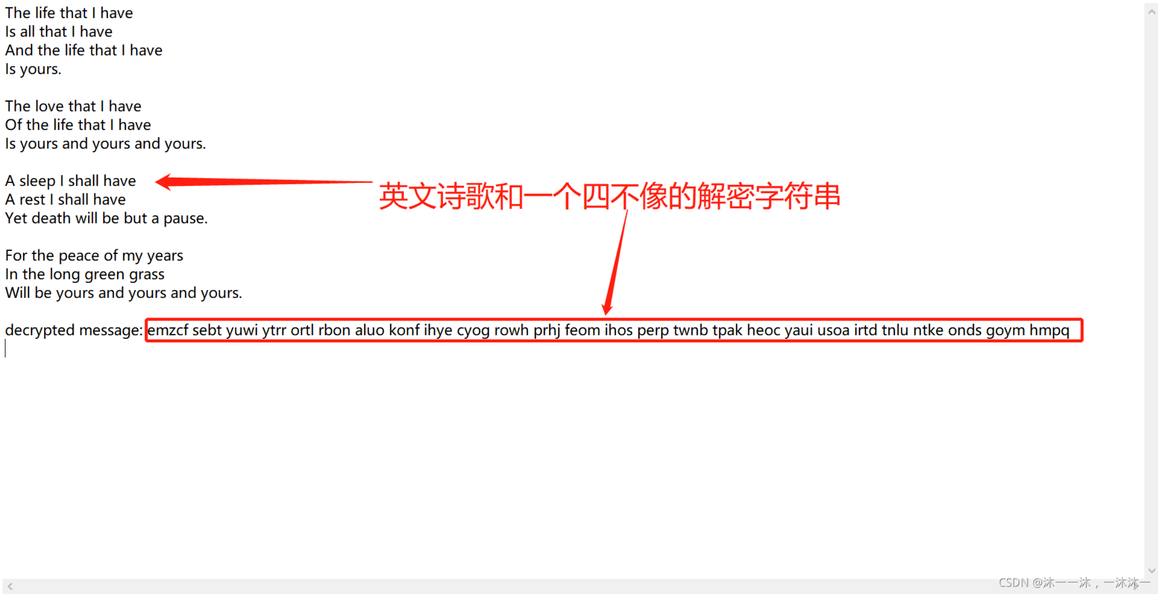
print(flag)

结果：



攻防世界之Decrypt-the-Message：（poem codes诗歌加密、）

下载附件，是个.txt文件，内容是诗歌，下面是一行四不像的英文，后来发现是加密后的密文：



诗歌类的加密：

一开始还以为是唐伯虎点秋香中句子开头组成实际内容，结果发现不是。查了查资料，是poem codes加密。

然后，我也不知道诗歌中关键词在哪，而且题目诗歌内容也太长了，所以只能用github的脚本了：

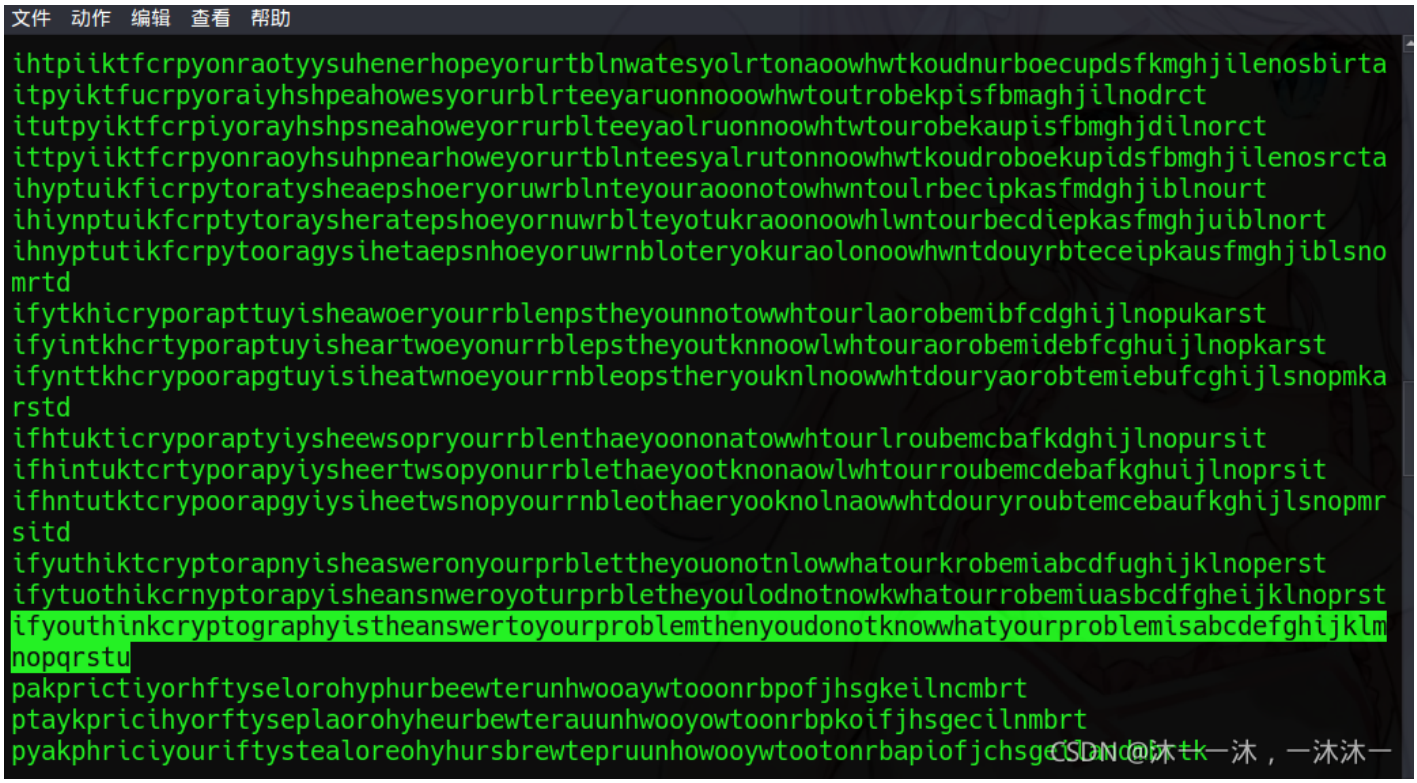
(单独复制poemcode.py是会报错的，因为文件中有其他依靠)：

git clone git://github.com/abpolym/crypto-tools

用法：(python2，ctfpoem是诗歌，ctfcip是加密密文)

python2 poemcode.py examples/2/ctfpoem examples/2/ctfcip

结果，在众多输出中找到通顺的句子，其实后面我也不知道后面开头的单词合不合理，英语太菜了~(哭 ~)



文件相关类型：

攻防世界之你猜猜：（16进制转字符、传统base64解密、16进制文件流）

下载附件，是一个.txt文件，打开，数字和字母：

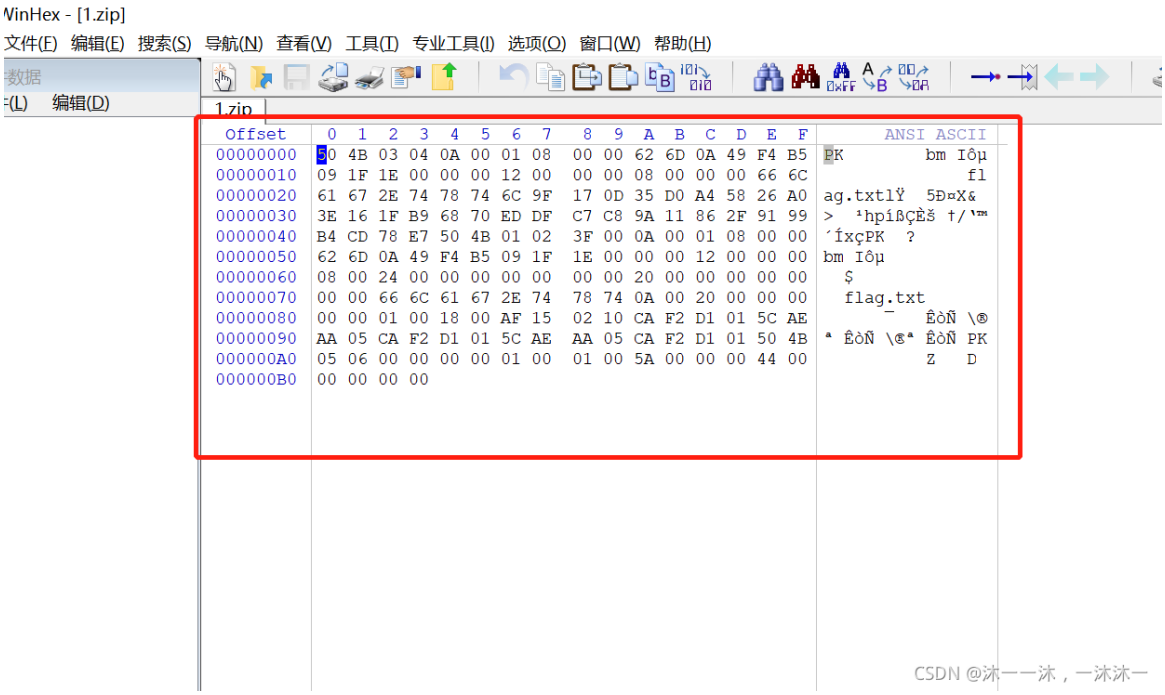
504B03040A0001080000626D0A49F4B5091F1E0000001200000008000000666C61672E7478746C9F170D35

往base家族想，字母只有A~F，base16解码：

嗯~好熟悉，后来想起来是web题中bp抓过的数据，查了资料发现是一个zip文件的16进制数据，竟然这么短也能组成zip的数据，也是开了眼界，zip里面也可以看到有个flag.txt文件。

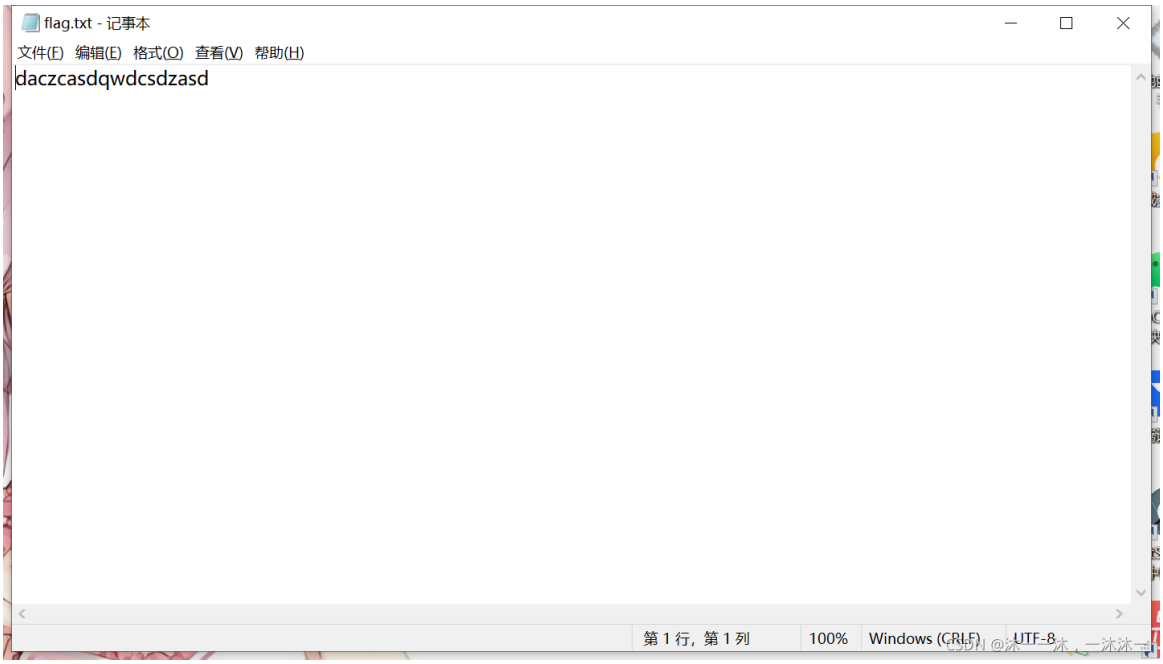
一开始还直接修改成.zip后缀，真是太天真了，这是把hex数据放入txt文件啊。

打开winhex64复制粘贴成zip文件即可：



解密的时候发现需要密码，也不是zip伪加密。嗯~查了资料，上网下载了Ziperello来爆破密码，密码组成只能一个个试了：

拿到了密码123456，解压得到flag：



攻防世界之banana-princess：（ROT13加密、文件流加密）

下载附件，是一个PDF文件，打不开，题目英文提示香蕉原则，好吧并没有什么用。(^ ~ ^)用记事本打开看一下内容：

看起来像是一个文件流：

0000001224 00000 a
0000001444 00000 a
0000036608 00000 a
0000040339 00000 a
0000040400 00000 a
0000040618 00000 a
0000207067 00000 a
0000215381 00000 a
0000310739 00000 a
0000000576 00000 a
genvyre
<</Fvmr 18/Ebbg 5 0 E/Vasb 3 0 E/VQ[<RQ0R7836S40NN64NN7RPP6S9Q25NO393><8S7R255035895R408O19PP23O(
fgnegkers
0
%%RBS

17 0 bow
<</Svygre/SyngrQrpbqr/V 87/Yratgu 76/F 36>>fgernz
u辭``鄱``猲```覷`?,□U<a(s`皵□o癵`~繵???uSL?□扯□循f??佞髖4#□?□0 vN
?
raqfgernz
raqbow

内容像是文件流

raqfgernz
raqbow
9 0 bow
<</OvgfCrePbzcbarag 8/PbybeFcnpr/QrivprETO/QrpbqrCnezf<</Oyraq 1/PbybeGenafsbez 1/Pbybef 3/Pbyhzaf 690/UFı
?? □Nqbor q □ ? ? □□□□□□"□#-)!")22/&)4+-</+4?6::O:6?PTTTTTPTTTTTTTTTTTTTTTTTTTT□□□□ □□6)66TTNTTTTT
□□□ □□ □ □□□□□□□
□□ □□□ □□□ □ □ □□□!□1□N□Dnd"2?□O懿绷oe□#$%掷狃□□
□□□□□&'()*3456789:PQRSTUVWEFGHIJKLMpqrstuvwfghijklm們剝唶垮娓跋枟槐殺￥ウЖ┆炒刀犯购旅呐魄壬恃矣哉肿刭阝 沅三
□□□□□□□□□□!"#$%&'()*123456789:NOPQRSTUVWDEFGHIJKLMnopqrstuvwdefghijklm岂懺庠噜塌憭摄晼榱橾、 ￥ウЖ┆辈灺
□8?□堉]'T洊噊鞑?=q憮sB#烟??p币□□焓□ □訣l?R頖B碰□<'□/ゥ赸
祋@嗣uv十?>□□q□=谆饨□j□a粕?簟Q□Q@QQ□Q@QQ□Q@QQ□Q@QQ□Q@QQ□Q@QQ□Q@QQ□Q@QQ□Q@R恸槂6g
??□??<;f□l
?\疵闫侮?甐x\踿鋉□〒続l+u Q?嵛?4蔚□┗€ 刮.□ 8狍尾焼3! 'Cb 閟餫□□=□??鰲疽9??X吪碁?戶彝嶮U釐??噆勦?f拒鱠
/c?T[<所□`礅蛸□Y鲩 ???歃.v+鸜A{Z?{(鮫6鈺 E 崙S\lwu□舫?灕J??趁W魨咋檸籃緩媜M
□ 幀蠵⊕\异鸏.鯣?凑l?怕=ac稅/^□v^瞱?"`:y .?T;剜j*?]-gE?K缌?:谲\?艭]"蚑?\使Yy-世5箩??d&? □?窜^ ?穖蚪岫$€\髯矗季粟L蠨穱
m□i□PV疆2m??U逞d殫螰□啉旐?? 7d捋K7'玥p陵瀹 Q??D硐x\伂□□H晲U荂?M3磐螫諨a秐嚜 P d:氠3厝?lxA?8巏卑倏$□u
Y蝨□i48哐;U□ (=DrL??fH樫s<啷?婺 蛙b~?RnH扭 4TPs〆./$1? 筋}€\j牶R@k鱊□X熤鏄?Q|□魶f6#濿籀獠□醾TO鸍?鋼?□恆Z荊3
菇崌z陕鋎L嫞?凑9氤軶5嵽呢抎nn媳□?父M; □?莉蚤舍靉z?儹y綟d7簣阵?耔劂agm□挹遛□M 骯@ V蝗Z□溏蛐□ ?]w7
鈺5?Z= 8□□□?硒?□jNg猩zRq城B、 □狧\ ?□炴闚□璅□ 犣韇?7嚘èw驷币Fgmぉlf儘澫□关赖 剜 工□蟢hA唵□垮!捏7谲徽姤┾
□攉瘽?k □w ??&: ??湛獌U婤婑T疒P(
g恤?桝?"鶭峨6Fkz○??娌Z5 u堪锎□苜尔諂=□h □□Z$z???u菸9瞙\☉d%\酟·M丞煟?畔□ □a蜷q□堍徒h□孰N?碑肿w迎鏽□鄴 苗

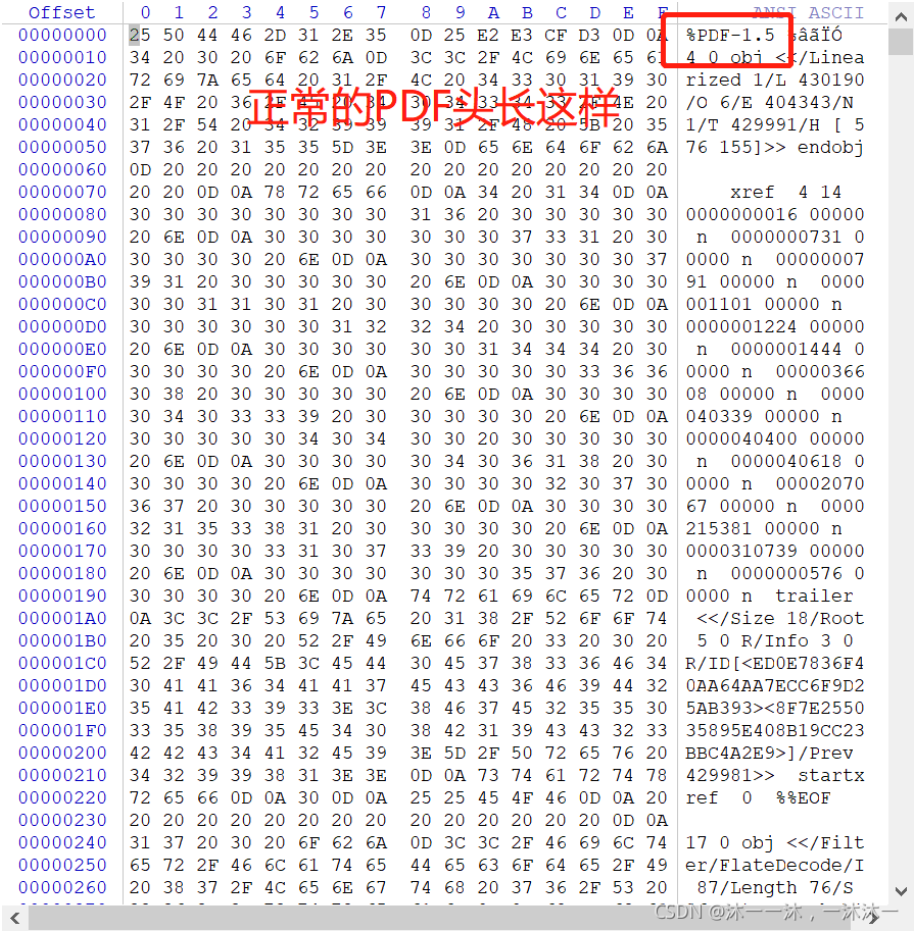
扔到winhex64中再看一下，嗯~其实一开始我也看不出有什么不妥，红框那里是查了资料说与正常PDF有不同。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ANSI ASCII
00000000	25	43	51	53	2D	31	2E	35	0D	25	E2	E3	CF	D3	0D	0A	%CQS-1.5 %âãÏÓ
00000010	34	20	30	20	62	6F	77	0D	3C	3C	2F	59	76	61	72	6E	4 0 bow <</Yvarn
00000020	65	76	6D	72	71	20	31	2F	59	30	34	33	30	31	39	30	evmrq 1/Y 430190
00000030	2F	42	20	36	2F	52	20	34	30	34	33	34	33	2F	41	20	/B 6/R 404343/A
00000040	31	2F	47	20	34	32	39	39	31	2F	55	20	5B	20	35	20	1/G 429991/U [5
00000050	37	36	20	31	35	35	5D	3E	3E	0D	72	61	71	62	6F	77	76 155]>> raqbow
00000060	0D	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	
00000070	20	20	0D	0A	6B	65	72	73	0D	0A	34	20	31	34	0D	0A	kers 4 14
00000080	30	30	30	30	30	30	30	30	31	36	20	30	30	30	30	30	0000000016 00000
00000090	20	61	0D	0A	30	30	30	30	30	30	30	30	37	33	31	20	30	a 0000000731 0
000000A0	30	30	30	30	20	61	0D	0A	30	30	30	30	30	30	30	37	0000 a 00000007
000000B0	39	31	20	30	30	30	30	30	20	61	0D	0A	30	30	30	30	91 00000 a 0000
000000C0	30	30	31	31	30	31	20	30	30	30	30	30	20	61	0D	0A	001101 00000 a
000000D0	30	30	30	30	30	30	31	32	32	34	20	30	30	30	30	30	0000001224 00000
000000E0	20	61	0D	0A	30	30	30	30	30	30	31	34	34	34	20	30	a 0000001444 0
000000F0	30	30	30	30	20	61	0D	0A	30	30	30	30	30	33	36	36	0000 a 00000366
00000100	30	38	20	30	30	30	30	30	20	61	0D	0A	30	30	30	30	08 00000 a 0000
00000110	30	34	30	33	33	39	20	30	30	30	30	30	20	61	0D	0A	040339 00000 a
00000120	30	30	30	30	30	34	30	34	30	30	20	30	30	30	30	30	0000040400 00000
00000130	20	61	0D	0A	30	30	30	30	30	34	30	36	31	38	20	30	a 0000040618 0
00000140	30	30	30	30	20	61	0D	0A	30	30	30	30	32	30	37	30	0000 a 00002070
00000150	36	37	20	30	30	30	30	30	20	61	0D	0A	30	30	30	30	67 00000 a 0000
00000160	32	31	35	33	38	31	20	30	30	30	30	30	20	61	0D	0A	215381 00000 a
00000170	30	30	30	30	33	31	30	37	33	39	20	30	30	30	30	30	0000310739 00000
00000180	20	61	0D	0A	30	30	30	30	30	30	30	35	37	36	20	30	a 0000000576 0
00000190	30	30	30	30	20	61	0D	0A	67	65	6E	76	79	72	65	0D	0000 a genvyre
000001A0	0A	3C	3C	2F	46	76	6D	72	20	31	38	2F	45	62	62	67	<</Fvmr 18/Ebbg
000001B0	20	35	20	30	20	45	2F	56	61	73	62	20	33	20	30	20	5 0 E/Vasb 3 0
000001C0	45	2F	56	51	5B	3C	52	51	30	52	37	38	33	36	53	34	E/VQ[<RQ0R7836S4
000001D0	30	4E	4E	36	34	4E	4E	37	52	50	50	36	53	39	51	32	0NN64NN7RPP6S9Q2
000001E0	35	4E	4F	33	39	33	3E	3C	38	53	37	52	32	35	35	30	5NO393><8S7R2550
000001F0	33	35	38	39	35	52	34	30	38	4F	31	39	50	50	32	33	35895R408O19PP23
00000200	4F	4F	50	34	4E	32	52	39	3E	5D	2F	43	65	72	69	20	OOP4N2R9>]/Ceri
00000210	34	32	39	39	38	31	3E	3E	0D	0A	66	67	6E	65	67	6B	429981>> fgnegk
00000220	65	72	73	0D	0A	30	0D	0A	25	25	52	42	53	0D	0A	20	ers 0 %%RBS
00000230	20	20	20	20	20	20	20	20	20	20	20	20	20	20	0D	0A	
00000240	31	37	20	30	20	62	6F	77	0D	3C	3C	2F	53	76	79	67	17 0 bow <</Svyg
00000250	72	65	2F	53	79	6E	67	72	51	72	70	62	71	72	2F	56	re/SyngrQrpbqr/V
00000260	20	38	37	2F	59	72	61	74	67	75	20	37	36	2F	46	20	87/Yratgu 76/F

题目给的是PDF文件，但是这里却是CQS头，按理来说应该是PDF头，所以这里有一个字符之间的转换。

正常PDF头：

Adobe Acrobat pdf %PDF-1.xx （1.xx是版本号）



到这里我的思想就和别人不一样了，别人是思考%CQS-1.5和%PDF-1.x的关系。而我的第一反应是%CQS-1.5是一个没学过的文件头。。。。。毕竟没学过的文件头大把，，所以才有这么菜的自己。(哭~)

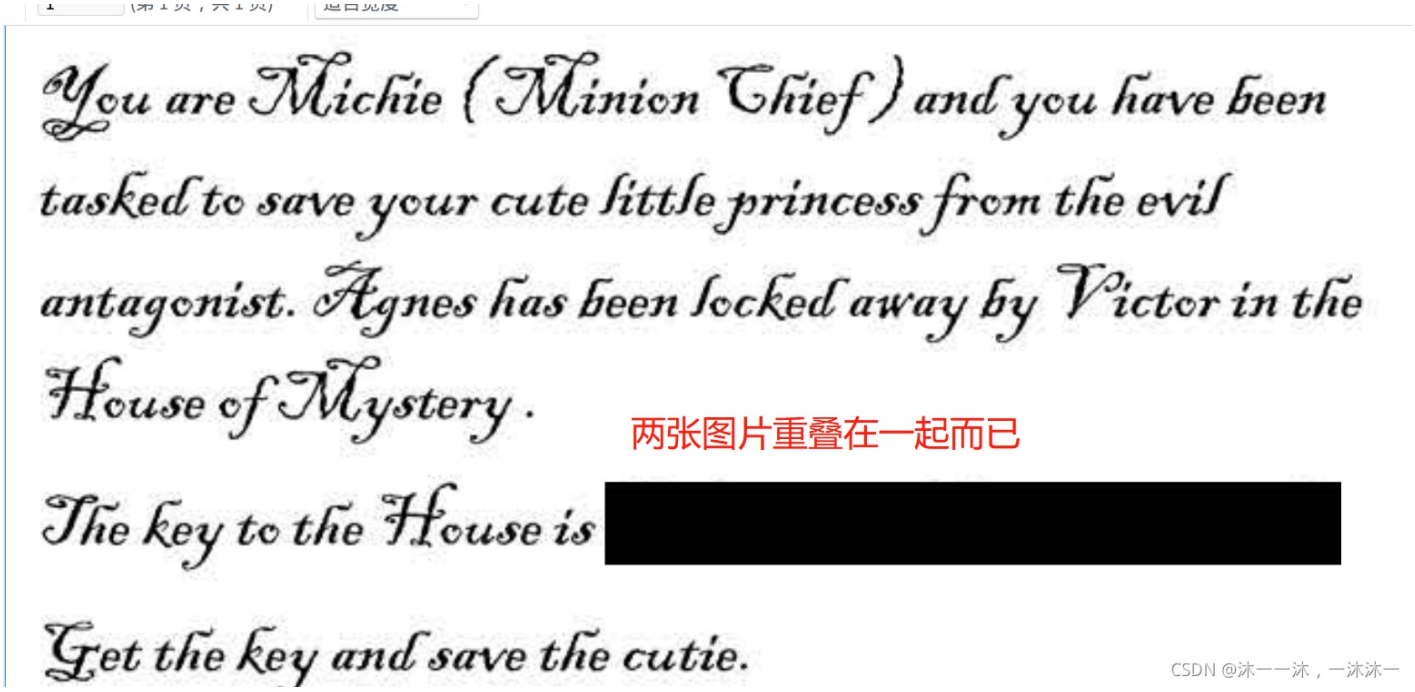
然后要问一个字母和另一个字母的关系，加密中能把一个字母加密后是另一个字母的目前学过的有培根，ROT13，24字母移动的凯撒密码。ROT13和凯撒是同一个类型，所以这里看看移动了多少位，发现都是移动了13位。那么这整个PDF文件的内容应该都是移动了13位才会有这种有规则数据和文件乱码数据的现象吧。

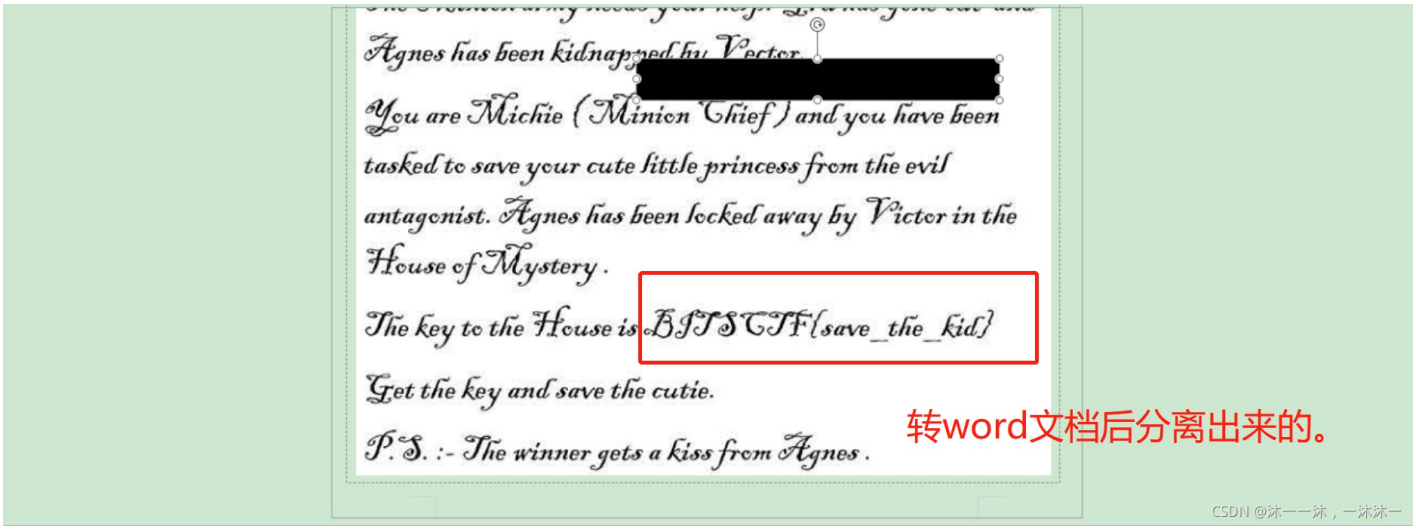
明文字母表	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
密文字母表	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

附上kali的字符移动13位的命令shell，这里tr命令的应用也是学到了知识，A-Za-z按顺序对应N-ZA-Mn-za-m，这连着的正则表达式之间不用空格也不用分号，无拘无束：

cat 1.pdf | tr A-Za-z N-ZA-Mn-za-m > 2.pdf

然后，转出来的PDF还自带黑格隐藏，一开始我都不记得怎么分离了，后来查资料说转html可以分离，因为这是两张图片重叠在一起而已。事实上用我的嗨格式永久VIP会员PDF转HTML、WORD、PPT都是可以分离的。



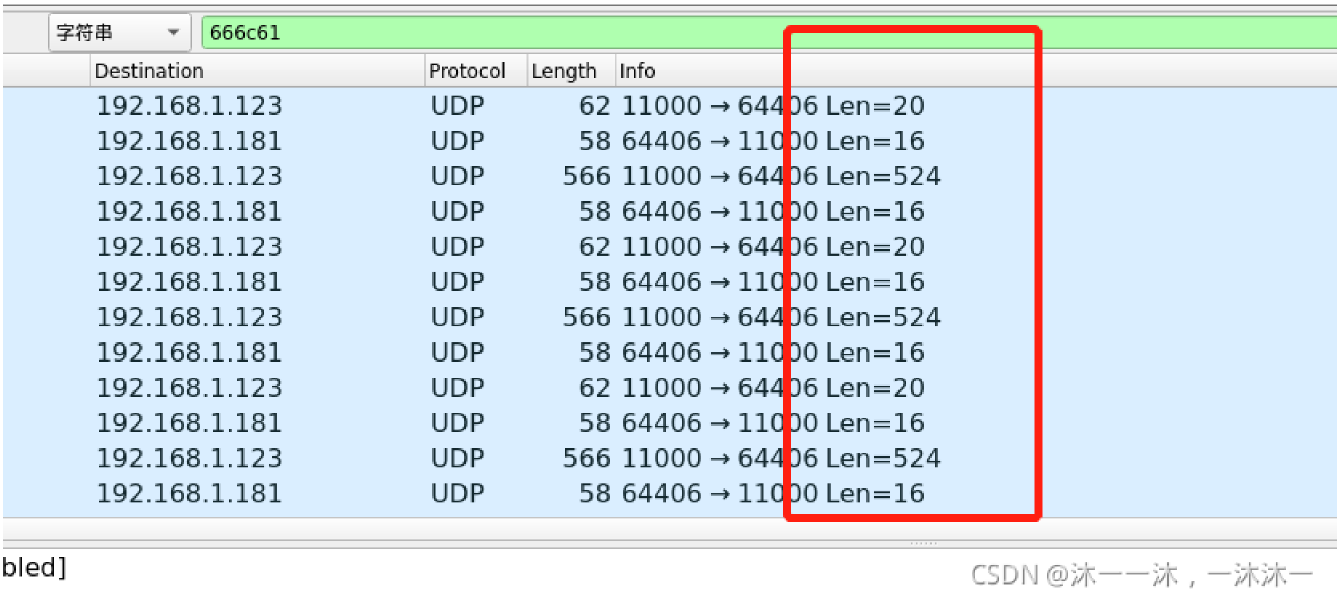


流量相关类型：

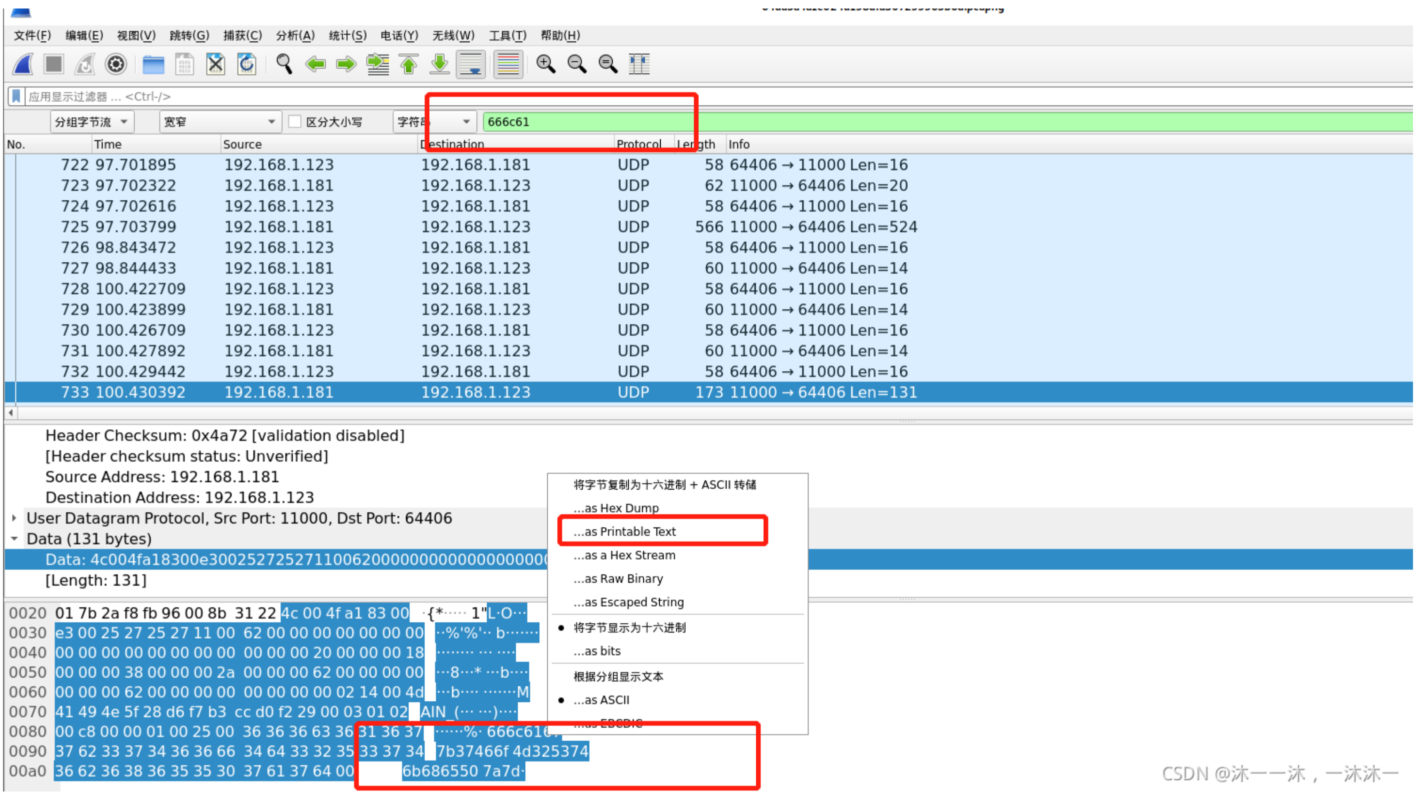
攻防世界之工业协议分析2：（16进制转字符、）

下载附件，是一个pcapng流量文件：

题目描述说已提取出上位机通信流量，尝试分析出异常点，获取FLAG。 flag形式为 flag{}，这其实就是一个暗示，flag就在通信流量中，至于哪里异常，查了资料，有的说UDP的长度有部分异常，要一个个点击查看。可是为什么我感觉UDP大的一堆，小的也一堆，也不知道哪里异常：



然后就是题目暗示说flag在异常流量中，那直接明文匹配查找不就行了？先搜索flag，无果，换十六进制666c6167试试，查到了，用as printable text 选项dump下来即可：



结果：

16进制到文本字符串

加密或解密字符串长度不可以超过10M

```
1  666c61677b37466f4d3253746b6865507a7d
```

16进制转字符 | 字符转16进制 | 测试用例 | 清空结果 | 复制结果


```
1  flag{7FoM2StkhePz}
```


加密逻辑平铺类型

传统密码加密逻辑平铺：

2021年9月绿城杯，CRYPTO的[warmup]加密算法：（暴力破解、仿射密码、下标对应解密）

下载附件，是一个文本文件，逻辑代码平铺在里面：

打开文件，发现是一个简单的加密表单元素下标对应加密：

```
from Crypto.Util.number import *
from flag import flag
assert flag[:5]=='flag{'

str1 = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
def encode(plain_text, a, b, m):
        cipher_text = ''
        for i in plain_text:
                if i in str1:
                        addr = str1.find(i)
                        cipher_text += str1[(a*addr+b) % m]
                else:
                        cipher_text += i
        print(cipher_text)

encode(flag,37,23,52)
# cipher_text = 'aoxL{XaaHKP_tHgwpc_hN_ToXnnht}'
```

加密表单

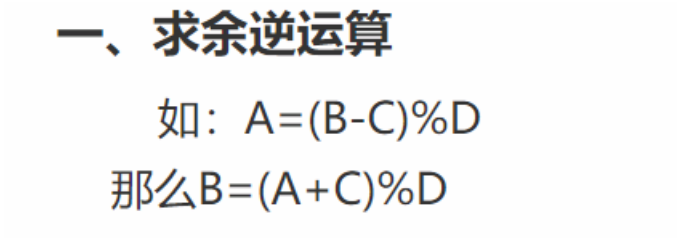
明文对应表单进行下标对应，下标是加密关键。

不在表单内的直接获取即可

下标加密的一些参数

密文

因为下标加密是求余运算加密，所以上网找了一下求余运算的逆运算，结果发现负数的求余运算我解决不了。。。(哭~)

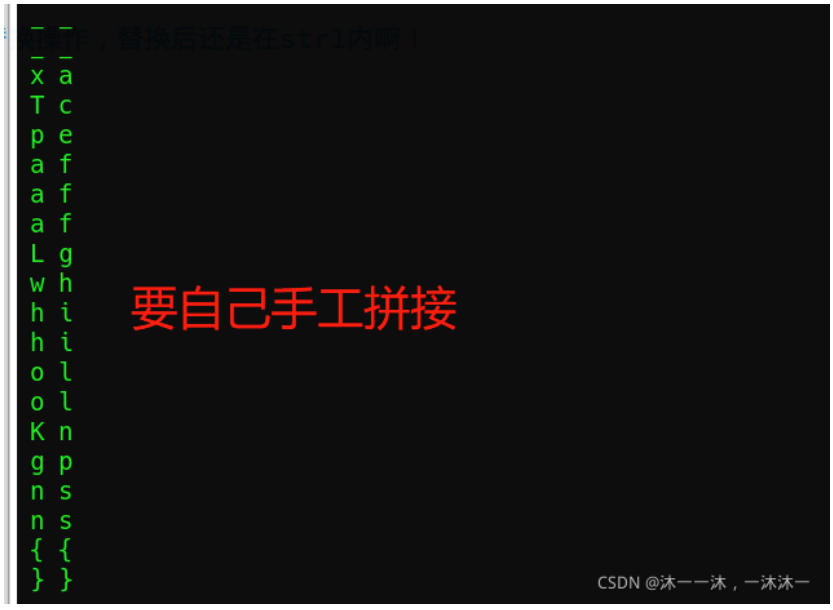


所以直接爆破算了，这是我第一次真正用爆破做题，所以比较生疏，先回顾一下一开始我犯的错误。

一开始我写的脚本是这样的，在ASCII的32~127中找到加密后与密文对应的字符，然后取chr(i)，然后错误就比较明显了，因为for循环会一直向前走，如果密文不按顺序摆放就会跳过，所以没法用flag+=chr(i)的方法来获取密文，结果就是得到明文字符----密文字符的对应，然后要自己一个个拼接：

```
mi_wen= 'aoxL{XaaHKP_tHgwpc_hN_ToXnnht}'

str1 = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ' #长度52

c=0

flag=[]

cipher_text = ''

for i in range(32,127,1):

if chr(i) in str1: #flag在a~Z内，就获取下标并进行下标替换操作，替换后还是在str1内啊！

addr = str1.find(chr(i))

cipher_text = str1[(37*addr+23) % 52] #求余52

else: #flag不在a~Z内，加1后还是不在str1内

cipher_text = chr(i) #flag在a~Z内，就不用加密直接用
```

```
for a in mi_wen:

if cipher_text==a:

print(a,chr(i))
```

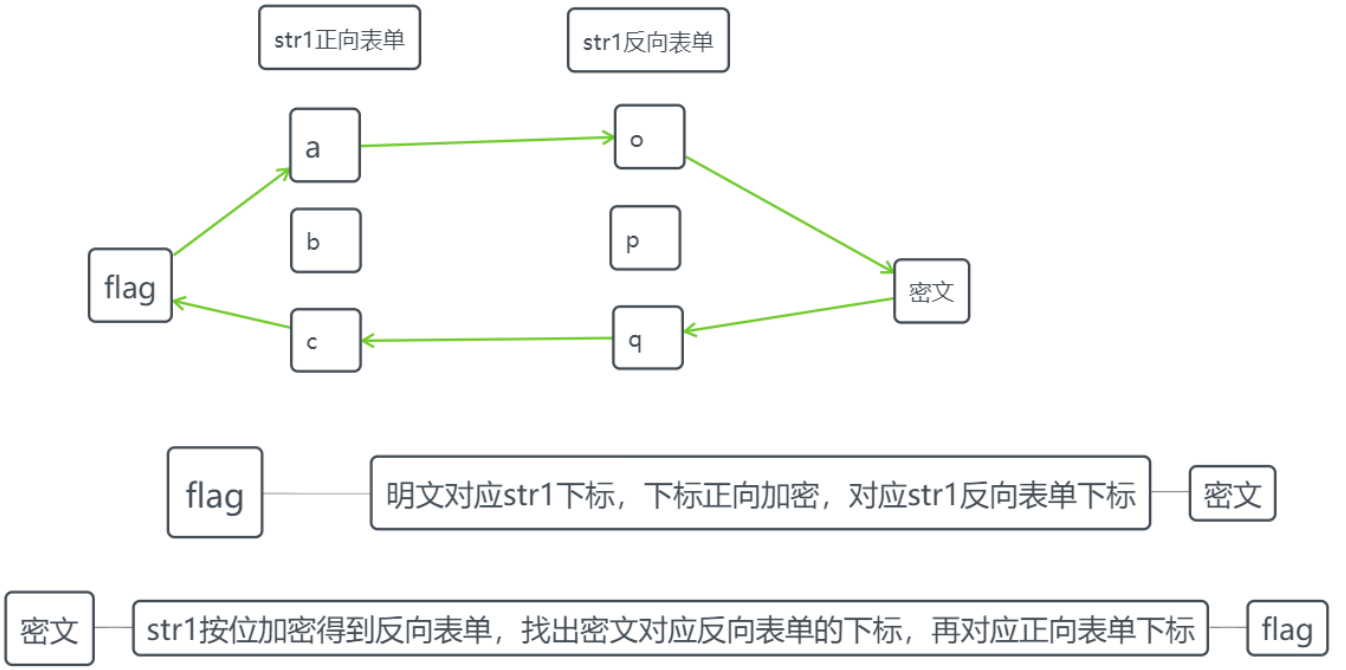


后来想了想，竟然for循环一直往前走，但是如果我换一下顺序，每个密文字符遍历一次ASCII字符，那不就密文也往前走了吗？虽然这样的算法复杂度大大增加，但是对电脑来说根本不值得一提好吧：

```
mi_wen= 'aoxL{XaaHKP_tHgwpc_hN_ToXnnht}'

str1 = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ' #长度52

c=0

flag=[]

cipher_text = ''

for a in mi_wen:

for i in range(32,127):

if chr(i) in str1: #flag在a~Z内，就获取下标并进行下标替换操作，替换后还是在str1内啊！

addr = str1.find(chr(i))

cipher_text = str1[(37*addr+23) % 52] #求余52

else: #flag不在a~Z内，加1后还是不在str1内

cipher_text = chr(i) #flag在a~Z内，就不用加密直接用

if cipher_text==a:

flag+=chr(i)

print(''.join(flag))
```

```
└─$ python 3.py
flag{AffInE_CIpheR_iS_clAssiC}
```

这里补充一些别人的做法，他直接把码表加密，之后按位找。是一种比较新颖的方法，是对正向反向下标搞操作，值得研究，我感觉他的逻辑应该是这样的：



还来有一天发现了是仿射密码：

仿射密码

仿射密码也是一个单表替换密码，字母表中的每个字母相应的值使用一个简单的数学函数映射到对应的数值，再把对应数值转换成字母。

仿射密码的加密函数是 $e(x) = ax + b \pmod{m}$，其中：

- a和m互质。
- m是字母的数目。

解密函数是 $d(x) = a^{-1}(x - b) \pmod{m}$，其中 a^-1 是 a 在集合 Z 上的逆元，满足 a * a^-1 ≡ 1 mod 26

求集合Z上某个数的逆元：

```
gmpy2.invert(x, Z)
```


下标逆向脚本：

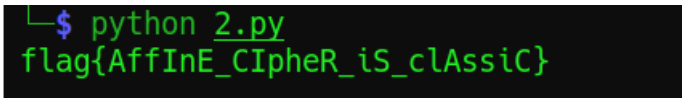
mi_wen= 'aoxL{XaaHKP_tHgwpc_hN_ToXnnht}'

str1 = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ' #长度52

flag=''

```
def encode(plain_text, a, b, m):

cipher_text = ''

for i in plain_text:

if i in str1:

addr = str1.find(i)

cipher_text += str1[(a*addr+b) % m]

else:

cipher_text += i

return cipher_text

reverse=encode(str1,37,23,52) #正向表单全部加密得出反向表单

for i in mi_wen: #对应密文下标

if i in str1:

addr=reverse.find(i) #对应反向str1下标，密文在反向表单中的下标等同与flag在正向表单的下标，就是下标等价，类似于base64表单替换。

flag+=str1[addr] #获取正向明文，因为小标是一样对应的

else:

flag+=i

print(flag)
```

结果：



最后还差原本的负数求余逆运算了，这个暂时找不到资料，先放着。~

复杂加密类型

RSA直接给参数类型：

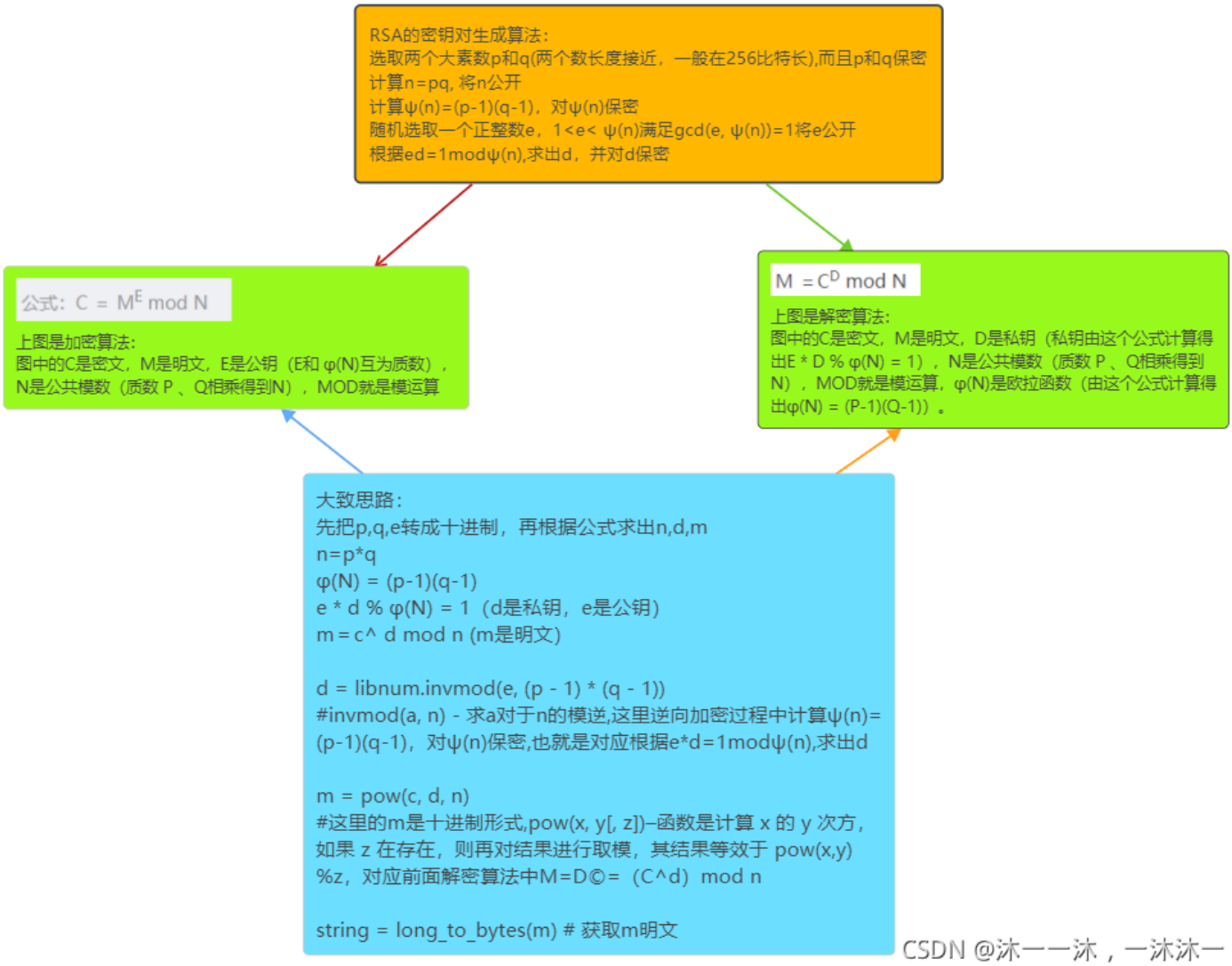
攻防世界之cr3-what-is-this-encryption：（RSA通用脚本解密）

没有附件，依稀看得出有q、p、e、c，是简单的RSA题目，用我之前积累的RSA脚本CTF-RSA-tool跑一下，。。。跑不出来：(反复试了好多次，还是报错，我放弃了)

然后在网上查资料中找到一个讲得比较好的脚本和讲解：

https://www.cnblogs.com/zhengna/p/13501563.html

RSA的密钥对生成算法：



讲得很透彻了，解密的算法也提供了，附上脚本和我的一点点见解：（这是RSA通用脚本）

```
import libnum

from Crypto.Util.number import long_to_bytes

q = int(

"0xa6055ec186de51800ddd6fcbf0192384ff42d707a55f57af4fcfb0d1dc7bd97055e8275cd4b78ec63c5d592f567

16)

p = int(

"0xfa0f9463ea0a93b929c099320d31c277e0b0dbc65b189ed76124f5a1218f5d91fd0102a4c8de11f28be5e4d0a

16)

e = int(

"0x6d1fdab4ce3217b3fc32c9ed480a31d067fd57d93a9ab52b472dc393ab7852fbcb11abbebfd6aaae8032db131

16)
```

c =
0x7fe1a4f743675d1987d25d38111fae0f78bbea6852cba5beda47db76d119a3efe24cb04b9449f53becd43b0b46

n = q * p

d = libnum.invmod(e, (p - 1) * (q - 1)) #invmod(a, n) - 求a对于n的模逆,这里逆向加密过程中计算φ(n)=(p-1)(q-1)，对φ(n)保密,也就是对应根据ed=1modφ(n),求出d

m = pow(c, d, n) # pow(x, y[, z])--函数是计算 x 的 y 次方，如果 z 在存在，则再对结果进行取模，其结果等效于 pow(x,y) %z，对应前面解密算法中M=D(C)=C^d(mod n)

#print(m) #明文的十进制格式

string = long_to_bytes(m) # m明文，用长字节划范围

print(string.decode())

攻防世界之OldDriver：（低加密指数广播攻击）

下载附件，一个.txt文件，打开，发现是c、n、e的RSA类型题：

[{"c": 736606757474117146172206513324291608049550591366325033008274746538367689397041147655074839484 1·
{"c": 219628253233004691517959202898868865627909427715468585008421798065664357671038039788851487721 39
{"c": 656968942027406695783598339058358528657008761904811014118770058419379269523540507781154435516 92
{"c": 450824616804451351845249388271353639063674154155180582179033897379761597127186724858437981311 41
{"c": 229661056702912823355888430182441615527644863731179428659669040761911223374355425532767439388 17
{"c": 179633130634050457429681369162198383521355617853895343812629792645853978968444708790236865085 40
{"c": 165241753470902945038057065397370532098611767959756387302268314080050748256048294831013154094 82
{"c": 155857717344883510394566313940404977595686794295106192197661917808076753617418592904907324511 12
{"c": 896512342163769405004421684452337916334747802912481503283281322505073255852423966064874628488 41
{"c": 135609457565430230085293881084469408471378530384370952445730358885312885773708290656663200693 97

上网查了查，发现类型是低加密指数广播攻击，这里附上别人的话：(里面的中国剩余定理我还没看懂)

首先介绍什么是广播，加入我们需要将一份明文进行多份加密，但是每份使用不同的密钥，密钥中的模数n不同但指数e相同且很小，我们只要拿到多份密文和对应的n就可以利用中国剩余定理进行解密。关于中国剩余定理请参考文章：

https://www.cnblogs.com/freinds/p/6388992.html

只要满足以下情况，我们便可以考虑使用低加密指数广播攻击：

加密指数e非常小

一份明文使用不同的模数n，相同的加密指数e进行多次加密

可以拿到每一份加密后的密文和对应的模数n、加密指数e

RSA秘钥的指数e=10，这些特征暗示低加密指数广播攻击；当加密一份消息给多人时，
$c_i = m^e \bmod N_i, i = 1, \cdots, k$，当$k \geq e$时，可由中国剩余定理求解$m^e$，再开方即得明文$m$。

本来想参考着写一个脚本的，发现CTF-RSA-tool工具里面好像可以解决低加密指数广播攻击。附上我以前的笔记：

Basic Broadcast Attack(低加密指数广播攻击)

python2 solve.py --verbose -i examples/Basic_Broadcast_Attack.txt

```
c : 68679409966918700315304117144859068445528181933255289063193054014288151083466807594332167633810967321824633144 4
e : 7
n : 24810910852704603048663349011054669655631146433543459534796438815331335687309113943583212235150241971378068933 1

c : 11179052201843296851154916696601291938235676417475847173247453892471333698920515860207084422135916963942522752 2
e : 7
n : 47127839105299361033791208737798899776781255381503030381686909082155757361019104103280620540716894699133142173 1

c : 40118076943735337559537379692982070943921515348000211097599356263330760075906748374129727526740438883695094503 1
e : 7
n : 43134291711046821358455351358884087777021003839470296505990450581706219379356272391794220129036895199873385802 5

c : 12649076592222649371192164869044025408231371717627780046219346377852024544337050152652676577122342534868958091 7
e : 7
n : 19300838921149221007298944887478599082800229045219271606272038103970656559943914197281654158587468730541828306 4

c : 28899089935435267588235897519846120393433214114341521238696384122507316899457327055029546972333281452563984838 4
e : 7
n : 30754121488827635692971849599267749375077949182550303145729325375314926401905783830931628738658879320179944880 0

c : 14086629413855672403639830676118042465846020320143823318815048070368505684208141652603596234960968703217788472 9
e : 7
n : 30430477983470426195631142659668071772256641205525929891985872996115858010744648779370983539942187689192406517 4

c : 20049299207588955907155276095787913402589652379134151403340360498371893119855957833576443856534037886298731701 9
e : 7
n : 35489275126536805974281635942907480463916089663069129771420548612817920002692423639961709000309976531819984030 3
```


那么把题目的enc.txt照着格式摆放，然后运行脚本即可：

c :
736606757474117146172206513324291608049550591366325033008274746538367689397041147655074 8

e : 10

n :
251625070523397144218396888737345961777511240367238310033009597611378114907152057429417 3

c :
219628253233004691517959202898868865627909427715468585008421798065664357671038039788851 4

e : 10

n :
239768595899044197983208120976818586523254737918912327104319972028978195806349370709006 2

c :
656968942027406695783598339058358528657008761904811014118770058419379269523540507781154 4

e : 10

n :
185037828368585400439745580356016546109489155056452198201502510623051201487455459065675 4

c :
450824616804451351845249388271353639063674154155180582179033897379761597127186724858437 9

e : 10

n :
233830874785455122187131579329347461107217068190774234180602200836577134285035828019098O

c :
229661056702912823355888430182441615527644863731179428659669040761911223374355425532767 4

e : 10

n :
317756490898614286710579090761441528707967225281125804794420733650539160125072734330284 5

c :
179633130634050457429681369162198383521355617853895343812629792645853978968444708790236 8

e : 10

n :
222463420229434328206961904441556652899283786538411726322832278881744954022486330610106 1

c :
165241753470902945038057065397370532098611767959756387302268314080050748256048294831013 1

e : 10

n :
253954611426706312681561061360283257443933584366175286779672493473535249246550011518495 4

c :
155857717344883510394566313940404977595686794295106192197661917808076753617418592904907 3

e : 10

n :
320565088927441849012894132877280398913038323115486081410882278763267536741541247751327 7

c :
896512342163769405004421684452337916334747802912481503283281322505073255852423966064874 6

e : 10

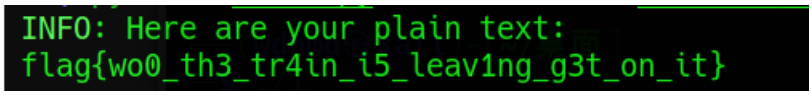
n :
528497662695418274742281894288206485741625395959853959922616498099074357422630205510500 6

c :
135609457565430230085293881084469408471378530384370952445730358885312885773708290656663 2

e : 10

n :
30415984800307578932946399987559088968355638354344823359397204419191241802721772499486610

结果：



积累一下别人的脚本以备后用：(python3)

import libnum

import gmpy2

dic =
[{"c":736606757474117146172206513324291608049550591366325033008274746538367689397041147655(
"e": 10,
"n":251625070523397144218396888737345961777511240367238310033009597611378114907152057429

{"c":219628253233004691517959202898868865627909427715468585008421798065664357671038039788
"e": 10,
"n":239768595899044197983208120976818586523254737918912327104319972028978195806349370709(

{"c":656968942027406695783598339058358528657008761904811014118770058419379269523540507781
"e": 10,
"n":185037828368585400439745580356016546109489155056452198201502510623051201487455459065

{"c":450824616804451351845249388271353639063674154155180582179033897379761597127186724858
"e": 10,
"n":233830874785455122187131579329347461107217068190774234180602200836577134285035828019

{"c":229661056702912823355888430182441615527644863731179428659669040761911223374355425532
"e": 10,
"n":317756490898614286710579090761441528707967225281125804794420733650539160125072734330

{"c":179633130634050457429681369162198383521355617853895343812629792645853978968444708790
"e": 10,
"n":222463420229434328206961904441556652899283786538411726322832278881744954022486330610

{"c":165241753470902945038057065397370532098611767959756387302268314080050748256048294831
"e": 10,
"n":253954611426706312681561061360283257443933584366175286779672493473535249246550011518

{"c":155857717344883510394566313940404977595686794295106192197661917808076753617418592904
"e": 10,
"n":320565088927441849012894132877280398913038323115486081410882278763267536741541247751

{"c":8965123421637694050044216844523379163347478029124815032832813225050732558524239660648
"e": 10,
"n":52849766269541827474228189428820648574162539595985395992261649809907435742263020551050

{"c":1356094575654302300852938810844694084713785303843709524457303588853128857737082906566
"e": 10,
"n":30415984800307578932946399987559088968355638354344823359397204419191241802721772499480

```
n = []

C = []

for i in dic:

n.append(i["n"])

C.append(i["c"])

N = 1

for i in n:

N *= i

Ni = []

for i in n:

Ni.append(N // i)

T = []

for i in range(10):

T.append(int(gmpy2.invert(Ni[i], n[i])))

X = 0

for i in range(10):

X += C[i] * Ni[i] * T[i]

m10 = X % N

m = gmpy2.iroot(m10, 10)

print (libnum.n2s(m[0]))
```

python2：

```
import libnum

import gmpy2
```

dic =
[{"c":736606757474117146172206513324291608049550591366325033008274746538367689397041147655(
"e": 10,
"n":251625070523397144218396888737345961777511240367238310033009597611378114907152057429

{"c":219628253233004691517959202898868865627909427715468585008421798065664357671038039788
"e": 10,
"n":239768595899044197983208120976818586523254737918912327104319972028978195806349370709

{"c":656968942027406695783598339058358528657008761904811014118770058419379269523540507781
"e": 10,
"n":185037828368585400439745580356016546109489155056452198201502510623051201487455459065

{"c":450824616804451351845249388271353639063674154155180582179033897379761597127186724858
"e": 10,
"n":233830874785455122187131579329347461107217068190774234180602200836577134285035828019

{"c":229661056702912823355888430182441615527644863731179428659669040761911223374355425532
"e": 10,
"n":317756490898614286710579090761441528707967225281125804794420733650539160125072734330

{"c":179633130634050457429681369162198383521355617853895343812629792645853978968444708790
"e": 10,
"n":222463420229434328206961904441556652899283786538411726322832278881744954022486330610

{"c":165241753470902945038057065397370532098611767959756387302268314080050748256048294831
"e": 10,
"n":253954611426706312681561061360283257443933584366175286779672493473535249246550011518

{"c":155857717344883510394566313940404977595686794295106192197661917808076753617418592904
"e": 10,
"n":320565088927441849012894132877280398913038323115486081410882278763267536741541247751

{"c":896512342163769405004421684452337916334747802912481503283281322505073255852423966064
"e": 10,
"n":528497662695418274742281894288206485741625395959853959922616498099074357422630205510

{"c":135609457565430230085293881084469408471378530384370952445730358885312885773708290656
"e": 10,
"n":304159848003075789329463999875590889683556383543448233593972044191912418027217724994

n = []

C = []

for i in dic:

n.append(i["n"])

```
C.append(i["c"])

N = 1

for i in n:

N *= i

Ni = []

for i in n:

Ni.append(N / i)

T = []

for i in xrange(10):

T.append(long(gmpy2.invert(Ni[i], n[i])))

X = 0

for i in xrange(10):

X += C[i] * Ni[i] * T[i]

m10 = X % N

m = gmpy2.iroot(m10, 10)

print libnum.n2s(m[0])
```

脚本解析：

- python2.x和python3.x环境脚本存在三个不同
- `print` 有无括号 （不提也罢）
- `Ni.append(N / i)` 在Python3中， `N / i` 返回的是 `float` 类型， `float` 最大值为 `1.7976931348623157e+308` 因此会报错 `OverflowError: integer division result too large for a float` 所以要将它绕过，用//代替/，这样返回的是 `Integer` 。
- `T.append(long(gmpy2.invert(Ni[i], n[i])))` 在Python3中，不存在 `long()` 函数，直接改为 `int()` 就可

RSA明文密钥文件类型：

攻防世界之best_rsa：（明文密钥文件提取参数、RSA共模攻击、CTF-RSA-tool脚本修改）

下载题目，是一个明文和密钥的4个附件：

其实一开始我并不知道RSA的明文和密钥是怎么生成的，CTF-RSA-tool中应对的也只有一对明文密钥文件而已，这里两对的话就只能查资料弄懂原理了再做了。

从别人的博客 攻防世界 best_rsa - vict0r - 博客园 中找到了从密钥文件和明文文件读取出对应数字的方法，提取n、e都是用Crypto.PublicKey.RSA模块，再抽取对应的n,c属性的。

提取加密密文c，则是直接二进制读取文件后用Crypto.Util.number模块的bytes_to_long函数转二进制流为数字的。

```
from Crypto.PublicKey import RSA

from Crypto.Util.number import *

f1 = open("publickey1.pem","rb").read()

f2 = open("publickey2.pem","rb").read()

c1 = open("cipher1.txt","rb").read()

c2 = open("cipher2.txt","rb").read()

pub1 = RSA.importKey(f1)

pub2 = RSA.importKey(f2)

n1 = pub1.n

e1 = pub1.e

n2 = pub2.n

e2 = pub2.e
```

c1 = bytes_to_long(c1)

c2 = bytes_to_long(c2)

print("n1 =",n1)

print("e1 =",e1)

print("c1 =",c1)

print("n2 =",n2)

print("e2 =",e2)

print("c2 =",c2)

结果：

n1 =
13060424286033164731705267935214411273739909173486948413518022752305313862238166593214777

e1 = 117

c1 =
12847007370626420814721007824489512747227554004777043129889885590168327306344216253180827

n2 =
13060424286033164731705267935214411273739909173486948413518022752305313862238166593214777

e2 = 65537

c2 =
68308576617031565989734336170550458032770042742873009976346488004482336557564980706935977

这里两个相同的n，就是共模攻击了，在CTF-RSA-tool工具里有对公模攻击的解法：

```
# 共模攻击
python2 solve.py --verbose -i examples/share_N.txt
n = 49717352238903813258167965634872644122363546007764600124211711250852904300466263993456901629465018966087
e = 13720370251305502198453722303188629715020031854527043028194234587721186750392338568840741980913602828877
c = 33995928221963087827338803441589901571528069870645123081701004401546919289265619397264329787933998397647

n = 49717352238903813258167965634872644122363546007764600124211711250852904300466263993456901629465018966087
e = 13521927979417175825463063347222803267672338126236919097685639546419640649137631879086526055040440600767
c = 20663727752331650736213993392793715555002283733124450484396596147661401938360252713595785342791730649227
```


所以摆好格式后直接跑脚本：

```
INFO: Here are your plain text:
flag{interesting_rsa}
```

当然完全依靠工具是不行的，所以抽取出CTF-RSA-tool对应代码做成脚本：

(注意：这里因为用python2运行，所以不能用直接用open().read()来读取文档内容，会报错。在外面要裹上libnum.s2n()才行)

```
from Crypto.PublicKey import RSA
from Crypto.Util.number import *
import gmpy2
import libnum
def share_N(N, e1, e2, c1, c2):
gcd, s, t = gmpy2.gcdext(e1, e2)
if s < 0:
s = -s
c1 = gmpy2.invert(c1, N)
if t < 0:
t = -t
c2 = gmpy2.invert(c2, N)
plain = gmpy2.powmod(c1, s, N) * gmpy2.powmod(c2, t, N) % N
print(libnum.n2s(plain))
#上面的函数是从CTF-RSA-tool工具中抽取出来的，下面的n、e、c属性是前面自己写的
c1=libnum.s2n(open('cipher1.txt','rb').read())
c2=libnum.s2n(open('cipher2.txt','rb').read())
pub1=RSA.importKey(open('publickey1.pem').read())
pub2=RSA.importKey(open('publickey2.pem').read())
n = pub1.n
e1= pub1.e
e2= pub2.e
share_N(n,e1,e2,c1,c2)
```

攻防世界之RSA256：（）

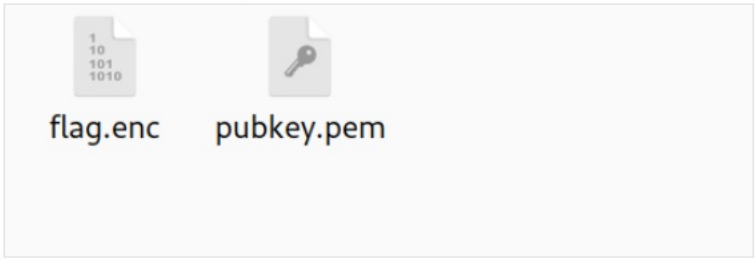
下载压缩包，解压，是两个附件，一个没有后缀，一个.txt文件，打开查看内容，感觉是RSA的密文密钥文件：

RSA256/ 0000755 0000000 0000000 00000000000 13446064553
峬??绘□癥W Z闊□℃?噭傌野□(?
BLIC KEY-----
MDwwDQYJKoZIhvcNAQEBBQADKwAwKAIhAKm9THp3YzcKBC/mvsfdyEFgLblCx6Ni
0bXTcqTQiRLZAgMBAAE=
-----END PUBLIC KEY-----

RSA密钥文件

翻一下以前的笔记，这是对CTF-RSA-tool使用的示例，感觉题目类型就是这种密钥和密文的文件：

```
# factordb.com
python2 solve.py --verbose  -k examples/jarvis_oj_mediumRSA/pubkey.pem --decrypt
examples/jarvis_oj_mediumRSA/flag.enc
```




```
-----BEGIN PUBLIC KEY-----
MDowDQYJKoZIhvcNAQEBBQADKQAwJgIhAMJjauXD2OQ/+5erCQKPGqxsC/bNPXDr
yigb/+l/vjDdAgEC
-----END PUBLIC KEY-----
```

默认 (UTF-8, 部分) 其他： ISO-8859-1

n>·ß#îáÓ¾ºx ëmÚdAy

验证猜想，直接脚本跑一下，得到flag：

```
└─$ python2 solve.py --verbose  -k /home/wdnmd/桌面/gy.key --decrypt /home/wdnmd/桌面/1.txt

DEBUG: factor N: try past ctf primes
DEBUG: factor N: try Gimmicky Primes method
DEBUG: factor N: try Wiener's attack
DEBUG: Starting new HTTP connection (1): www.factordb.com:80
DEBUG: http://www.factordb.com:80 "GET /index.php?query=767753333402239611394270507078404178
111569780851469703123158866715466662591 61 HTTP/1.1" 200 993
DEBUG: http://www.factordb.com:80 "GET /index.php?id=1100000001249442187 HTTP/1.1" 200 873
DEBUG: http://www.factordb.com:80 "GET /index.php?id=1100000001249442188 HTTP/1.1" 200 875
DEBUG: d = 0x6e1e3e007175af5e532b14628edda62d1d0f7561a926a10f29853bde0a5ed611L
INFO: c@L@u@5flag{_2o!9_CTF_ECUN_}
```

RSA脚本逻辑加密类型：

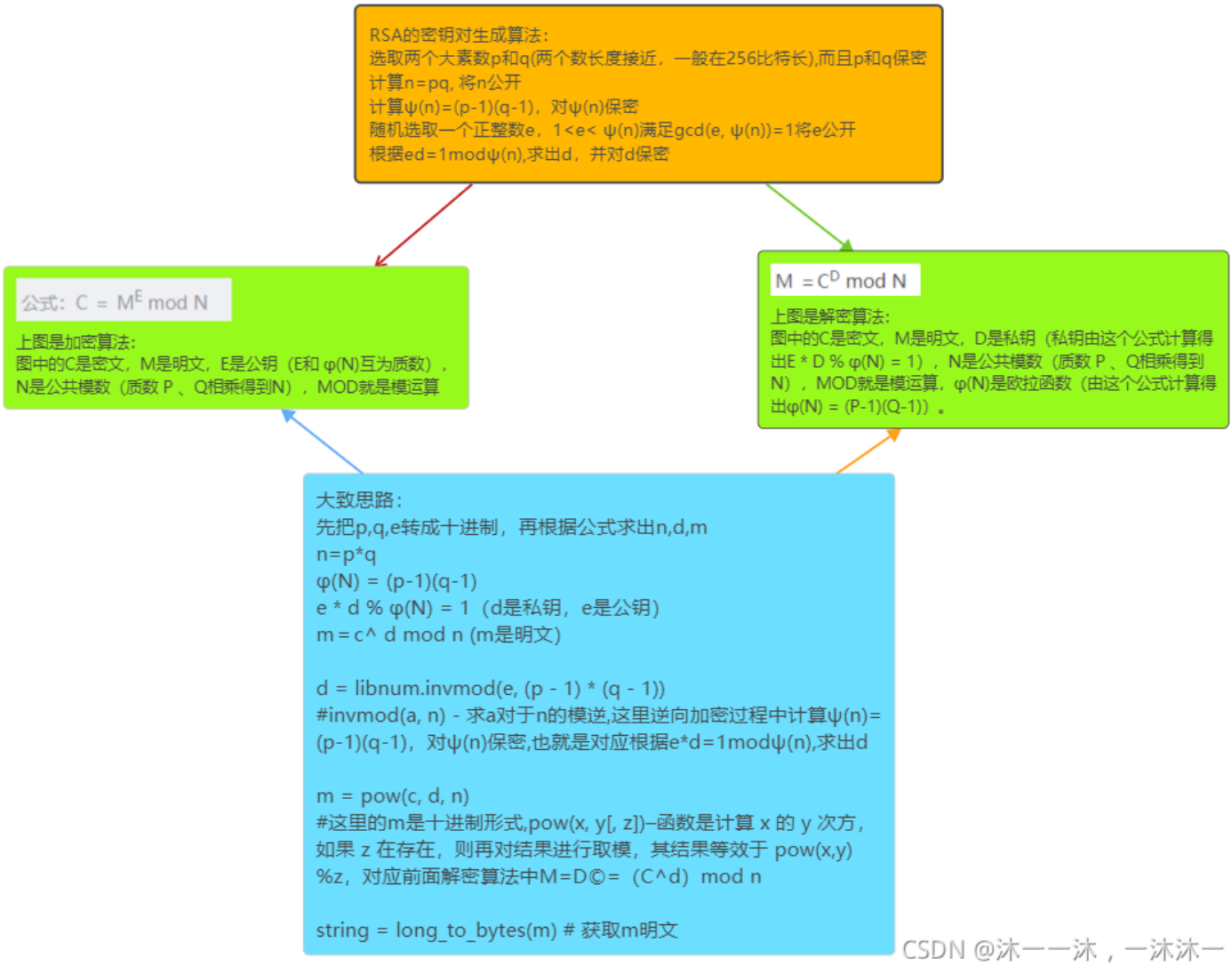
2021年9月广州羊城杯，CRYPTO的BigRsa：（RSA多层模n加密、CTF-RSA-tool脚本修改、RSA模不互素）

下载附件，pub.py：

打开文件，发现内容是RSA加密过程：

```
from Crypto.Util.number import *

from flag import *

n1 =
10383529640908175186077053551474658681539589842726033432568031364836913266105784068082320

n2 =
11538319858467714748755601433644831072185384116875801244563418281418031448050182892716000

e = 65537

m = bytes_to_long(flag)

c = pow(m, e, n1)

c = pow(c, e, n2)

print("c = %d" % c)

# output

# c =
60406168302768860804211220055708551816238816061772464557956985699400782163597251861675960
```

首先回顾一下以前的积累的RSA脚本解题的知识：



然后回到羊城杯的题目你就会发现，题目先用公共模数n1对flag m加密一次，再用公共模数n2对RSA一次加密后的密文再加密一次，这是两层加密：

```
e = 65537
m = bytes_to_long(flag)
c = pow(m, e, n1)
c = pow(c, e, n2)

print("c = %d" % c)
```

两层加密

按理来说两层常规的RSA解密即可，关键就是这里的n1、n2太大了，无法硬分解，常规的分解素数网站没法用。

1153831985846771474875560143364483107218538411687580124456341828141803144805018289271600710151 Factorize!

Result:		
	digits	number
	309 (show)	1153831985...73<309> = 1153831985...73<309>

分解不出

More information

然后就去查资料了，因为RSA自己也不在行，下面是从 CTF密码学中RSA学习以及总结_韦全敏的博客-CSDN博客

中找到的。

RSA小指数e攻击

如果RSA系统的公钥e选取较小的值，可以使得加密和验证签名的速度有所提高，但是如果e的选取太小，就容易受到攻击。

有三个分别使用不同的模数n1,n2,n3，但是都选取e=3，加密同一个明文可以得到：

c1 = pow(m,3,n1)

c2 = pow(m,3,n2)

c3 = pow(m,3,n3)

一般情况下，n1,n2,n3互素，否则会比较容易求出公因子，从而安全性大幅度的减低。

利用公约数：

如果两次加密的n1和n2具有相同的素因子，可以利用欧几里德算法直接分解n1和n2.

通过欧几里德算法计算出两个n的最大公约数p：

```
def gcd(a, b):

if a < b:

a, b = b, a

while b != 0:

temp = a % b

a = b

b = temp

def gcd_digui(a, b):

if b != 0:

return a

return gcd(b,a%b)

p = gcd(n1,n2)
```

RSA小指数e攻击题目特征识别：

识别此类题目，通常会发现题目给了若干个n，均不相同，并且都是2048bit，4096bit级别，无法正面硬杠，并且明文都没什么联系，e也一般取65537。

根据欧几里德算法算出的p之后，再用n除以p即可求出q，由此可以得到的参数有p、q、n、e，再使用常规方法计算出d，即可破解密文。

和题目八九不离十，那这道bigrsa就是这样解的了，用两个公共模数n来求公因子：

在此之前中间还有个小插曲，那就是CTF-RSA-tool工具，一开始我以为这中两个n、一个e、一个c的是CTF-RSA-tool工具中example的模不互素：(下面是我以前的笔记)

模不互素

python2 solve.py --verbose -i examples/share_factor.txt

```
n = 208233691145562607629135888444718697257629858122159879938677836300514202410579123850554827880163279784683180670782338
n = 190838216137364299584320249800744053754089532692768396963192655968554261892568656506514604600798193689235761097230795
e = 65537
c = 132340339973043167780377237555402951765664171675831253347481153138562724618734859751767696399005566727346172733590195
```

后来怎么跑都跑不出来，看了工具内对应代码才发现模不互素是给你两个单独分解不了的大数级n，方便让你求公因子。然后，只用一个n来加密，对，只用一个n。(我也是现在才理解模不互素的内容)

```
77
78 # 模不互素: 需要(n1,e1,c1)及(n2,e2),且e1 == e2。解密c1
79 def share_factor(n1, n2, e, c1):
80     p1 = gmpy2.gcd(n1, n2)
81     q1 = n1 / p1
82     d = gmpy2.invert(e, (p1 - 1) * (q1 - 1))
83     plain = gmpy2.powmod(c1, d, n1)
84     plain = hex(plain)[2:]
85     if len(plain) % 2 != 0:
86         plain = '0' + plain
87     log.info('Here are your plain text: \n' + plain.decode('hex'))
88
89
```

顺序传入n1,n2,e,c1
然后求公共因子，最后
只解出n1加密的明文。

前面博客的欧几里德算法脚本不太会用，所以直接抽出CTF-RSA-tool工具的这段脚本来用算了，因为上面的求公因子代码已经写好了，只要再加多一层解密即可，脚本如下：

```
import gmpy2

def share_factor(n1, n2, e, c1):

p1 = gmpy2.gcd(n1, n2) #gcd是求共因子用的，相当于欧几里德算法的封装实现

q1 = n1 / p1

q2 = n2 / p1

d1 = gmpy2.invert(e, (p1 - 1) * (q1 - 1))

plain = gmpy2.powmod(c1, d1, n1)

d2 = gmpy2.invert(e, (p1 - 1) * (q2 - 1))

plain = gmpy2.powmod(plain, d2, n2)

plain = hex(plain)[2:]

if len(plain) % 2 != 0:

plain = '0' + plain

print('Here are your plain text: \n' + plain.decode('hex'))

if __name__ == "__main__":

n1 =
11538319858467714748755601433644831072185384116875801244563418281418031448050182892716000
```

n2 =
10383529640908175186077053551474658681539589842726033432568031364836913266105784068082329551223694895337089556841972133117083455781254146830929881949726774689281

e = 65537

c1 =
60406168302768860804211220055708551816238816061772464557956985699400782163597251861675961

share_factor(n1, n2, e, c1)

最后附上别人WP的脚本(只能截图了)：

不同在于我的脚本直接用p1 = gmpy2.gcd(n1, n2)取公因子，而他使用自定义gcd函数求公约数，就是利用了前面的欧几里得算法：

```
from gmpy2 import invert
from Crypto.Util.number import long_to_bytes

def gcd(a, b):
    if a < b:
        a, b = b, a
    while b != 0:
        temp = a % b
        a = b
        b = temp
    return a


n1 =
10383529640908175186077053551474658681539589842726033432568031364836913266105784068082329551223694895337089556841972133117083455781254146830929881949726774689281
```

```
145838064230271673828254791579513658230856390787388476476344068413313070355938107129145453472016190042536026921273702658330920825430671536068280490615 n2 = 1153831985846771474875560143364483107218538411687580124456341828141803144805018289271600710151970894560424721858508938473704818173258688240762452907357497173847696616988950001764414972423718739813536896077111468528915514911685287998143119924714496400145018587634954722671682240156659066273824905655079272720735 e = 65537 p = gcd(n1, n2)

q1 = n1 // p
q2 = n2 // p

c = 60406168302768860804211220055708551816238816061772464557956985699400782163597251861675967909246187833328847989530950308053492202064477410641014045601986036822451416365957817685047102703301347664879870026582087365822433436251615243854347490600004857861059245403674349457345319269266645006969222744554974358264

d2 = invert(e, (p-1)*(q2-1))
c = pow(c, d2, n2)

d1 = invert(e, (p-1)*(q1-1))
flag = pow(c, d1, n1)
print(long_to_bytes(flag))
```


2021年9月绿城杯，CRYPTO的RSA-1：（CTF-RSA-tool脚本修改、RSA通用脚本解密）

下载附件，是逻辑平铺类型：

打开文件，发现是很常规的RSA加密，就是明文进行了二次处理：

```
from Crypto.Util.number import *
import gmpy2
from flag import flag
assert flag[:5]==b'flag{'

m = bytes_to_long(flag)
p = getPrime(1024)
q = getPrime(1024)
n = p * q
print('n =',n)
e = 0x10001
M = 2021 * m * 1001 * p
c = pow(M,e,n)
print('c =',c)

#n = 173652311549263483644782768725584927759117606030023943537236034618984057402347150018201115486
#c = 69449671088154377354289412867841194031383197134557321559250559286465369625976729418058313121306
```

这里的明文经过了简单的乘法进行二次处理

照例用工具CTF-RSA-tool工具，首先提取出来符合CTF-RSA-tool的格式的信息先：

N is
17365231154926348364478276872558492775911760603002394353723603461898405740234715001820111

e is 65537

c is
69449671088154377354289412867841194031383197134557321559250559286465369625976729418058311

运行脚本看一下回显，明文必然是错误的，因为M经过了二次简单乘法处理，所以我们要在脚本中找到对应的地方修改一下才行：

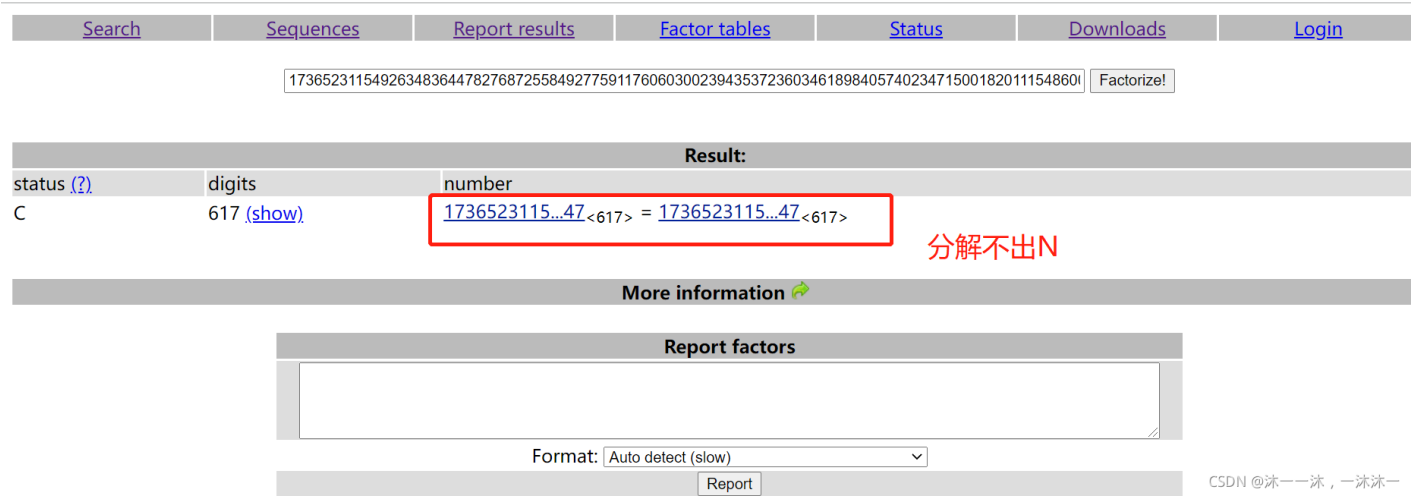
```
└─$ python2 solve.py --verbose  -i /home/wdnmd/桌面/1.txt
DEBUG: factor N: try past ctf primes
DEBUG: factor N: try Gimmicky Primes method
DEBUG: factor N: try Wiener's attack
DEBUG: Starting new HTTP connection (1): www.factordb.com:80
DEBUG: factor N: try Common factor between ciphertext and modulus attack
DEBUG: d = 0x233648b8421f05cf2307820eedcfae249e38abb155880d8f9f44622103ad8e9e1375f6abe86
ef6af69fb29958a3526c0a60923bd30630c66e624275d3dd7125698ca508003c3b3c8316be8c4e13340a9b2c
87f4e3649742a8b7e9f440bee7260c42195b5e601d08b083c559a5acd10cdc17e1cbac588704afc9dfff1a32
7c7eb1d692fde6db036ff44f36ac25e0b01ebe40043c5c1b78c4feb53ce1952fddfae3223b982cc9126e7fb2
c899e5bf8d91f3616e681158824c3a2816c7ee1f4908e1a002c5cc4f4b7e20557f0eebebe9a788f20ceac80e
f1f3bc760dc0e2880146d46820b8fac2cb52f68cf44b160bcc94b660a0a37e1ad95c81L
INFO:
S<%'������A���sU��A@
@                    @S���L����"ю���e+@@▒Ф@@
 N@@O@@□@@@@}T3@%@K@Q_@Z@'d@c@1Tg@#H@@+@.@@@m@g@"@@/H@D@1C@@@@
```


这里就属于CTF-RSA-tool工具的进阶理解了：

首先根据回显DEBUG: factor N: try Common factor between ciphertext and modulus attack我在文件factor_N.py找到了对应的部分：

```python
# 密文与模数不互素
def comfact_cn(N, c):
    log.debug('factor N: try Common factor between ciphertext and modulus attack')
    # Try an attack where the public key has a common factor with the ciphertext - sourcekris
    if c:
        commonfactor = libnum.gcd(N, c)
        if commonfactor > 1:
            q = commonfactor
            p = N / q
            return p, q
```


原来是密文与模数不互素，难怪我在http://www.factordb.com/index.php网站中分解不出N。而且也不是像以前积累的2021年9月广州羊城杯的BigRsa一样有两个N的模不互素：



我们修改一下CTF-RSA-tool打印出p、q：

```
# 密文与模数不互素
def comfact_cn(N, c):
    log.debug('factor N: try Common factor between ciphertext and modulus attack')
    # Try an attack where the public key has a common factor with the ciphertext - sourcekris
    if c:
        commonfactor = libnum.gcd(N, c)
        if commonfactor > 1:
            q = commonfactor
            p = N / q
            print('p=',p)
            print('q=',q)
            return p, q
```

打印p、q然后常规RSA解密

然后用常规的脚本来计算试一下，结果错误又出现了，除数太大，精度丢失了，只有前缀flag了：

import libnum

from Crypto.Util.number import long_to_bytes

q =
11554435340107681330370795502680996038121623273618972020169179464072451867699676554698060

p =
15029060827099243984405482330315426379419780356169578605686061517457518127716003222285950

e = 65537

c =
69449671088154377354289412867841194031383197134557321559250559286465369625976729418058310

n =
17365231154926348364478276872558492775911760603002394353723603461898405740234715001820110

d = libnum.invmod(e, (p - 1) * (q - 1)) #invmod(a, n) - 求a对于n的模逆,这里逆向加密过程中计算ψ(n)=(p-1)(q-1)，对ψ(n)保密,也就是对应根据e*d=1modψ(n),求出d

#print(hex(d))

m = pow(c, d, n) # 这里的m是十进制形式,pow(x, y[, z])--函数是计算 x 的 y 次方，如果 z 在存在，则再对结果进行取模，其结果等效于 pow(x,y) %z，对应前面解密算法中M=D(C)=（C^d）mod n

print(long_to_bytes(m/(2021*1001*p)))

```
└─$ python 1.py
b'flag{M`\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00'
```

然后一开始我的思路是用在线大数除法工具解决精度丢失的问题，找不到。于是进一步修改CTF-RSA-tool工具，终于在RSAutils.py中找到输出结果部分修改一下，成功输出flag：

```
    # 分解得到p q，或用户输入了p和q，计算d
        if self.p and self.q:
            self.d = libnum.invmod(self.e, (self.p - 1) * (self.q - 1))
            log.debug('d我在这里啊！！！=' + hex(self.d))
        else:
            log.error('can not factor N, please offer p and q or d')
            return

    # --private 是否需要打印私钥
    if self.args.private:
        log.info('\np=%d\nq=%d\nd=%d\n' % (self.p, self.q, self.d))
        log.info('private key:\n' + RSA.construct((long(self.n), long(self.e),
                                                    long(self.d), long(self.p), long(self.q))).export

  # 不需要解密，直接返回
    if not self.c:
        return
    self.plain = pow(self.c, self.d, self.n)
  # 打印解密出来的明文
    print '666'
    print 'plain =',self.plain
    print '666'
    log.info(libnum.n2s(self.plain/(2021*1001*self.q)))
```

标记1

标记2和对应修改

```
# d泄露攻击，根据过期的(N，e1，d1)，和一个新的e2，返回d2
```

CSDN @沐一一沐，一沐沐一

```
9561978986910375048128808860432864671882543L)
('q=', 150290608270992439844054823303154263794197803561695786056860615174575181277160032222
8595323354544869143578508493430361738389608201808675951696236703639637323159015879466395771
0720278031774852570940715332746360154801232194575939241684608918952215185113882137755142796
0151260776474250605261723480167088408148729L)
DEBUG: d我在这里啊！！！=0x233648b8421f05cf2307820eedcfae249e38abb155880d8f9f44622103ad8e9e
1375f6abc860c5ef6af69fb29958a3526c0a60923bd30630c66e624275d3dd7125698ca508003c3b3c8316be8c4
e13340a9b2c3a387f4e3649742a8b7e9f440bee7260c42195b5e601d08b083c559a5acd10cdc17e1cbac588704a
fc9dfff1a324f37c7eb1d692fde6db036ff44f36ac25e0b01ebe40043c5c1b78c4feb53ce1952fddfae3223b982
cc9126e7fb2b97c899e5bf8d91f3616e681158824c3a2816c7ee1f4908e1a002c5cc4f4b7e20557f0eebebe9a78
8f20ceac80ea7af1f3bc760dc0e2880146d46820b8fac2cb52f68cf44b160bcc94b660a0a37e1ad95c81L
666
plain = 8395571914768579416654762333305535017137164972943391521259329928396505927214851604 0
3872136110418315458948348727530876368684706703897783475419961094360953385279781752500013759
9792288206687311308578772408754460921111852885408000831747915336593162066492546425615088065
1344398199415001538769542757627221834716158111019220573707210169792787358775228739258183832
2987297150976125070114122865
666
INFO: flag{Math_1s_1nterest1ng_hah}
```

CSDN @沐一一沐，一沐沐一

2021年9月绿城杯，CRYPTO的RSA-2：（费马分解算法）

下载附件，是逻辑平铺类型：

打开文件，发现是分成两段的RSA加密，第一段加密flag[:20]，第二段加密flag[20:]：

```
from Crypto.Util.number import *
import gmpy2
from flag import flag
assert flag[:5]==b'flag{'
m1 = bytes_to_long(flag[:20])
p = getPrime(512)
p1 = gmpy2.next_prime(p)
q = getPrime(512)
q1 = gmpy2.next_prime(q)
n1 = p*q*p1*q1
print('n1 =',n1)
e = 0x10001
c1 = pow(m1,e,n1)
print('c1 =',c1)
m2 = bytes_to_long(flag[20:])
p2 = getPrime(1024)
q2 = getPrime(1024)
print('p2+q2 =',p2+q2)
print('q2*q2 =',p2*q2)
n2 = p2*p2*q2*q2*q2
print('n2 =',n2)
c2 = pow(m2,e,n2)
print('c2 =',c2)
#n1 =
6348779979606280884589422188738902470575876294643492831465947360363568026280963989291591
#c1 =
6201882078995455673376327652982610102807874783073703018551044780440620679217833227711395
```

#p2+q2 =
27477314676113846270813758230909738643779389179369138303385652430301081129410193345482 44

#q2*q2 =
18514724270030962172566965941723224386374076294232652258701085781018776172843355920566 03

#n2 =
40588227045595304080360385041082238507044292731344465815296032905633525556943787610712 65

#c2 =
25591090168544821761746024178724660839590948190451329227481168576490717242294520739865 60

先看第一段：

gmpy2.next_prime(p)函数的意思是取邻近p的下一个素数，那么很明显p和p1，q和q1都是两两相邻的素数，相差很小，n1是2048bit级别的，没法硬钢。

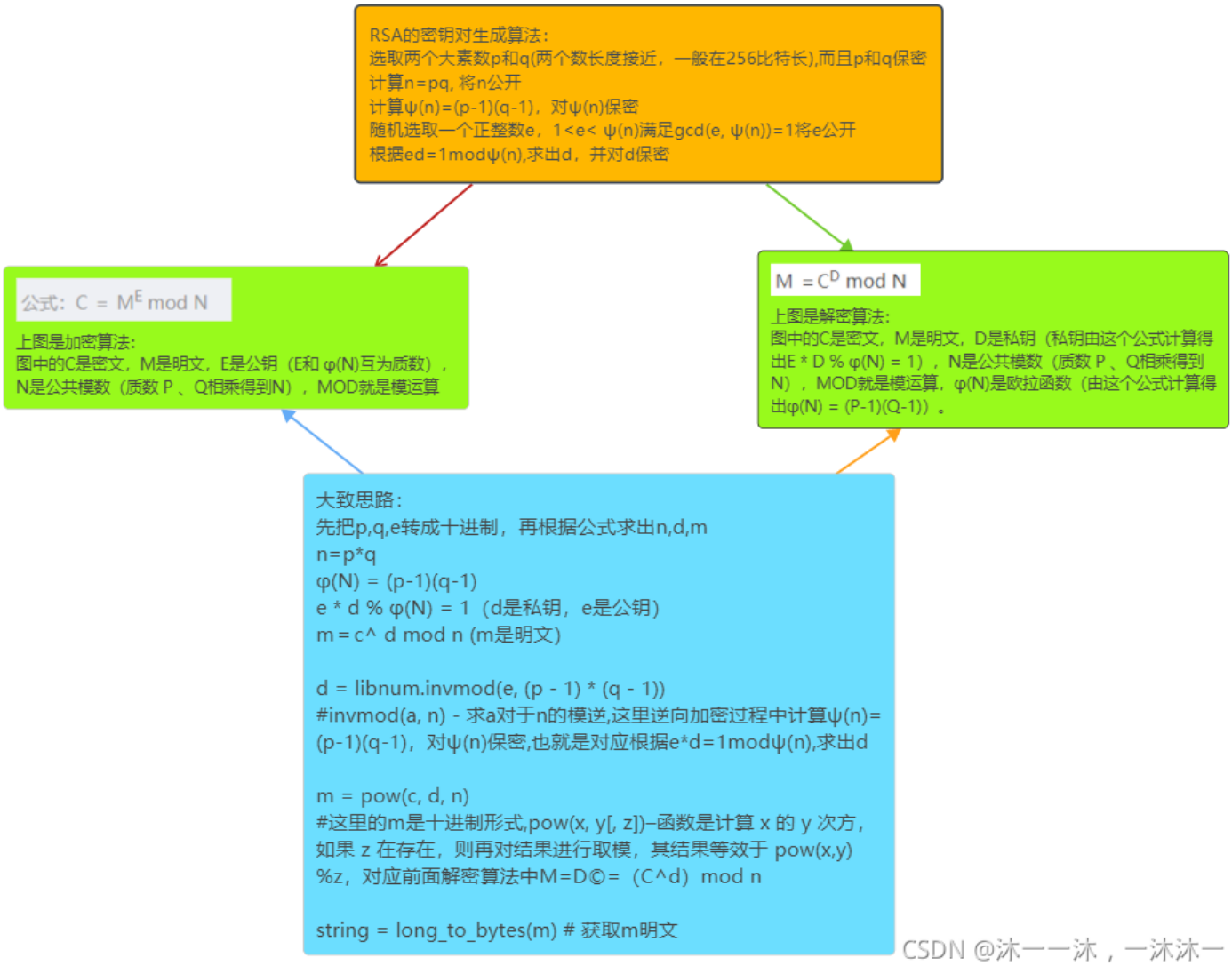
后半段的n2和前半段的加密逻辑差别很大，所以不是两个n的模不互素。也不是像上一道题RSA1一样的密文与模数不互素。

```
m1 = bytes_to_long(flag[:20])
p  = getPrime(512)
p1 = gmpy2.next_prime(p)
q  = getPrime(512)
q1 = gmpy2.next_prime(q)
n1 = p*q*p1*q1
print('n1 =',n1)
e = 0x10001
c1 = pow(m1,e,n1)
print('c1 =',c1)
```


然后查了资料后发现关键在p和p1，q和q1都是两两相邻的素数这里，也就是说p,q,p1,q1两两互素。单纯的factordb.com和yafu分解n肯定是分解不出来的，毕竟是2048bit嘛。所以我们需要一些算法，比如前面模不互素的欧几里得算法这样，这里我们要用的是费马分解算法。

费马分解算法的特征就是n是4个数的乘积，分解n之后我们会得到p、p1、q、q1四组隔开的排列组合，但是我们的脚本可以把组合限定成 p * q1， p1 * q 和 p * q ，p1 * q 这样。然后通过欧几里得算法求公因子的封装函数 gcd(pq1,pq) = p、gcd(p1q,p1q) = q 求出两组各一个数，然后就可以求出 $\varphi(n)=\varphi(p)\cdot\varphi(p1)\cdot\varphi(q)\cdot\varphi(q1)=(p-1)\cdot(q-1)\cdot(p1-1)\cdot(q1-1)$ 了。

然后后面的参考以前积累的RSA解密流程做即可：



费马因子分解代码：

取自博客：

https://blog.csdn.net/weixin_56678592/article/details/120555169?utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-1.no_search_link&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-1.no_search_link

```
import gmpy2

from gmpy2 import *

from Crypto.Util.number import *

import sympy

e = 0x10001

n1 = 63487799796062808845894221887389024705758762946434928314659473603635680262809639892915911

c1 = 62018820789954556733763276529826101028078747830737030185510447804406206792178333227711395

def fermat_factorization(n):
```

```
factor_list = []
get_context().precision = 2048 #这里应该是n1的数量级吧，len(bin(n1))就等于2048bit。
sqrt_n = int(sqrt(n))
c = sqrt_n
while True:
c += 1
d_square = c**2 - n
if is_square(d_square):
d_square = mpz(d_square)
get_context().precision = 2048
d = int(sqrt(d_square))
factor_list.append([c+d,c-d])
if len(factor_list)==2:
break
return factor_list
factor_list = fermat_factorization(n1)
[X1,Y1] = factor_list[0] #费马函数分解最大的特征就是输出两组互相交叉的p * q1， p1 * q,和p * q ，p1 * q，我们必须用gcd(X1,X2)和gcd(Y1,Y2)求出对应的p、q
[X2,Y2] = factor_list[1]
assert X1*Y1 == n1
assert X2*Y2 == n1
p1 = gcd(X1,X2)
q1 = X1 // p1 #这是python向下取整除运算符,我真的第一次发现python有这个运算符。
p2 = gcd(Y1,Y2)
q2 = Y1 // p2
phi1 = (p1-1)*(q1-1)*(p2-1)*(q2-1) #求φ(n)
d1 = invert(e,phi1) #常规RSA解密流程求d
print(long_to_bytes(gmpy2.powmod(c1,d1,n1)),end='') #常规RSA解密流程求明文
#flag{Euler_funct1ons
```

结果：

```
└─$ python 2.py
b'flag{Euler_funct1ons'
```

接下来分析后半段：

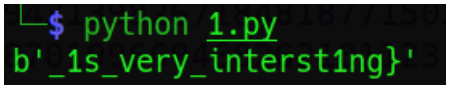
后半段给了x+y和x*y的值，按照初中数学的逻辑直接列个方程就可以解出x和y的值了，但是我看别人博客各种算法来解这个简单的逻辑，什么欧拉函数，因式方程看得头都晕。

然后翻着翻着突然发现一个秀儿，他p2和q2求法是这样的，因为n2=p2*p2*q2*q2*q2，所以 q2 = n2 // (p2_mul_q2*p2_mul_q2)、p2 = p2_mul_q2 // q2，真的我都鼓掌了，太棒了，简单题就简单做啊，简单的逻辑就应该这样求才对啊。

所以直接套用脚本即可，这里要注意的是φ(n)这里，因为因子只有p2和q2，所以φ(n)=p2*(p2 - 1)*q2*q2*(q2 - 1)，只用对单个因子减1即可。

```
import libnum

from Crypto.Util.number import long_to_bytes

n2=40588227045595304080360385041082238507044292731344465815296032905633525556943787610711

p2_add_q2=27477314676113846270813758230909738643779389179369138303385652430301081129410

p2_mul_q2=18514724270030962172566965941723224386374076294232652258701085781018776172843

e = 0x10001

c2 =
25591090168544821761746024178724660839590948190451329227481168576490717242294520739865600

q2 = n2 // (p2_mul_q2*p2_mul_q2) #//是python的除法下取整运算符

p2 = p2_mul_q2 // q2

d2 = libnum.invmod(e, p2*(p2 - 1)*q2*q2*(q2 - 1)) #invmod(a, n) - 求a对于n的模逆,这里逆向加密过程中计算ψ(n)=(p-1)(q-1)，对ψ(n)保密,也就是对应根据e*d=1modψ(n),求出d

m = pow(c2, d2, n2) # 这里的m是十进制形式,pow(x, y[, z])--函数是计算 x 的 y 次方，如果 z 在存在，则再对结果进行取模，其结果等效于 pow(x,y) %z，对应前面解密算法中M=D(C)=（C^d）mod n

string = long_to_bytes(m) # 获取m明文

print(string)
```

结果：



2021年10月广东强网杯，CRYPTO的RSA AND BASE?：（排列组合算法、下标对应解密）

下载附件，是一个txt文件，打开，发现RSA密文和类似base32的变码表，也符合题目暗示：

RSA:
n=566612435194265632999200581340928623707373979499472103948430218564774209596151325536108
e=362697880447032674261773409928261721401744043905777362814788913816122942076668915290199
c=137954301101369152742229874240507191901061563449586247819350394387527789763579249250710

BASE:
"GHI45FQRSCX****UVWJK67DELMNOPAB3"

CSDN @若秀

RSA题目照例先用CTF-RSA-TOOL工具跑一下，发现跑得出来：

```
└─$ python2 solve.py --verbose  -i /home/wdnmd/桌面/1.txt
DEBUG: factor N: try past ctf primes
DEBUG: factor N: try Gimmicky Primes method
DEBUG: factor N: try Wiener's attack
DEBUG: d = 0x1678d3c2f5a164ffffffffffffffffffffffffffffffffffffffffffffffffdL
INFO: flag{TCMDIEOH2MJFBLKHT2J7BLYZ2WUE5NYR2HNG====}
```

这应该是一层的flag，而且看起来像base32的四个等号，联想题目和TXT文件中后面的BASE码，可以猜出是BASE32变码的加密，而且还有4位未知：

RSA:
n=566612435194265632999200581340928623707373979499472103948430218564774209596151325536108
e=362697880447032674261773409928261721401744043905777362814788913816122942076668915290199
c=137954301101369152742229874240507191901061563449586247819350394387527789763579249250710

BASE:
'GHI45FQRSCX****UVWJK67DELMNOPAB3"

BASE32变码表

CSDN @若秀

首先通过和密文以及传统base32的码表对比可以发现缺了2 T Y Z四个字母，这四个字母共有24种排列组合，可以用下面代码求出排列组合：

```
list1=['2','T','Z','Y']

len1=len(list1)

list2=[]

w=""

for i in list1:
```

for j in list1:

if j!=i:

for k in list1:

if k!=j and k!=i:

for l in list1:

if l !=k and l !=j and l !=i:

w=i+j+k+l

list2.append(w)

print(list2)

print(len(list2))

结果：

```
└─$ python 4.py                                                          1 ×
['2TZY', '2TYZ', '2ZTY', '2ZYT', '2YTZ', '2YZT', 'T2ZY', 'T2YZ', 'TZ2Y', 'TZY2', 'TY2Z', 'T
YZ2', 'Z2TY', 'Z2YT', 'ZT2Y', 'ZTY2', 'ZY2T', 'ZYT2', 'Y2TZ', 'Y2ZT', 'YT2Z', 'YTZ2', 'YZ2T
', 'YZT2']
24
```

然后就是获取BASE32的编码实现来替换码表了，由于我在网上找不到base32的python编码实现，而且我也不会GO语言，没法直接替换封装函数码表，所以我用了我广州羊城杯的BabySmc技巧，通过下标对应来转为传统的base32密文。

下标对应法，这就要了解base32加密解密的本质了。

base32加密是5*8变8*5，获取的5个8位数对应着0~32内的范围，而base32的基本字符表ABCDEFGHIJKLMNOPQRSTUVWXYZ234567不过是0~32范围内对应的映射下标而已。

解密的时候也是用每个加密字符在0 ~32范围的数来拆分解密，关键就是这个解密时的5位数是怎么找的呢？是通过base32.index('加密字符')来找对应的0 ~ 32的下标。

所以加密字符表单的作用只是用来根据下标来映射5位数的0~32的范围而已，本质是0 ~64的下标。

那么我们就可以通过一一对应的方法来把题目中新的加密表单的下标来对应原生的base32加密下标了，因为base32在线解密工具只能通过base32.index('加密字符')来找0 ~ 32的下标。

脚本如下，注意抽出'='来，并且在最后要把'='加上去，因为base32解密必须是8的倍数：

import base64

#key1="GHI45FQRSCX****UVWJK67DELMNOPAB3" #少了2TYZ

key1="GHI45FQRSCX" #变形表单1

key2="UVWJK67DELMNOPAB3" #变形表单2

base32="ABCDEFGHIJKLMNOPQRSTUVWXYZ234567" #传统表单

key3="TCMDIEOH2MJFBLKHT2J7BLYZ2WUE5NYR2HNG" #变形密文

```
list1=['2TZY', '2TYZ', '2ZTY', '2ZYT', '2YTZ', '2YZT', 'T2ZY', 'T2YZ', 'TZ2Y', 'TZY2', 'TY2Z', 'TYZ2', 'Z2TY',
'Z2YT', 'ZT2Y', 'ZTY2', 'ZY2T', 'ZYT2', 'Y2TZ', 'Y2ZT', 'YT2Z', 'YTZ2', 'YZ2T', 'YZT2']

secret=""

for i in list1:

key4=key1+i+key2 #最终变形表单

#print(key4)

for m in key3:

g=key4.index(m)

secret+=base32[g]

secret+='====' #注意抽出'='来，并且在最后要把'='加上去，因为base32解密必须是8的倍数

print(secret)

print(base64.b32decode(secret))

secret=""
```

结果：

```
MJZWCX3BNZSF6YTBMNSV6YOLNRPXE2OHNB2A====
b'bsa_and_bace_a\xcbl_ri\xc7ht'
MJZWCX3BOZSF6YTBMOSV6YNLORPXE2NHOB2A====
b'bsa_avd_bac\xa5_a\xabt_ri\xa7pt'
OJZWCX3BNZSF6YTBONSV6YMLNRPXE2MHNB2A====
b'rsa_and_base_a\x8bl_ri\x87ht'
NJZWCX3BOZSF6YTBNOSV6YMLORPXE2MHOB2A====
b'jsa_avd_bak\xa5_a\x8bt_ri\x87pt'
NJZWCX3BMZSF6YTBNMSV6YLOMRPXE2LHMB2A====
b'jsa_afd_bak%_and_rig`t'
OJZWCX3BMZSF6YTBOMSV6YLNMRPXE2LHMB2A====
b'rsa_afd_bas%_amd_rig`t'
MJZWCX3BNZSF6YTBMNSV6YLONRPXE2LHNB2A====
b'bsa_and_bace_anl_right'
MJZWCX3BOZSF6YTBMOSV6YLNORPXE2LHOB2A====
b'bsa_avd_bac\xa5_amt_rigpt'
OJZWCX3BNZSF6YTBONSV6YLMNRPXE2LHNB2A====
b'rsa_and_base_all_right'
NJZWCX3BOZSF6YTBNOSV6YLMORPXE2LHOB2A====
b'jsa_avd_bak\xa5_alt_rigpt'
```

找出有含义的结果

ECC椭圆曲线加密：

攻防世界之easy_ECC：（ECC加密）

下载附件，打开：

已知椭圆曲线加密Ep(a,b)参数为

p = 15424654874903

a = 16546484

b = 4548674875

G(6478678675,5636379357093)

私钥为

k = 546768

求公钥K(x,y)

椭圆曲线ECC加密，没接触过，不懂原理，主要是找不到集成脚本，算了，浏览中发现一篇讲得透彻的博客：

https://blog.csdn.net/weixin_30951231/article/details/95919343

这里直接使用脚本：

```
#!/usr/bin/env python3
# -*- coding: UTF-8 -*-
# @Time :2020/9/28
# @Author :PeterJoin
import collections
import random
EllipticCurve = collections.namedtuple('EllipticCurve', 'name p a b g n h')
def banner():
print(("""%s
_____ _____ _____
| ____/ ___/ ___|
| _|| | | |
| |__| |__| |___
|_____\____\____|
%s%s
```

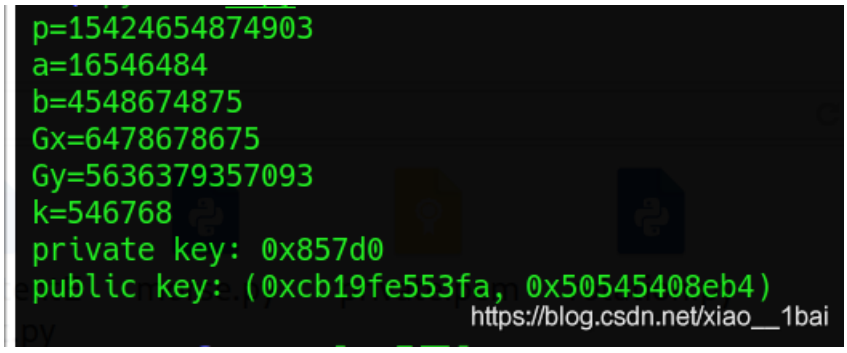
```
# Coded By PeterJoin -椭圆曲线加密（´・ω・）%s
""" % ('\033[91m', '\033[0m', '\033[93m', '\033[0m')))
curve = EllipticCurve(
'secp256k1',
# Field characteristic.
p=int(input('p=')),
# Curve coefficients.
a=int(input('a=')),
b=int(input('b=')),
# Base point.
g=(int(input('Gx=')),
int(input('Gy='))),
# Subgroup order.
n=int(input('k=')),
# Subgroup cofactor.
h=1,
)
# Modular arithmetic ########################################################
def inverse_mod(k, p):
"""Returns the inverse of k modulo p.
This function returns the only integer x such that (x * k) % p == 1.
k must be non-zero and p must be a prime.
"""
if k == 0:
raise ZeroDivisionError('division by zero')
if k < 0:
# k ** -1 = p - (-k) ** -1 (mod p)
return p - inverse_mod(-k, p)
# Extended Euclidean algorithm.
s, old_s = 0, 1
t, old_t = 1, 0
```

```
r, old_r = p, k
while r != 0:
quotient = old_r // r
old_r, r = r, old_r - quotient * r
old_s, s = s, old_s - quotient * s
old_t, t = t, old_t - quotient * t
gcd, x, y = old_r, old_s, old_t
assert gcd == 1
assert (k * x) % p == 1
return x % p
# Functions that work on curve points #############################################
def is_on_curve(point):
"""Returns True if the given point lies on the elliptic curve."""
if point is None:
# None represents the point at infinity.
return True
x, y = point
return (y * y - x * x * x - curve.a * x - curve.b) % curve.p == 0
def point_neg(point):
"""Returns -point."""
assert is_on_curve(point)
if point is None:
# -0 = 0
return None
x, y = point
result = (x, -y % curve.p)
assert is_on_curve(result)
return result
def point_add(point1, point2):
"""Returns the result of point1 + point2 according to the group law."""
assert is_on_curve(point1)
```

```
assert is_on_curve(point2)
if point1 is None:
# 0 + point2 = point2
return point2
if point2 is None:
# point1 + 0 = point1
return point1
x1, y1 = point1
x2, y2 = point2
if x1 == x2 and y1 != y2:
# point1 + (-point1) = 0
return None
if x1 == x2:
# This is the case point1 == point2.
m = (3 * x1 * x1 + curve.a) * inverse_mod(2 * y1, curve.p)
else:
# This is the case point1 != point2.
m = (y1 - y2) * inverse_mod(x1 - x2, curve.p)
x3 = m * m - x1 - x2
y3 = y1 + m * (x3 - x1)
result = (x3 % curve.p,
-y3 % curve.p)
assert is_on_curve(result)
return result
def scalar_mult(k, point):
"""Returns k * point computed using the double and point_add algorithm."""
assert is_on_curve(point)
if k < 0:
# k * point = -k * (-point)
return scalar_mult(-k, point_neg(point))
result = None
```

```
addend = point
while k:
if k & 1:
# Add.
result = point_add(result, addend)
# Double.
addend = point_add(addend, addend)
k >>= 1
assert is_on_curve(result)
return result
# Keypair generation and ECDHE ##################################################
def make_keypair():
"""Generates a random private-public key pair."""
private_key = curve.n
public_key = scalar_mult(private_key, curve.g)
return private_key, public_key
private_key, public_key = make_keypair()
print("private key:", hex(private_key))
print("public key: (0x{:x}, 0x{:x})".format(*public_key))
if __name__ == '__main__':
banner()
make_keypair()
```

运行之后照着输入题目附件给的参数即可：



解出的x+y就是flag了，原理太难了，以后有机会再接触。

LFSR反馈移位寄存器类型：

攻防世界之streamgame1：（CTF中的LFSR考点(一)、文件读取对齐的二进制）

下载附件，是典型的LFSR类型：

```
from flag import flag

assert flag.startswith("flag{")

# 作用：判断字符串是否以指定字符或子字符串开头flag{

assert flag.endswith("}")

# 作用：判断字符串是否以指定字符或子字符串结尾}，flag{}，6个字节

assert len(flag)==25

# flag的长度为25字节，25-6=19个字节

#3<<2可以这么算，bin(3)=0b11向左移动2位变成1100，0b1100=12(十进制)

def lfsr(R,mask):

output = (R << 1) & 0xffffff #将R向左移动1位，bin(0xffffff)='0b111111111111111111111111'=0xffffff的二进制补码

i=(R&mask)&0xffffff #按位与运算符&：参与运算的两个值,如果两个相应位都为1,则该位的结果为1,否则为0

lastbit=0

while i!=0:

lastbit^=(i&1) #按位异或运算符：当两对应的二进位相异时，结果为1

i=i>>1

output^=lastbit

return (output,lastbit)

R=int(flag[5:-1],2)

mask = 0b10100110001000110 0

f=open("key","ab") #以二进制追加模式打开

for i in range(12):

tmp=0

for j in range(8):

(R,out)=lfsr(R,mask)

tmp=(tmp << 1)^out #按位异或运算符：当两对应的二进位相异时，结果为1

f.write(chr(tmp)) #chr() 用一个范围在 range（256）内的（就是0～255）整数作参数，返回一个对应的字符。

f.close()
```

明文key如下，需要转成能推出初始状态值(flag)的对应位数的flag：

LFSR类型我在后面非传统密码类型解析中已经总结过了，所以这里直接应用前面的总结步骤仿照着做即可：

```
from flag import flag
assert flag.startswith("flag{")
# 作用：判断字符串是否以指定字符或子字符串开头flag{
assert flag.endswith("}")
# 作用：判断字符串是否以指定字符或子字符串结尾}，flag{}，6个字节
assert len(flag)==25
# flag的长度为25字节，25-6=19个字节
#3<<2可以这么算，bin(3)=0b11向左移动2位变成1100，0b1100=12(十进制)
def lfsr(R,mask):
    output = (R << 1) & 0xffffff    #将R向左移动1位，bin(0xffffff)='0b111111111111111111111111'=0xffffff的二进制补码
    i=(R&mask)&0xffffff             #按位与运算符&：参与运算的两个值,如果两个相应位都为1,则该位的结果为1,否则为0
    lastbit=0
    while i!=0:
        lastbit^=(i&1)    #按位异或运算符：当两对应的二进位相异时，结果为1
        i=i>>1
    output^=lastbit
    return (output,lastbit)


R=int(flag[5:-1],2)
mask    =   0b10100110001000110 0
f=open("key","ab")    #以二进制追加模式打开
for i in range(12):
    tmp=0
    for j in range(8):
        (R,out)=lfsr(R,mask)
        tmp=(tmp << 1)^out    #按位异或运算符：当两对应的二进位相异时，结果为1
    f.write(chr(tmp))  #chr() 用一个范围在 range（256）内的（就是0～255）整数作参数，返回一个对应的字符。
f.close()
```

但是flag不可能有76位这么长，所以这19位应该是2进制数，这才能对应后面的mask的19位为一个循环。

flag长度19字节，4*19=76位

同样的逻辑，out左移动一位，右边补上lastbit值

可以推出反馈函数lastbit=R3⊕R4⊕R5⊕R9⊕R13⊕R14⊕R17⊕R19

也就是说19位为一个循环，那么key值只要取前19位即可反推flag

补充：

最后的 12 内嵌 8 循环一共产生12个字符。其中的前 3 个字符的前 19 位二进制数是一个加密循环。19 位后面的数是继续用这 19 个加密后的二进制数继续新一轮加密，就是多层加密。所以我们取 19 位即可，结果 flag 也是 19 个二进制数。

关键就是这个key值怎么取，一开始我直接复制粘贴字符转16进制，然后取前19位，可想而知当然是错的了：

然后查了好些资料发些他们从文件中读取二进制数的代码还是不够简便，于是我吸取了我前面base64编码的代码写了一串从文件中读取对齐的二进制的代码，以后也直接拿来用即可。

(在线转换都会省略最开头的0导致结果位数错误，进而导致结果错误，所以自己要注意。)

f = open('5key','rb') #以二进制格式打开文件

content = f.read() #读取的是\xhh类型的十六进制

key=['{:0>8}'.format(str(bin(i)).replace('0b','')) for i in content] #从base64编码汲取的经验，二进制8位对齐。

print(''.join(key)[:19])

最终解题脚本，还是一样要注意小端顺序：

f = open('5key','rb') #以二进制格式打开文件

content = f.read() #读取的是\xhh类型的十六进制

key=['{:0>8}'.format(str(bin(i)).replace('0b','')) for i in content] #从base64编码汲取的经验，二进制8位对齐。

key1=''.join(key)[:19] #lastbit全部值，就是反馈函数生成的值，32位的key1

key2=key1

flag=[]

for i in range(19):

output='?'+key1[:18] #?01000001111110111101110111111000,因为后面有key1=str(lastbit)+key1[:31]，key1不断填补，output不断取前31位，所以这里output每次把?定在第i位上，注意output和key1是独立的分开的。

flag.append(str(int(key2[-1-i])^int(output[-3])^int(output[-4])^int(output[-5])^int(output[-9])^int(output[-13])^int(output[-14])^int(output[-17])))

#这里之所以取负数是因为我们截取是从左往右取，而在计算机中是小端顺序，应该从右往左取才对，这里的key2[-1-i]就是小端左到右的第i位，也就是前面分析的反馈函数生成值的第i位。flag值的原第i位,现在在第19位，可以通过⊕的可逆性来求，就是R19=lastbit ⊕ R3 ⊕ R4 ⊕ R5 ⊕ R9 ⊕ R13 ⊕ R14 ⊕ R17

key1=str(flag[i])+key1[:18] #不断填补key1,让key1向右推进，

print("flag{"+bin(int(''.join(flag[::-1]),2)).replace('0b','')+"}") #这里是经过一系列操作，首先把flag变成小端顺序的[::-1]，然后就是转十六进制。

结果：

```
└─$ python 2.py
flag{1110101100001101011}
```

攻防世界之streamgame2：（CTF中的LFSR考点(一)、文件读取对齐的二进制）

下载附件，还是典型的LFSR类型：

from flag import flag

assert flag.startswith("flag{")

assert flag.endswith("}")

assert len(flag)==27

def lfsr(R,mask):

output = (R << 1) & 0xffffff

i=(R&mask)&0xffffff

lastbit=0

while i!=0:

lastbit^=(i&1)

i=i>>1

output^=lastbit

return (output,lastbit)

R=int(flag[5:-1],2)

mask=0x100002

f=open("key","ab")

for i in range(12):

tmp=0

for j in range(8):

(R,out)=lfsr(R,mask)

tmp=(tmp << 1)^out

f.write(chr(tmp))

f.close()

原理同streamgame1，这里唯一的不同就是mask也是要截取的，因为mask要看成常数，要求的是21位初始值，所以24位的mask要截取成21位。

```
1 from flag import flag
2 assert flag.startswith("flag{")
3 assert flag.endswith("}")
4 assert len(flag)==27
5
6 def lfsr(R,mask):
7     output = (R << 1) & 0xffffff
8     i=(R&mask)&0xffffff
9     lastbit=0
10    while i!=0:
11        lastbit^=(i&1)
12        i=i>>1
13    output^=lastbit
14    return (output,lastbit)
15
16
17
18 R=int(flag[5:-1],2)
19 mask=0x100002
20
21 f=open("key","ab")
22 for i in range(12):
23     tmp=0
24     for j in range(8):
25         (R,out)=lfsr(R,mask)
26         tmp=(tmp << 1)^out
27     f.write(chr(tmp))
28 f.close()
```

这里应该是21位二进制数

又是同样的补位逻辑

这里6位十六进制数共24位，应该取前21位即可。
前21位就是100000000000000000010，那么反馈函数就是lastbit=R2⊕R21

补充：

最后的 12 内嵌 8 循环一共产生12个字符。其中的前 3 个字符的前 21 位二进制数是一个加密循环。21 位后面的数是继续用这 21 个加密后的二进制数继续新一轮加密，就是多层加密。所以我们取 21 位即可，结果 flag 也是 21 个二进制数。

直接上脚本：

f = open('3key','rb') #以二进制格式打开文件

content = f.read() #读取的是\xhh类型的十六进制

key=['{:0>8}'.format(str(bin(i)).replace('0b','')) for i in content] #从base64编码汲取的经验，二进制8位对齐。

key1=''.join(key)[:21] #lastbit全部值，就是反馈函数生成的值，32位的key1

key2=key1

flag=[]

for i in range(21):

output='?'+key1[:20] #?01000001111110111101110111111000,因为后面有key1=str(lastbit)+key1[:31]，key1不断填补，output不断取前31位，所以这里output每次把?定在第i位上，注意output和key1是独立的分开的。

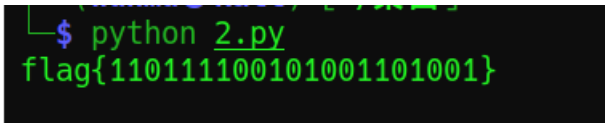
flag.append(str(int(key2[-1-i])^int(output[-2])))

#这里之所以取负数是因为我们截取是从左往右取，而在计算机中是小端顺序，应该从右往左取才对，这里的key2[-1-i]就是小端左到右的第i位，也就是前面分析的反馈函数生成值的第i位。flag值的原第i位,现在在第21位，可以通过⊕的可逆性来求，就是R21=lastbit ⊕ R2

key1=str(flag[i])+key1[:20] #不断填补key1,让key1向右推进，

print("flag{"+bin(int(''.join(flag[::-1]),2)).replace('0b','')+"}") #这里是经过一系列操作，首先把flag变成小端顺序的[::-1]，然后就是转十六进制。

结果：

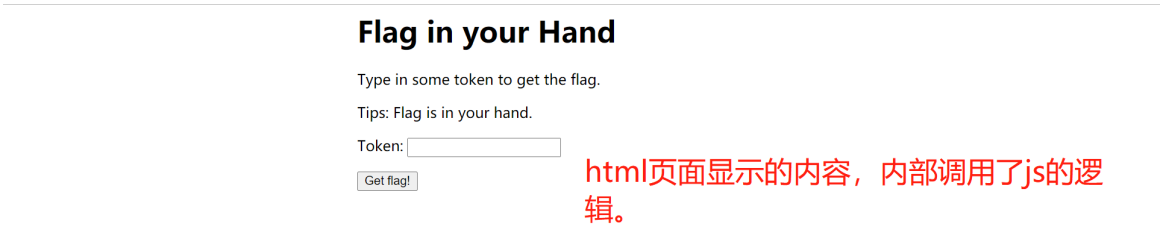


js类型加密

js逻辑平铺类型：

攻防世界之flag_in_your_hand1:（字符串中文含义暗示、冗余中锁定关键代码）

下载附件，解压，一个js一个html，两边出击，浏览器看html页面，记事本看js和html的逻辑和它们之间的关系：



html页面显示的内容，内部调用了js的逻辑。

```
        }
    </style>
    <script src="script-min.js"></script>
    <script type="text/javascript">
        var ic = false;
        var fg = "";

        function getFlag() {
            var token = document.getElementById("secToken").value;
            ic = checkToken(token);
            fg = bm(token);
            showFlag()
        }

        function showFlag() {
            var t = document.getElementById("flagTitle");
            var f = document.getElementById("flag");
            t.innerText = !!ic ? "You got the flag below!!" : "Wrong!";
            t.className = !!ic ? "rightflag" : "wrongflag";
            f.innerText = fg;
        }
    </script>
```

html代码，可以看到这里先用checkToken(token)对token进行一层操作，后面bm(token)对js进行二层操作，最后才显示flag。


```
function hm(s) {
    return rh(rstr(str2rstr_utf8(s)));
}
function bm(s) {
    return rb(rstr(str2rstr_utf8(s)));
}
function rstr(s) {
    return binl2rstr(binl(rstr2binl(s), s.length * 8));
}
function checkToken(s) {
    return s === "FAKE-TOKEN";
}
function rh(ip) {
    try {
        hc
    } catch (e) {
        hc = 0;
    }
    var ht = hc ? "0123456789ABCDEF" : "0123456789abcdef";
    var op = "";
    var x;
    for (var i = 0; i < ip.length; i++) {
        x = ip.charCodeAt(i);
        op += ht.charAt((x >>> 4) & 0x0F) + ht.charAt(x & 0x0F);
    }
    return op;
}
function rb(ip) {
    try {
        bp
    } catch (e) {
```

js代码中先看checkToken函数，发现是一个FAKE-TOKEN，中文暗示就是假的TOKEN。

bm函数，我在js代码那里跟踪啊跟踪，发现它是好多个函数的嵌套，没办法了，根本不知道关键逻辑放在哪里。查了资料说看跟踪ic，真是一语惊醒梦中人！ic是判断条件，归根结底还是判断ic,找ic即可：

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
    for (var i = 0; i < len; i += 3) {
        var t = (ip.charCodeAt(i) << 16) | (i + 1 < len ? ip.charCodeAt(i + 1) << 8 : 0) | (i + 2 < len ? ip.charCodeAt(i + 2
        for (var j = 0; j < 4; j++) {
            if (i * 8 + j * 6 > ip.length * 8)
                op += bp;
            else
                op += b.charAt((t >>> 6 * (3 - j)) & 0x3F);
        }
    }
    return op;
}
function ck(s) {
    try {
        ic
    } catch (e) {
        return;
    }
    var a = [118, 104, 102, 120, 117, 108, 119, 124, 48, 123, 101, 120];
    if (s.length == a.length) {
        for (i = 0; i < s.length; i++) {
            if (a[i] - s.charCodeAt(i) != 3)
                return ic = false;
        }
        return ic = true;
    }
    return ic = false;
}
function str2rstr_utf8(input) {
```

bm函数中引用了很多函数，基本把整个js逻辑串起来了，所以要自己锁定关键代码。

ic是判断条件，跟踪ic就可以发现判断逻辑也在这里。

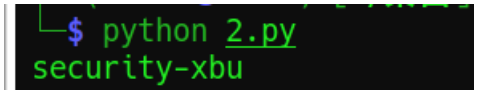
那么前面bm串起来的函数就是对用户输入的函数进行加密处理了，相较而言逆向判断条件更加容易。

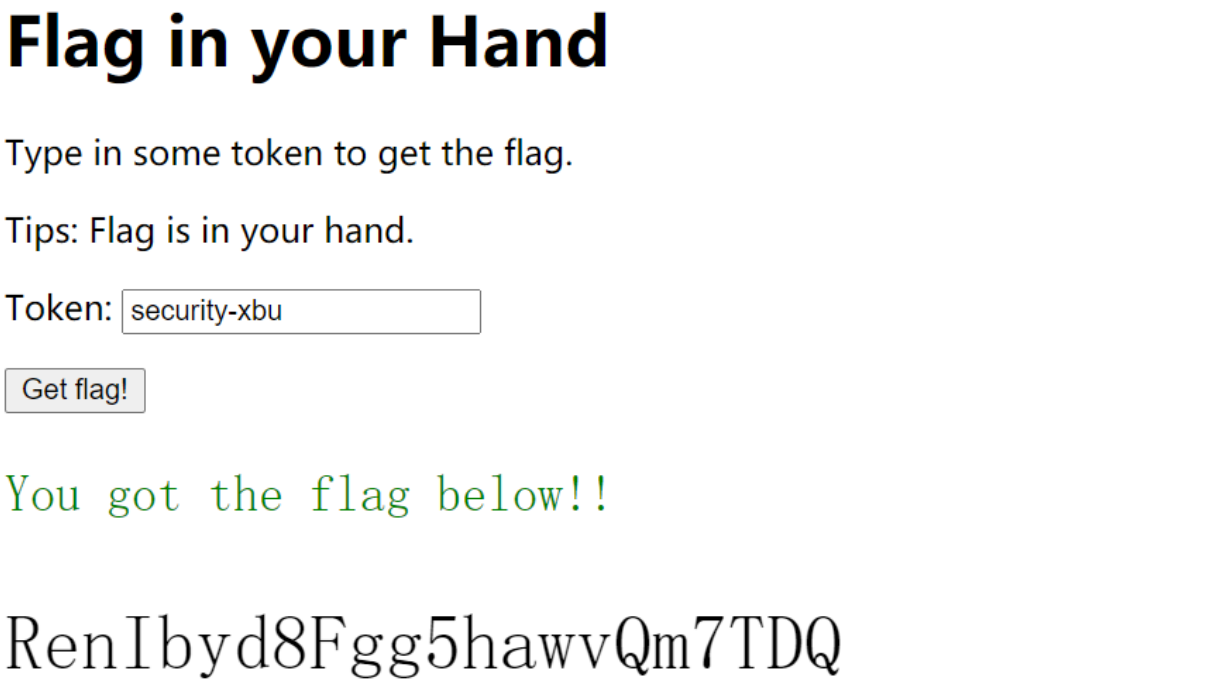
找到ic了，这个ic也的确在bm函数嵌套内，由于在学逆向，所以直接写逆向脚本即可：

key1=[118, 104, 102, 120, 117, 108, 119, 124, 48,123,101,120]

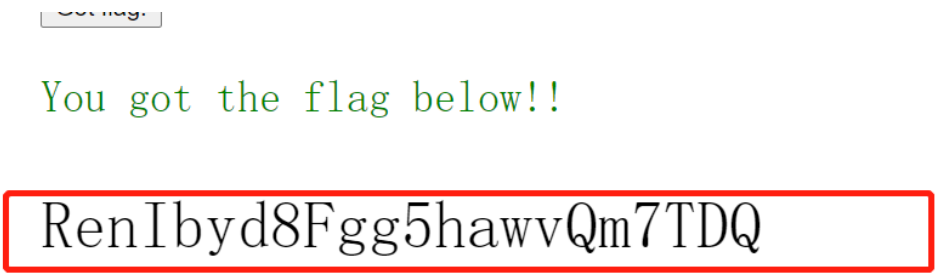
```
token=""
for i in key1:
token+=chr(i-3)
print(token)
```

结果：





最后回顾一下，为什么下面这个每次都有且每次都不一样呢，观察js代码可以发现每个js函数的接受参数s就是我们传入token，而fg生成函数bm嵌套了很多函数，所以直接逆向bm函数生成flag不现实，而参数token作为s传入后就会加密回显，所以输入错误的token也会显示出不同的加密串：



python类型逻辑加密

pyc文件反编译：

攻防世界之easychallenge：（源代码修改逻辑解密、）

下载附件，是pyc文件，于是要反编译，一开始看资料说用uncompyle6，我也不知道为什么我的老是报错，后来又找了个在线反编译，可以支持的Python版本比较多：

https://tool.lu/pyc/

反编译代码：

```
#!/usr/bin/env python

# visit https://tool.lu/pyc/ for more information

import base64

def encode1(ans):

s = ""

for i in ans:

x = ord(i) ^ 36

x = x + 25

s += chr(x)

return s

def encode2(ans):

s = ""

for i in ans:

x = ord(i) + 36

x = x ^ 36

s += chr(x)

return s

def encode3(ans):

return base64.b32encode(ans)

flag = " "

print "Please Input your flag:"

flag = raw_input()

final = "UC7KOWVXWVNKNIC2XCXKHKK2W5NLBKNOUOSK3LNNVWW3E==="

if encode3(encode2(encode1(flag))) == final:

print "correct"

else:

print "wrong"
```

本来是自己一层层改的，怎么加密就怎么逆过来解密，后来发现又犯了以前的错误，有源码就要用源码啊！！！

直接在源码上修改即可：

```
#!/usr/bin/env python

# visit https://tool.lu/pyc/ for more information

import base64

def decode1(ans):

s = ''

for i in ans:

x = ord(i) -25

x = x ^ 36

s += chr(x)

return s

def decode2(ans):

s = ''

for i in ans:

x = i ^ 36

x = x - 36

s += chr(x)

return s

def decode3(ans):

return base64.b32decode(ans)

final = 'UC7KOWVXWVNKNIC2XCXKHKK2W5NLBKNOUOSK3LNNVWW3E==='

print(decode1(decode2(decode3(final))) )
```

传统密码类型解析

凯撒密码(24个字母)：

特点:24个字母间的移动

在密码学中，恺撒密码（英语：Caesar cipher），或称恺撒加密、恺撒变换、变换加密，是一种最简单且最广为人知的加密技术。它是一种替换加密的技术，明文中的所有字母都在字母表上向后（或向前）按照一个固定数目进行偏移后被替换成密文。例如，当偏移量是3的时候，所有的字母A将被替换成D，B变成E，以此类推。

加密就是明文向后移动展现出对应的密文。

解密就是密文向前移动展现出对应的明文。

明文字母表	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
密文字母表	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

摩斯密码(只有01(无规则)或.-，空格或/做分隔符)：

摩尔斯电码也被称作摩斯密码，是一种时通时断的信号代码，通过不同的排列顺序来表达不同的英文字母、数字和标点符号。它发明于1837年，是一种早期的数字化通信形式。不同于现代化的数字通讯，摩尔斯电码只使用零和一两种状态的二进制代码，它的代码包括五种：短促的点信号“·”，读“滴”（Di）保持一定时间的长信号“-”，读“嗒”（Da）表示点和划之间的停顿、每个词之间中等的停顿，以及句子之间长的停顿。

摩斯电码：（格式要求：可用空格或单斜杠/来分隔摩斯电码，但只可用一种，不可混用）

摩 尔 斯 电 码 表

字符	电码符号	字符	电码符号	字符	电码符号
A	•—	N	—•	1	•————
B	—•••	O	———	2	••———
C	—•—•	P	•——•	3	•••——
D	—••	Q	——•—	4	••••—
E	•	R	•—•	5	•••••
F	••—•	S	•••	6	—••••
G	——•	T	—	7	——•••
H	••••	U	••—	8	———••
I	••	V	•••—	9	————•
J	•———	W	•——	0	—————
K	—•—	X	—••—	?	••——••
L	•—••	Y	—•——	/	—••—•
M	——	Z	——••	( )	—•——•—
				—	—••••—
				•	•—•—•—

云影密码(01248)：

此密码运用了1248代码,因为本人才疏学法,问未发现有过使用的先例,因此暂归为原创密码。由于这个密码,我和片风云影初识,为了纪念,将其命名为“云影密码”。

有了1,2,4,8这四个简单的数字,你可以以加法表示出0~9任何个数字,例如0=28,7=124,9=18这样,再用1-26来表示A-Z,就可以用作密码了。为了不至于混乱,我个人引入了第五个数字0,来用作间隔,以遍免翻译错误,所以还可以称“01248密码”

注意(3个及以上数字时)：

虽然是相加，但是可以在数字内不按顺序相加，如124可写成(12)4和1(24)结果分别是7和16，只要保证不大于26即可

题目(总不超过26)：

12401011801180212011401804

第一步分割：

即124 、1、118、118、212、114、18、4

第二步基本翻译：

例如124可以表示7,也可以表示16(但不可能是34,因为不会超过26),所以可以放弃来翻译其他没有异议的,可得:124、a、s、s、w、o、18、d

第三步推测得出明文：

可以推测后面的18表示r,前面的为p最合适。

所以最后明文:

password(密码)

栅栏密码(分组数作密钥):

所谓栅栏密码，就是把要加密的明文分成N个一组，然后把每组的第1个字连起来，形成一段无规律的话。 不过栅栏密码本身有一个潜规则，就是组成栅栏的字母一般不会太多。（一般不超过30个，也就是一、两句话）

传统栅栏密码(矩阵行列，密钥是行数):

假如有一个字符串：123456789

取字符串长度的因数进行分组，假如key=3

1 2 3 \\分组情况，每三个数字一组，分为三组

4 5 6

7 8 9

然后每一组依次取一个数字组成一个新字符串：147258369 \\加密完成的字符串

解题：

试一般的栅栏密码,取5为矩阵行数,得到" cyperrocaegireeol} eahfocec gnbip不正确，取5为矩阵列数,得到" cebgccfe en eohplprgecrayoii aoreg”,也不正确，除了常规的栅栏密码,还有一种w型的栅栏密码。

w型的栅栏密码(第一行是Key数，后面排成w型横竖读取):

同样一个字符串：123456789

key=3

1----5----9 \\让数字以W型组织，同样是三组，但每组的数量不一定相同

-2--4-6--8

--3----7--

加密密文：159246837

解题：

题目提示:栅栏密码,密钥长度为5

是将明文按照w型排列,并横(竖)向读取密文,其解密过程就是加密的逆过程,即将密文横(竖)向按一定规律排列后,以w型读取,对于此题,根据题目提示为以下5行。

c.....c.....e.....h.....g
.y.....a..e.....f..n.....p..e.....o..o
..b....e.....{....l.....c.....i.....r.....g.....}
...e..p.....r..i.....e..c....._..o
.....r.....a....._.....g

培根密码(大小写的ABab，而且必须是5个一组，不是5个就考虑摩斯密码):

培根所用的密码是一种本质上用二进制数设计的，没有用通常的0和1来表示，而是采用a和b

培根密码加密方式

第一种方式：

A aaaaa B aaaab C aaaba D aaabb E aabaa F aabab G aabba H aabbb I abaaa J abaab

K ababa L ababb M abbaa N abbab O abbba P abbbb Q baaaa R baaab S baaba T baabb

U babaa V babab W babba X babbb Y bbaaa Z bbaab

第二种方式:

a AAAAA g AABBA n ABBAA t BAABA

b AAAAB h AABBB o ABBAB u-v BAABB

c AAABA i-j ABAAA p ABBBA w BABAA

d AAABB k ABAAB q ABBBB x BABAB

e AABAA l ABABA r BAAAA y BABBA

f AABAB m ABABB s BAAAB z BABBB

举例

例1、 baabaaabbbabaaabbaaaaaaaaabbabaaaabaaaaaabaaabaabaaaabaabbbaabbbaababb

baaba aabbb abaaa bbaaa aaaaa abbab aaaab aaaaa abaaa baaba aaaba abbba abbba ababb

s h i y a n b a i s c o o l

附加解密Python脚本如下：

```
#!/usr/bin/python

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

import re

alphabet = ['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z']

first_cipher =
["aaaaa","aaaab","aaaba","aaabb","aabaa","aabab","aabba","aabbb","abaaa","abaab","ababa","ababb","abbaa","

second_cipher =
["aaaaa","aaaab","aaaba","aaabb","aabaa","aabab","aabba","aabbb","abaaa","abaaa","abaab","ababa","ababb","

def encode():

string = raw_input("please input string to encode:\n") #这里接收要加密的字符串

e_string1 = ""

e_string2 = ""

for index in string:

for i in range(0,26):

if index == alphabet[i]:

e_string1 += first_cipher[i]
```

```
e_string2 += second_cipher[i]
break
print "first encode method result is:\n"+e_string1
print "second encode method result is:\n"+e_string2
return
def decode():
e_string = raw_input("please input string to decode:\n") #这里接收要解密的字符串
e_array = re.findall(".{5}",e_string)
d_string1 = ""
d_string2 = ""
for index in e_array:
for i in range(0,26):
if index == first_cipher[i]:
d_string1 += alphabet[i]
if index == second_cipher[i]:
d_string2 += alphabet[i]
print "first decode method result is:\n"+d_string1
print "second decode method result is:\n"+d_string2
return
if __name__ == '__main__':
while True:
print "\t*******Bacon Encode_Decode System*******"
print "input should be lowercase,cipher just include a b"
print "1.encode\n2.decode\n3.exit"
s_number = raw_input("please input number to choose\n")
if s_number == "1":
encode()
raw_input()
elif s_number == "2":
decode()
raw_input()
```

elif s_number == "3":

break

else:

continue

与佛论禅编码，要加上佛曰：才能转换(BASE64类型转不了就ROT13一下)

网址：

与佛论禅

题目文字：

夜哆悉諳多苦奢陀奢諦冥神哆盧穆皤三侄三即諸諳即冥迦冥隸數顛耶迦奢若吉怯陀諳怖奢智侄諸若奢數菩奢集遠俱老竟寫明奢若梵等盧皤豆蒙密離怯婆皤礙他哆提哆多鉢以南哆心曰姪罰蒙吶神。舍切真怯勝吶得俱沙罰娑是怯遠得吶數罰輸哆遠薩得槃漫夢盧皤亦醯吶娑皤瑟輸諳尼摩罰薩冥大倒參夢侄阿心罰等奢大度地冥殿皤沙蘇輸奢恐豆侄得罰提哆伽諳沙楞鉢三死怯摩大蘇者數一遮

转换后的

MzkuM3gvMUAwnzuvn3cgozMlMTuvqzAenJchMUAeqzWenzEmLJW9

的确是BASE64类型，但是直接BASE64是转换不出来的，还要先ROT13一下，可以算是一个小混淆，长见识了。

转轮机加密：

特点是等长的分好组的乱序字母，原理是转齿轮把一个字母换成另一个来拼成一句话。

格式是这样的：

```
1: < ZWAXJGDLUBVIQHKYPNTCRMOSFE <
2: < KPBELNACZDTRXMJQOYHGVSFUWI <
3: < BDMAIZVRNSJUWFHTEQGYXPLOCK <
4: < RPLNDVHGFCUKTEBSXQYIZMJWAO <
5: < IHFRLABEUOTSGJVDKCPMNZQWXY <
6: < AMKGHIWPNYCJBFZDRUSLOQXVET <
7: < GWTHSPYBXIZULVKMRAFDCEONJQ <
8: < NOZUTWDCVRJLXKISEFAPMYGHBQ <
9: < XPLTDSRFHENYVUBMCQWAOIKZGJ <
10: < UDNAJFBOWTGVRSCZQKELMXYIHP <
11: < MNBVCXZQWERTPOIUYALSKDJFHG <
12: < LVNCMXZPQOWEIURYTASBKJDFHG <
13: < JZQAWSXCDERFVBGTYHNUMKILOP <

密钥为：2,3,7,5,13,12,9,1,8,10,4,11,6
密文为：NFQKSEVOQOFNP
```

脚本解题积累：

rotor = [#这里是要输入的转轮机原始字符串

"ZWAXJGDLUBVIQHKYPNTCRMOSFE", "KPBELNACZDTRXMJQOYHGVSFUWI",

"BDMAIZVRNSJUWFHTEQGYXPLOCK", "RPLNDVHGFCUKTEBSXQYIZMJWAO",

"IHFRLABEUOTSGJVDKCPMNZQWXY", "AMKGHIWPNYCJBFZDRUSLOQXVET",

"GWTHSPYBXIZULVKMRAFDCEONJQ", "NOZUTWDCVRJLXKISEFAPMYGHBQ",

"XPLTDSRFHENYVUBMCQWAOIKZGJ", "UDNAJFBOWTGVRSCZQKELMXYIHP",

```
"MNBVCXZQWERTPOIUYALSKDJFHG", "LVNCMXZPQOWEIURYTASBKJDFHG",
"JZQAWSXCDERFVBGTYHNUMKILOP"
]
cipher = "NFQKSEVOQOFNP" #这是要输入转轮机密文
key = [2,3,7,5,13,12,9,1,8,10,4,11,6] #这是要输入转轮机密钥
tmp_list=[]
for i in range(0, len(rotor)):
tmp=""
k = key[i] - 1
for j in range(0, len(rotor[k])):
if cipher[i] == rotor[k][j]:
if j == 0:
tmp=rotor[k]
break
else:
tmp=rotor[k][j:] + rotor[k][0:j]
break
tmp_list.append(tmp)
# print(tmp_list)
message_list = []
for i in range(0, len(tmp_list[i])):
tmp = ""
for j in range(0, len(tmp_list)):
tmp += tmp_list[j][i]
message_list.append(tmp)
print(message_list)
def spread_list(lst):
for item in lst:
if isinstance(item,(list,tuple)):
yield from spread_list(item)
else:
```

```
yield item

pass

if __name__ == '__main__':

for i in spread_list(message_list):

print("*"*25)

print(i) #在多个输出中查找有语义的字符串即为flag内容
```

键盘密码：

键盘密码应该不算是一种加密算法，但是一种有趣的设置密码方式。他就是a-z(A-Z)对应成键盘上的字母，把键盘字母一行一行的对应即可。包围的键就是要找的值。

每个围起来的圈之间通常会有明显的间隔，比如空格。如：r5yG lp9l BjM tFhB T6uh y7iJ QsZ bhM

poem codes诗歌加密：

(内容地址出处)https://blog.csdn.net/weixin_45530599/article/details/108027293：

① 给出一首诗歌

for my purpose holds to sail beyond the sunset, and the baths of all the western stars until I die.

② 给出5个关键单词。

“for”, “sail”, “all”, “stars”, “die.”

对其进行拆散：

f o r s a i l a l l s t a r s d i e

接下来按照 字母表顺序 进行编号，若遇相同字母，则继续 +1

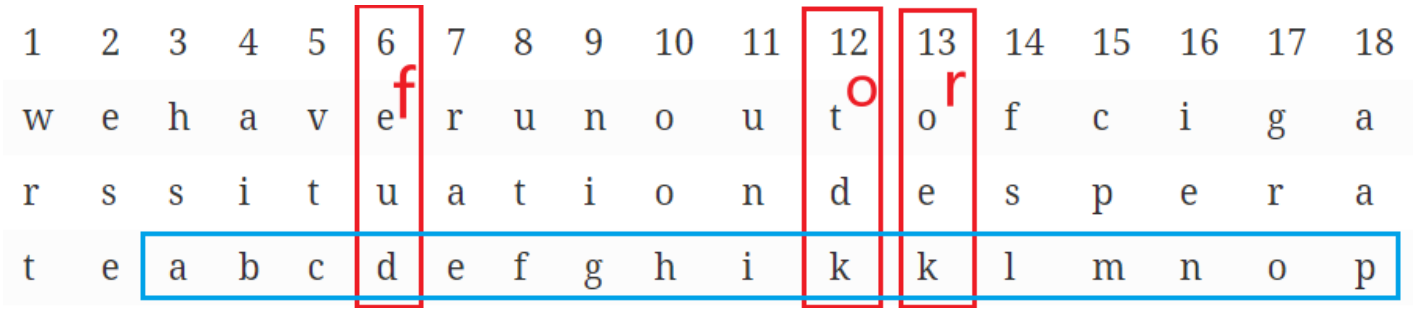
f	o	r	s	a	i	l	a
6	12	13	15	1	7	9	2
l	l	s	t	a	r	s	d
10	11	16	18	3	14	17	4
i	e						
8	5						

③ 将要传递的消息进行加密。

We have run out of cigars, situation desperate。

先对其进行编码。因为给出的5个关键词，其长度为18.所以以18为一组。

若一组长度不满18，则用abc(不要求有序)进行补充。



将排好的消息，按照之前给出的诗歌字母编号写下密文。

for my purpose holds to sail beyond the sunset, and the baths of all the western stars until I die.

如， for --> eud tdk oek 那么得到的又可以按照5个（适当个数）为一组进行重新分组，得到最后密文。

我的看法：

其实排序逻辑挺常规的，就是诗歌 --> 关键词，原文 --> 参照顺序排列，密文 --> 按诗歌关键词对原文映射取值。

解题脚本：

用github的脚本：(单独复制poemcode.py是会报错的，因为文件中有其他依靠)：

git clone git://github.com/abpolym/crypto-tools

用法：(python2，ctfpoem是诗歌，ctfcip是加密密文)

python2 poemcode.py examples/2/ctfpoem examples/2/ctfcip

URL编码规则：

URL 编码使用 "%" 其后跟随两位的十六进制数来替换非 ASCII 字符。

ASCII Value	URL-encode	ASCII Value	URL-encode	ASCII Value	URL-encode	ASCII Value	URL-encode	ASCII Value	URL-encode	ASCII Value	URL-encode
NULL结束符	%00	0	%30	`	%60		%90	À	%c0	ð	%f0
	%01	1	%31	a	%61	‘	%91	Á	%c1	ñ	%f1
	%02	2	%32	b	%62	’	%92	Â	%c2	ò	%f2
	%03	3	%33	c	%63	“	%93	Ã	%c3	ó	%f3
	%04	4	%34	d	%64	”	%94	Ä	%c4	ô	%f4
	%05	5	%35	e	%65	•	%95	Å	%c5	õ	%f5
	%06	6	%36	f	%66	–	%96	Æ	%c6	ö	%f6
	%07	7	%37	g	%67	—	%97	Ç	%c7	÷	%f7
backspace	%08	8	%38	h	%68	˜	%98	È	%c8	ø	%f8
tab	%09	9	%39	i	%69	™	%99	É	%c9	ù	%f9

换行符	%0a	:	%3a	j	%6a	š	%9a	Ê	%ca	ú	%fa
	%0b	;	%3b	k	%6b	›	%9b	Ë	%cb	û	%fb
	%0c	<	%3c	l	%6c	œ	%9c	Ì	%cc	ü	%fc
c return	%0d	=	%3d	m	%6d		%9d	Í	%cd	ý	%fd
	%0e	>	%3e	n	%6e	ž	%9e	Î	%ce	þ	%fe
	%0f	?	%3f	o	%6f	Ÿ	%9f	Ï	%cf	ÿ	%ff
	%10	@	%40	p	%70		%a0	Đ	%d0		
	%11	A	%41	q	%71	¡	%a1	Ñ	%d1		
	%12	B	%42	r	%72	¢	%a2	Ò	%d2		
	%13	C	%43	s	%73	£	%a3	Ó	%d3		
	%14	D	%44	t	%74		%a4	Ô	%d4		
	%15	E	%45	u	%75	¥	%a5	Õ	%d5		
	%16	F	%46	v	%76	|	%a6	Ö	%d6		
	%17	G	%47	w	%77	§	%a7		%d7		
	%18	H	%48	x	%78	¨	%a8	Ø	%d8		
	%19	I	%49	y	%79	©	%a9	Ù	%d9		
	%1a	J	%4a	z	%7a	ª	%aa	Ú	%da		
	%1b	K	%4b	{	%7b	«	%ab	Û	%db		
	%1c	L	%4c	|	%7c	¬	%ac	Ü	%dc		
	%1d	M	%4d	}	%7d	¯	%ad	Ý	%dd		
	%1e	N	%4e	~	%7e	®	%ae	Þ	%de		
	%1f	O	%4f		%7f	¯	%af	ß	%df		
空格	%20	P	%50	€	%80	°	%b0	à	%e0		
!	%21	Q	%51		%81	±	%b1	á	%e1		
"	%22	R	%52	‚	%82	²	%b2	â	%e2		
#	%23	S	%53	*ƒ*	%83	³	%b3	ã	%e3		
$	%24	T	%54	„	%84	´	%b4	ä	%e4		
%	%25	U	%55	…	%85	µ	%b5	å	%e5		
&	%26	V	%56	†	%86	¶	%b6	æ	%e6		
'	%27	W	%57	‡	%87	·	%b7	ç	%e7		

(	%28	X	%58	ˆ	%88	¸	%b8	è	%e8		
)	%29	Y	%59	‰	%89	¹	%b9	é	%e9		
*	%2a	Z	%5a	Š	%8a	º	%ba	ê	%ea		
+	%2b	[	%5b	‹	%8b	»	%bb	ë	%eb		
,	%2c	\	%5c	Œ	%8c	¼	%bc	ì	%ec		
-	%2d	]	%5d		%8d	½	%bd	í	%ed		
.	%2e	^	%5e	Ž	%8e	¾	%be	î	%ee		
/	%2f	_	%5f		%8f	¿	%bf	ï	%ef		

仿射密码：

仿射密码是一种替换密码，它是一个字母对一个字母的。为单表加密的一种，字母系统中所有字母都藉一简单数学方程加密，对应至数值，或转回字母。 其仍有所有替代密码之弱处。所有字母皆借由方程加密，b为移动大小。

在仿射加密中，大小为m之字母系统首先对应至0..m-1范围内之数值， 接着使用模数算数来将原文件中之字母转换为对应加密文件中的数字。 （其实这有点像我的base64表单下标替换）

它的加密函数是：（其中a和m互质，m是字母的数目）

$$e(x) = ax + b \pmod{m}$$

解密函数是：

$$d(x) = a^{-1}(x - b) \pmod{m}$$

补充求集合Z上数x的逆元的：

gmpy2.invert(x,Z) 或 libnum.invmode(e,(p-1)*(q-1))

a之乘法逆元素仅存在于a与m互质条件下。 由此，没有a的限制，可能无法解密。 易知解密方程逆于加密方程：

$$\mathrm{D}(\mathrm{E}(x)) = a^{-1}(\mathrm{E}(x) - b) \mod m = a^{-1}(((ax + b) \mod m) - b) \mod m = a^{-1}(ax + b - b) \mod m = a^{-1}ax \mod m = x \mod m.$$

非传统密码类型解析

CTF中的LFSR考点(一)：

前提概要：

这是我在理解了作者：道路结冰的博客深入分析CTF中的LFSR类题目（一）下写的一次回顾和分析，只是在其中加上自己的见识和理解来加深印象。

博客地址： 深入分析CTF中的LFSR类题目（一） - 安全客，安全资讯平台

前言:

LFSR（线性反馈移位寄存器）已经成为如今CTF中密码学方向题目的一个常见考点了，在今年上半年的一些国内赛和国际赛上，也出现了非常多的这类题目，但是其中绝大多数题目目前都没有writeups（或者writeups并没有做cryptanalysis，而是通过爆破的方法解决，这种思路只适用于部分类似去年强网杯出现的几道非常基础的LFSR类题目有效，对于绝大多数国际赛上的题目不仅是没有任何效果的，也是没有任何意义的，只有真正掌握了LFSR的密码学原理，才有可能在国际赛上解决一道高分值的LFSR类题目），网上针对这类考点的详细分析也不多，因此接下来我将通过几篇文章，对这类知识点进行一个详细的分析。

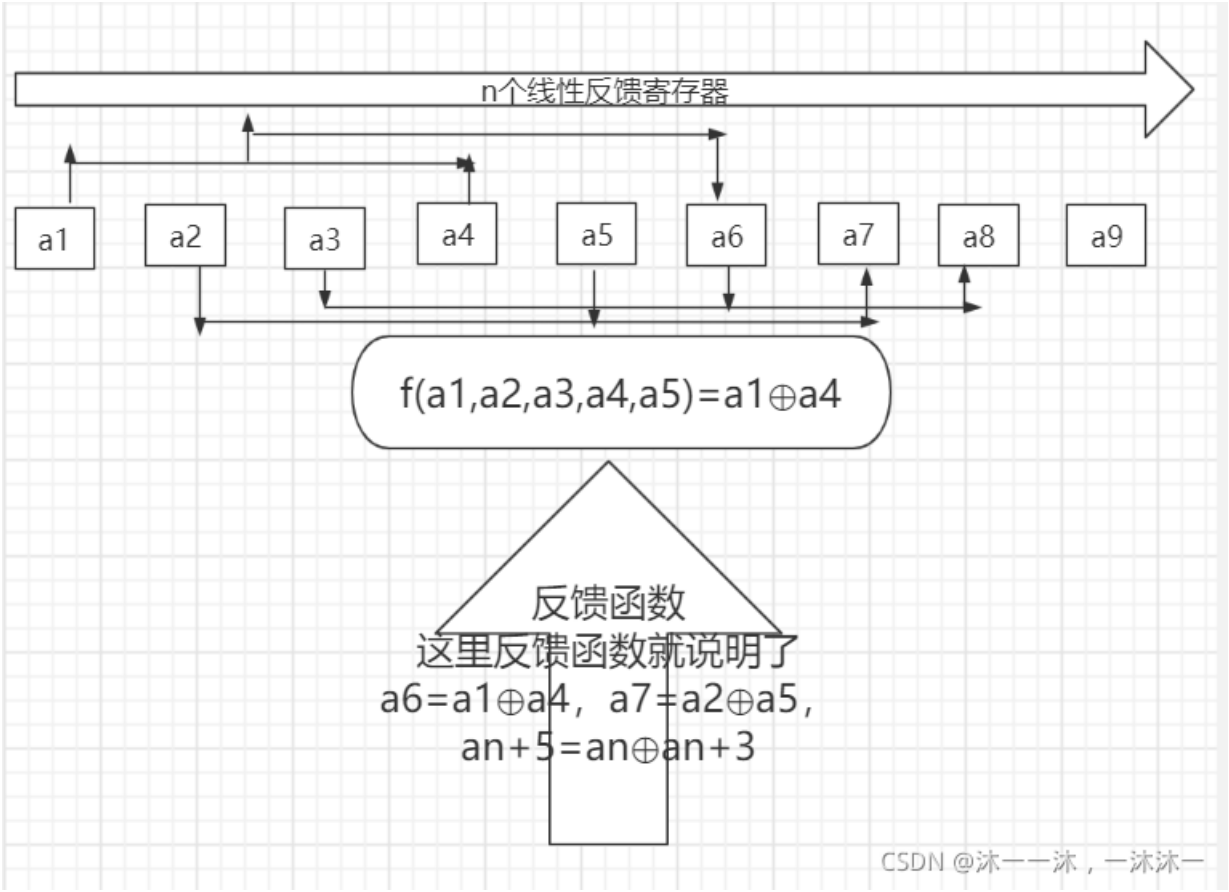
LFSR简介：

LFSR简介：

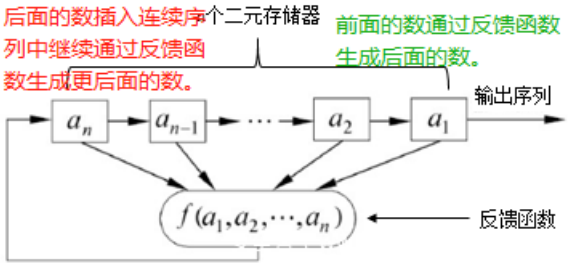
LFSR是属于FSR（反馈移位寄存器）的一种，除了LFSR之外，还包括NFSR（非线性反馈移位寄存器）。

附加：

反馈移位就是可以通过前面已经存在的寄存器中的值反馈出后面的的寄存器的值，通过不断移位对应不同的前寄存器值一直反馈出后面连续的后寄存器值。



FSR是流密码产生密钥流的一个重要组成部分，在GF(2)上的一个n级FSR通常由n个二元存储器和一个反馈函数组成，如下图所示：



如果这里的反馈函数是线性的，我们则将其称为LFSR，此时该反馈函数可以表示为：

（其中cn=0或1，⊕表示异或（模二加），也就是说反馈函数必然为a1……⊕……an形式中的一部分）

$$f(a_1, a_2, \cdots, a_n) = c_n a_1 \oplus c_{n-1} a_2 \oplus \cdots c_1 a_n$$

我们接下来通过一个例子来更直观的明确LFSR的概念：

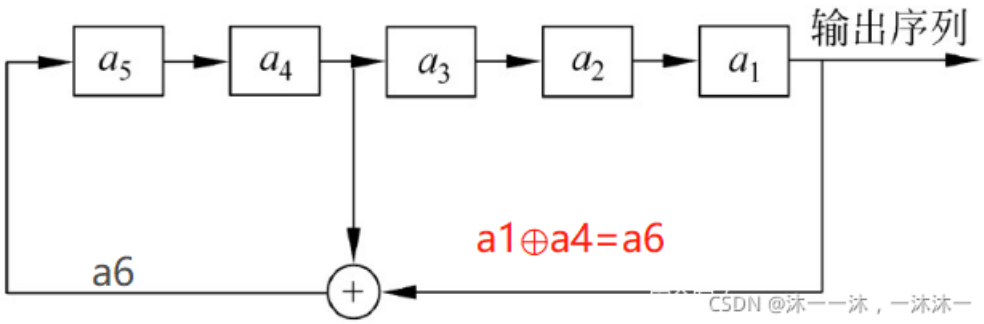
假设给定一个5级的LFSR，其初始状态（即a1到a5这5个二元存储器的值）为：

$$(a_1, a_2, a_3, a_4, a_5) = (1,0,0,1,1)$$

其反馈函数为：

$$f(a_1, a_2, a_3, a_4, a_5) = a_4 \oplus a_1$$

整个过程可以表示为下图所示的形式：



接下来我们来计算该LFSR的输出序列，输出序列的前5位即为我们的初始状态10011，第6位的计算过程如下：

$$a_6 = a_4 \oplus a_1 = 1 \oplus 1 = 0$$

第7位的计算过程如下:

$$a_7 = a_5 \oplus a_2 = 1 \oplus 0 = 1$$

由此类推，可以得到前31位的计算结果如下：

1001101001000010101110110001111

对于一个 n 级的LFSR来讲，其最大周期为2^n-1 ，因此对于我们上面的 5 级 LFSR 来讲，其最大周期为 2^5-1=31，再后面的输出序列即为前 31 位的循环。

注意：

这里的级就是能通过反馈函数生成完整连续序列的最少寄存器数，因为a6=a1⊕a4，所以a5必须包含在内，所以是5级寄存器。

通过上面的例子我们可以看到，对于一个LFSR来讲，我们目前主要关心三个部分：

初始状态、反馈函数和输出序列。

CTF的LFSR题目示例：

那么对于CTF中考察LFSR的题目来讲也是如此，大多数情况下，我们在CTF中的考察方式都可以概括为：

给出反馈函数和输出序列，要求我们反推出初始状态，初始状态即为我们需要提交的flag，另外大多数情况下，初始状态的长度我们也是已知的。

显然，这个反推并不是一个容易的过程，尤其当反馈函数十分复杂的时候，接下来我们就通过一些比赛当中出现过的具体的CTF题目，来看一下在比赛当中我们应该如何解决这类问题，由于不同题目之间难度差异会很大，所以我们先从最简单的题目开始，我将尽可能的用最通俗的语言和脚本来进行演示，在后面会逐渐提升题目的难度，同时补充相应的代数知识。

2018 CISCN 线上赛 oldstreamgame

题目给出的脚本如下：

```
flag = "flag{xxxxxxxxxxxxxxx}"

assert flag.startswith("flag{")

assert flag.endswith("}")

assert len(flag)==14

def lfsr(R,mask):

output = (R << 1) & 0xffffffff

i=(R&mask)&0xffffffff

lastbit=0

while i!=0:

lastbit^=(i&1)

i=i>>1

output^=lastbit

return (output,lastbit)

R=int(flag[5:-1],16)

mask = 0b10100100000010000000100010010100

f=open("key","w")

for i in range(100):

tmp=0

for j in range(8):

(R,out)=lfsr(R,mask)

tmp=(tmp << 1)^out

f.write(chr(tmp))

f.close()
```

分析一下我们的已知条件（1）：

已知初始状态的长度为4个十六进制数，即32位，初始状态的值即我们要去求的flag，所以初步判断这里的级是32，那么最大周期就是2^32-1。

已知反馈函数lfsr，只不过这里的反馈函数是代码的形式，我们需要提取出它的数学表达式。 已知输出序列。

```
assert flag.startswith("flag{")
assert flag.endswith("}")
assert len(flag)==14


def lfsr(R,mask):
    output = (R << 1) & 0xffffffff
    i=(R&mask)&0xffffffff
    lastbit=0
    while i!=0:
        lastbit^=(i&1)
        i=i>>1
    output^=lastbit
    return (output,lastbit)


R=int(flag[5:-1],16)
mask = 0b10100100000010000000100010010100
```

(14-6)*4=32位

这里说明原本是16进制，所以是4位

那么我们的任务很明确，就是通过分析lfsr函数，整理成数学表达式的形式求解即可，接下来我们一行一行的来分析这个函数：

接收两个参数，R是32位的初始状态(即flag)，mask是32位的掩码，由于mask已知，所以我们就直接把他当做一个常数即可，它只在推导反馈函数的时候起作用。

把R左移一位后取低32位（即抹去R的最高位，然后在R的最低位补0）的值赋给output变量，最低位的0要在后面补上反馈函数lastbit计算的值来造成循环。

```
def lfsr(R,mask):
    output = (R << 1) & 0xffffffff
    i=(R&mask)&0xffffffff
    lastbit=0
    while i!=0:
        lastbit^=(i&1)
        i=i>>1
    output^=lastbit
    return (output,lastbit)
```

把传入的R和mask做按位与运算，运算结果取低32位，将该值赋给i变量，注意这里和上面的output变量是独立分开的。

从i的最低位向i的最高位依次做异或运算，将运算结果赋给lastbit变量，最后再补到output上去。

将output变量的最后一位设置成lastbit变量的值，对应前面造成循环。

返回output变量和lastbit变量的值，output即经过一轮lfsr之后的新序列，就是左移且补了一位反馈函数生成的lastbit后的值。lastbit即经过一轮lfsr之后输出的一位。

把反馈函数从代码形式整理成反馈函数表达式：

通过上面的分析，我们可以看出在这道题的情境下，lfsr函数本质上就是一个输入R输出lastbit的函数，虽然我们现在已经清楚了R是如何经过一系列运算得到lastbit的，但是我们前面的反馈函数都是数学表达式的形式，我们能否将上述过程整理成一个表达式的形式呢？这就需要我们再进一步进行分析：

mask只有第3、5、8、12、20、27、30、32这几位为1，其余位均为0。

mask与R做按位与运算得到i，当且仅当R的第3、5、8、12、20、27、30、32这几位中也出现1时，i中才可能出现1，否则i中将全为0。

lastbit是由 i 的最低位向i的最高位依次做异或运算得到的，在这个过程中，所有为 0 的位我们可以忽略不计（因为0异或任何数等于任何数本身，不影响最后运算结果），因此lastbit的值仅取决于 i 中有多少个1：当i中有奇数个1时，lastbit等于1；当i中有偶数个1时，lastbit等于0。

当R的第3、5、8、12、20、27、30、32这几位依次异或结果为1时，即R中有奇数个1，因此将导致 i 中有奇数个1；当R的第3、5、8、12、20、27、30、32这几位依次异或结果为0时，即R中有偶数个1，因此将导致 i 中有偶数个1。

因此我们可以建立出联系：lastbit等于R的第3、5、8、12、20、27、30、32这几位依次异或的结果。

将其写成数学表示式的形式，即为我们要求的反馈函数，反馈函数的形式也说明了初始位数要32位：

$$lastbit = R_3 \oplus R_5 \oplus R_8 \oplus R_{12} \oplus R_{20} \oplus R_{27} \oplus R_{30} \oplus R_{32}$$

然后我们看一下明文key是怎么来的：

key=20FDEEF8A4C9F4083F331DA8238AE5ED083DF0CB0E7A83355696345DF44D7C186C1F459BCE135F

最后八行代码，对flag，它做了一百次循环，每次循环内都嵌套了8位的循环来产生一个结果字符，并将这个结果写到key里面去，所以key里面总共有一百个ASCII字符，共两百个16进制数。（4位为1个16进制数，1个字符byte是8位）

补充：

题目之所以会出现 100 个ASCII字符是因为 4 循环次加内嵌的 8 位一次共 32 位循环往后产生的 4 个flag生成的加密字符共 8 个 16 进制数后，继续用这 4 个加密后的 flag 字符继续新一轮加密，就是多层加密。所以我们取 32 位即可，结果 flag 也是 4 个ASCII字符拆分出的 8 个 16 进制数。

```
f=open("key","w")
for i in range(100):
    tmp=0
    for j in range(8):
        (R,out)=lfsr(R,mask)
        tmp=(tmp << 1)^out
    f.write(chr(tmp))
f.close()
```

文件内是100个ASCII字符，这是前面提到的给反馈函数和生成的值反推初始值。
100个字符每个都是用lastbit补成的8位数生成的ASCII码对应的字符。

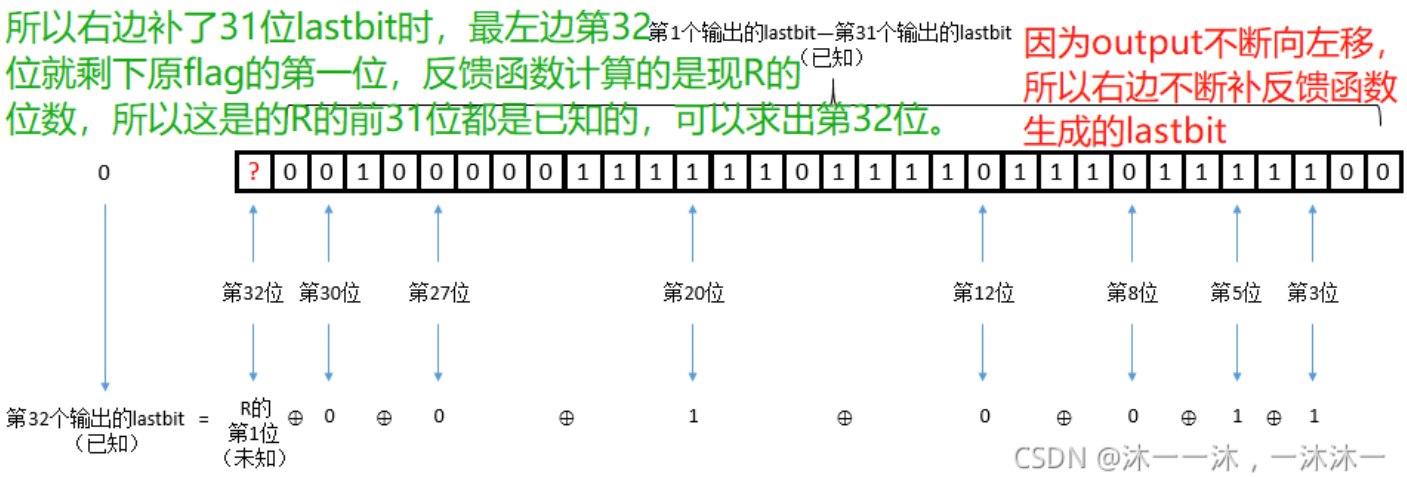
显然，lastbit和R之间满足线性关系，那么接下来我们就可以开始求解了：

而且从这里我们可以知道，这种移位的CTF题目类型要反推32位的初始值要多少位输出序列呢，答案就是32位，因为这种移位的题目是32位为一个循环，这就和我们前面说的2^32-1的循环不太同了，因为这里是移位操作。

32位Key值：00100000111111011111011011111000

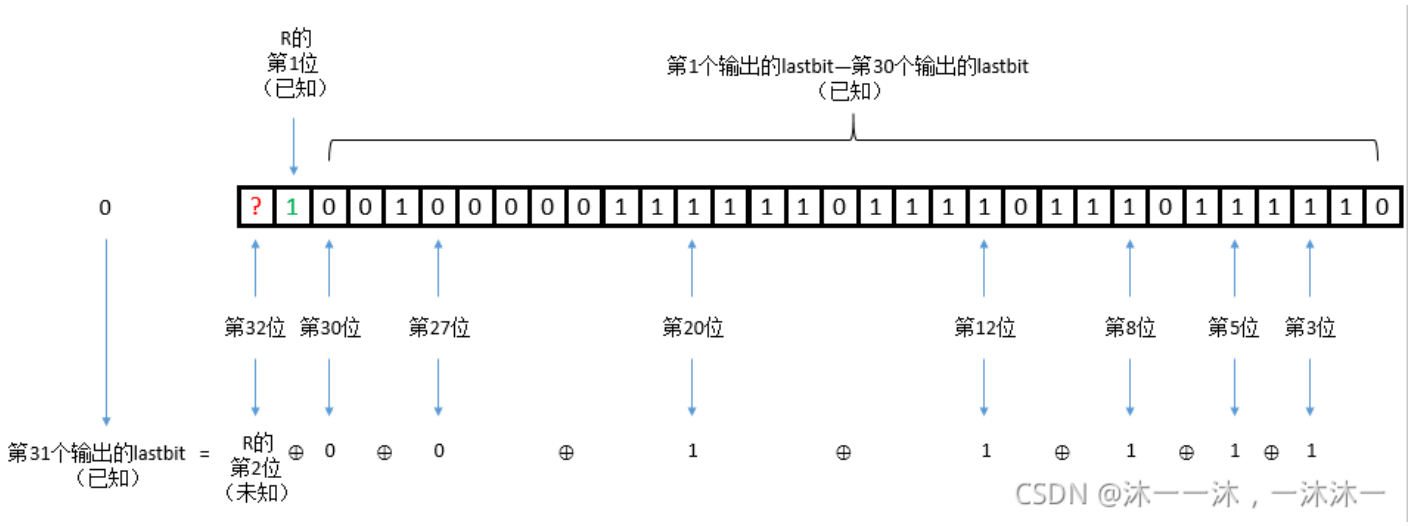
解题的关键是发现在明文只剩最后一位时,可以通过最后的结果来求出排在第32位的第一位，然后就可以反推回去了。

我们想象这样一个场景，当即将输出第32位lastbit时，此时R已经左移了31位，根据上面的数学表达式，我们有：



这样我们就可以求出R的第1位，同样的方法，我们可以求出R的第2位：

以此类推，R的全部32位我们都可以依次求出了。



最终脚本，这里注意小端顺序的反序取位：

key1="00100000111111011111011011111000" #lastbit全部值，就是反馈函数生成的值，32位的key1

key2=key1

flag=[]

for i in range(32):

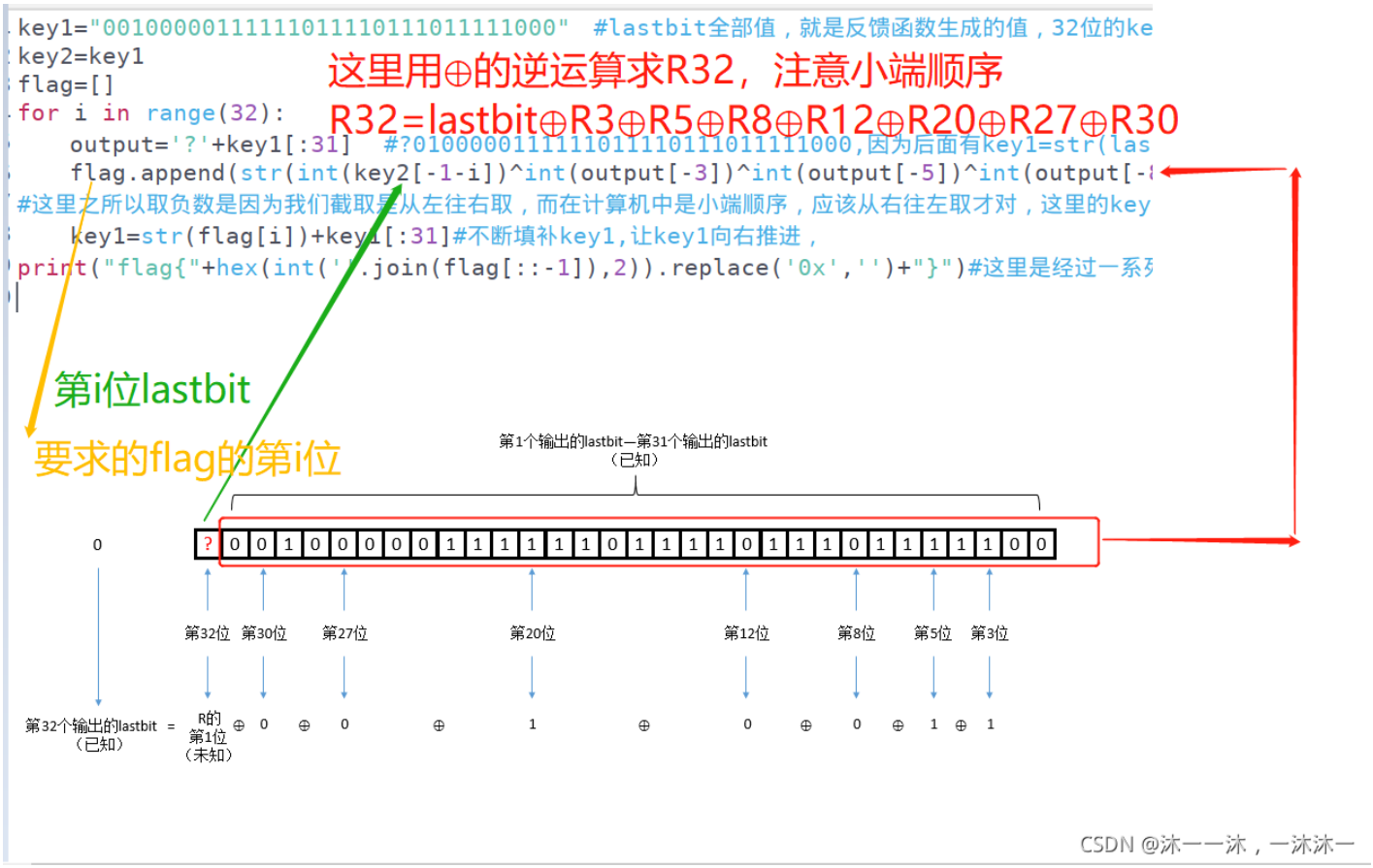
output='?'+key1[:31] #?00100000111111011111011011111000,因为后面有key1=str(lastbit)+key1[:31]，key1不断填补，output不断取前31位，所以这里output每次把?定在第i位上，注意output和key1是独立的分开的。

flag.append(str(int(key2[-1-i])^int(output[-3])^int(output[-5])^int(output[-8])^int(output[-12])^int(output[-20])^int(output[-27])^int(output[-30])))

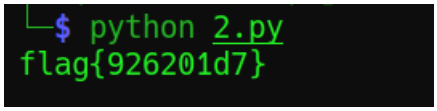
#这里之所以取负数是因为我们截取是从左往右取，而在计算机中是小端顺序，应该从右往左取才对，这里的key2[-1-i]就是小端左到右的第i位，也就是前面分析的反馈函数生成值的第i位。flag值的原第i位,现在在第32位，可以通过⊕的可逆性来求，就是R32=lastbit ⊕ R3 ⊕ R5 ⊕ R8 ⊕ R12 ⊕ R20 ⊕ R27 ⊕ R30

key1=str(flag[i])+key1[:31] #不断填补key1,让key1向右推进，

print("flag{"+hex(int(''.join(flag[::-1]),2)).replace('0x','')+"}") #这里是经过一系列操作，首先把flag变成小端顺序的[::-1]，然后就是转十六进制。



结果：



解密工具、脚本积累

Cipher工具：

Ciphey 是一个使用自然语言处理和人工智能的全自动解密/解码/破解工具，只需要输入加密文本，它就能给你返回解密文本。

Ciphey 基本使用：

文件输入：

ciphey -f encrypted.txt

或

python -m ciphey -f encrypted.txt

不规范的方法：

ciphey -- "Encrypted input"

或

python -m ciphey -- "Encrypted input"

正常方式：

ciphey -t "Encrypted input"

或

python -m ciphey -t "Encrypted input"

要去除进度条、概率表和所有噪音，请使用安静模式：

ciphey -t "encrypted text here" -q

Ciphey 支持解密的密文和编码多达51种，下面列出一些基本的选项基本密码：

Caesar Cipher

ROT47 (up to ROT94 with the ROT47 alphabet)

ASCII shift (up to ROT127 with the full ASCII alphabet)

Vigenère Cipher

Affine Cipher

Binary Substitution Cipher (XY-Cipher)

Baconian Cipher (both variants)

Soundex

Transposition Cipher

Pig Latin

现代密码学：

Repeating-key XOR

Single XOR

编码：

Base32

Base64

Z85 (release candidate stage)

Base65536 (release candidate stage)

ASCII

Reversed text

Morse Code

DNA codons (release candidate stage)

Atbash

Standard Galactic Alphabet (aka Minecraft Enchanting Language)

Leetspeak

Baudot ITA2

URL encoding

SMS Multi-tap

DMTF (release candidate stage)

UUencode

Braille (Grade 1)

Ciphey 的功能不仅于本文介绍的这些，本文所介绍的只是冰山一角，它还可以添加属于你自己的解码器：

https://github.com/Ciphey/Ciphey/wiki/Adding-your-own-ciphers

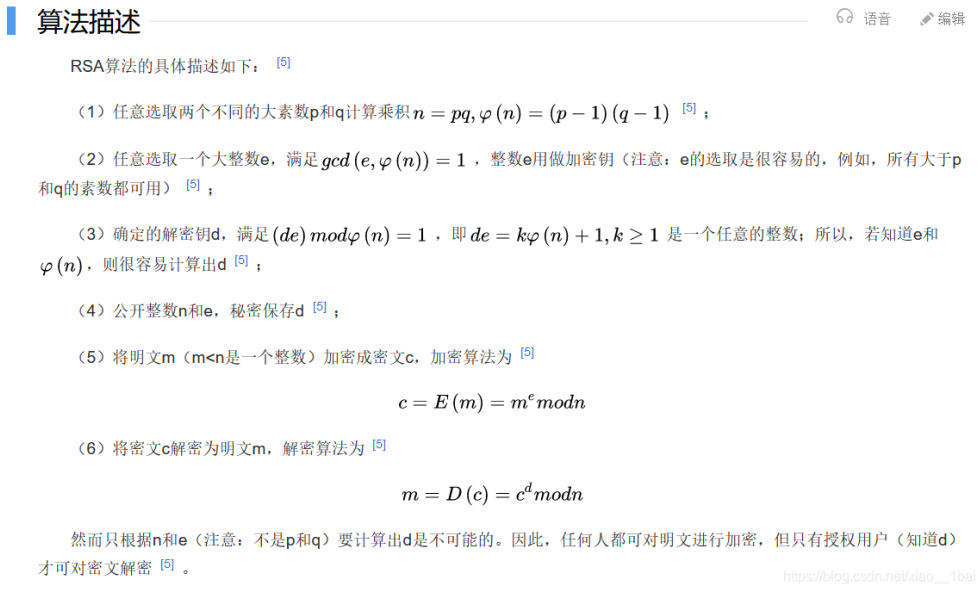
CTF-RSA-tool工具：

RSA前景知识：

RSA公开密钥密码体制是一种使用不同的加密密钥与解密密钥，“由已知加密密钥推导出解密密钥在计算上是不可行的”密码体制 。

在公开密钥密码体制中，加密密钥（即公开密钥）PK是公开信息，而解密密钥（即秘密密钥）SK是需要保密的。加密算法E和解密算法D也都是公开的。虽然解密密钥SK是由公开密钥PK决定的，但却不能根据PK计算出SK [2] 。

RSA公开密钥密码体制的原理是：根据数论，寻求两个大素数比较简单，而将它们的乘积进行因式分解却极其困难，因此可以将乘积公开作为加密密钥 [4] 。



算法描述

语音 编辑

RSA算法的具体描述如下： [5]

（1）任意选取两个不同的大素数p和q计算乘积 $n = pq, \varphi(n) = (p-1)(q-1)$ [5] ；

（2）任意选取一个大整数e，满足 $gcd(e, \varphi(n)) = 1$ ，整数e用做加密钥（注意：e的选取是很容易的，例如，所有大于p和q的素数都可用） [5] ；

（3）确定的解密钥d，满足 $(de) mod \varphi(n) = 1$ ，即 $de = k\varphi(n) + 1, k \geq 1$ 是一个任意的整数；所以，若知道e和 $\varphi(n)$ ，则很容易计算出d [5] ；

（4）公开整数n和e，秘密保存d [5] ；

（5）将明文m（m<n是一个整数）加密成密文c，加密算法为 [5]

$$c = E(m) = m^e mod n$$

（6）将密文c解密为明文m，解密算法为 [5]

$$m = D(c) = c^d mod n$$

然而只根据n和e（注意：不是p和q）要计算出d是不可能的。因此，任何人都可对明文进行加密，但只有授权用户（知道d）才可对密文解密 [5] 。

RSA脚本工具使用说明：

usage: solve.py [-h]

用法：solve.py[-h] (--decrypt DECRYPT | -c DECRYPT_INT | --private | -i INPUT | -g)

[--createpub] [-o OUTPUT] [--dumpkey] [--enc2dec ENC2DEC] [-k KEY] [-N N] [-e E] [-d D] [-p P] [-q Q] [--KHBFA KHBFA][--pbits PBITS]

[-v]

It helps CTFer to get first blood of RSA-base CTF problems 它有助于CTFer获得RSA基础CTF问题的第一滴血

-v, --verbose print details 详细的打印细节

optional arguments:可选参数(注意！！！这里之间只可选一个，且必选一个！)：

-h, --help show this help message and exit --帮助显示此帮助消息并退出

--decrypt DECRYPT decrypt a file, usually like "flag.enc" 解密文件，通常类似于“flag.enc”

(通常搭配k的.pem或.pub一起使用)

-c DECRYPT_INT,

--decrypt_int DECRYPT_INT 解密长整形数

--private Print private key if recovered 打印私钥（如果已解密）

-i INPUT input a file with all necessary parameters (see examples/input_example.txt)

输入包含所有必要参数的文件（请参见示例/输入（示例.txt）

-g, --gadget Use some gadgets to pre-process your data first 使用一些小工具先预处理数据

some gadgets:一些小工具预处理数据(全部可选)：

--createpub Take N and e and output to file specified by "-o" or just print it

获取N和e并输出到由“-o”指定的文件或者直接打印出来就行了

-o OUTPUT, --output OUTPUT 输出 Specify the output file path in --createpub mode.

在--createpub模式下指定输出文件路径。

--dumpkey Just print the RSA variables from a key - n,e,d,p,q

只打印一个key-n、e、d、p、q中的RSA变量

--enc2dec ENC2DEC get cipher (in decimalism) from a encrypted file

从加密文件中获取密码（十进制）

the RSA variables:RSA变量：Specify the variables whatever you got指定您得到的变量：(全部可选，实验中发现输入时只能用N、e参数进行命令行输入)

-k KEY, pem file, usually like ".pub" or ".pem", and it begins with "-----BEGIN"

pem文件，通常类似于“.pub”或“.pem”，并以“-----BEGIN”开始

-N N the modulus 模量

-e E the public exponent 公共指数

-d D the private exponent 私人指数

-p P one factor of modulus 模量的一个因子

-q Q one factor of modulus 模量的一个因子

extra variables:额外变量：Used in some special methods 在一些特殊的方法中使用：

--KHBFA KHBFA use Known High Bits Factor Attack, this specify the High Bits of factor

使用已知的高位因子攻击，这指定因子高位

--pbits PBITS customize the bits lenth of factor, default is half of n`s bits lenth

自定义因子的位长度，默认值为n's比特长度

多组n,e,c在解题时长这个样子：

```
N is 203872343041197070988331406754084460184035797431363253379911752970643927197089590754176729406403532638725563324515843987123855955261892854132
e is 46957
d is 10025376989936072505039846057794501927528527372889258010084848452510038807649542645621941049733296360842498419248238502260955678307712093819
e is 56167
c is 91174026432228072347362717897275685291913109679763305989422324622813788291728363437794254475222850079514104811831817481943576914980040450543
```

```
n = 208233691145562607629135888444718697257629858122159879938677
n = 190838216137364299584320249800744053754089532692768396963192
e = 65537
c = 132340339973043167780377237555402951765664171675831253347481
```

```
n = 49717352238903813258167965634872644122363546007764600124211711250852904300466263993456901629465
e = 13720370251305502198453722303188629715020031854527043028194234587721186750392338568840741980913
c = 33995928221963087827338803441589901571528069870645123081701004401546919289265619397264329787933

n = 49717352238903813258167965634872644122363546007764600124211711250852904300466263993456901629465
e = 13521927979417175825463063347222803267672338126236919097685639546419640649137631879086526055040
c = 20663727752331650736213993392793715555002283733124450484396596147661401938360252713595785342791
```



```
c : 68679409966918700315304117144859068445528181933255289063193054014288151083466807594332167633810967321824633144
e : 7
n : 24810910852704603048663349011054669655631146433543459534796438815331335687309113943583212235150241971378068933

c : 11179052201843296851154916696601291938235676417475847173247453892471333698920515860207084422135916963942522752
e : 7
n : 47127839105299361033791208737798899776781255381503030381686909082155757361019104103280620540716894699133142173

c : 40118076943735337559537379692982070943921515348000211097599356263330760075906748374129727526740438883695094503
e : 7
n : 43134291711046821358455351358884087777021003839470296505990450581706219379356272391794220129036895199873385802

c : 12649076592222649371192164869044025408231371717627780046219346377852024544337050152652676577122342534868958091
e : 7
n : 19300838921149221007298944887478599082800229045219271606272038103970656559943914197281654158587468730541828306

c : 28899089935435267588235897519846120393433214114341521238696384122507316899457327055029546972333281452563984838
e : 7
n : 30754121488827635692971849599267749375077949182550303145729325375314926401905783830931628738658879320179944880

c : 14086629413855672403639830676118042465846020320143823318815048070368505684208141652603596234960968703217788472
e : 7
n : 30430477983470426195631142659668071772256641205525929891985872996115858010744648779370983539942187689192406517

c : 20049299207588955907155276095787913402589652379134151403340360498371893119855957833576443856534037886298731701
e : 7
n : 35489275126536805974281635942907480463916089663069129771420[redacted]
```

可以看到特征真的就是多个n，e，c，甚至还有d，且不管这里的n,e,c是大还是小，长还是短，都列入多组n,e,c类型里。

一组n,e,c的题目样式：

```
N : 46065781388428960989637205658554417248531811702624626389974432923749270182062721955600
e : 35461110244130757205657218182792589919834535022875373093108939327546391654445662689424
```

```
N is 966808932627497190635859236054960349099463975227350564265384
e is 65537
c is 168502910088858295634315070244377409556567637139736308082186
```

```
hbop = 0xf3a5f928e11c5901f9f4289e513f046748efb99d4f8e706e207a943e1d2c9
n = 0x7e7007c7c85788b9b77cda64c9b3f5d2a795fe1b1f8d3f120288a30a168c3ea9
e = 65537
```

从这里可以看到一组,n,e,c里面甚至可以没有c，这里的n,e,c也不管大小，长短，这里最后一个hbop的解题要用到前面说得sagemath，这里暂且不说。

不同情景下工具运行示例：

补充：

单个n,e,c,q,p,的时候最好用单个参数输入的方式，不要用文本读取的方式，因为文本读取的时候DEBUG显示的十六进制的d有时并不是我们想要的

只需要一组密钥的

wiener_attack

python2 solve.py --verbose -i examples/wiener_attack.txt

```
N : 46065781388428960989637205658554417248531811702624626389974432923749270182062721955
e : 35461110244130757205657218182792589919834535022875373093108939327546391654445662268
```

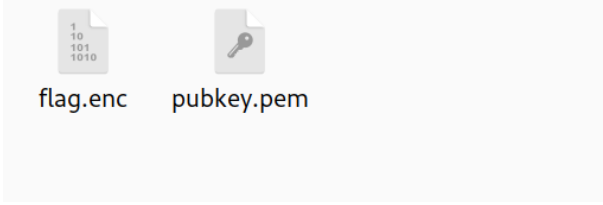
或者通过命令行，只要指定对应参数就行了

python2 solve.py --verbose --private -N
46065781388428960989637205658554417248531811702624626389974432923749270182062721955600077
-e
35461110244130757205657218182792589919834535022875373093108939327546391654445662689424454

```
N : 46065781388428960989637205658554417248531811702624626389974432923749270182062721955
e : 35461110244130757205657218182792589919834535022875373093108939327546391654445662268
```

factordb.com

python2 solve.py --verbose -k examples/jarvis_oj_mediumRSA/pubkey.pem --decrypt examples/jarvis_oj_mediumRSA/flag.enc




```
-----BEGIN PUBLIC KEY-----
MDowDQYJKoZIhvcNAQEBBQADKQAwJgIhAMJjauXD2OQ/+5erCQKPGqxsC/bNPXDr
yigb/+l/vjDdAgEC
-----END PUBLIC KEY-----

```

默认 (UTF-8, 部分) 其他： ISO-8859-1

n>·ß#îáÓ¾ºx e½=ImÚdAy

Boneh and Durfee attack

TODO: get an example public key solvable by boneh_durfee but not wiener

small q attack

python2 solve.py --verbose --private -k examples/small_q.pub

```
-----BEGIN PUBLIC KEY-----
MIGhMA0GCSqGSIb3DQEBAQUAA4GPADCBiwKBgwC60gz5ftUELfaWzk3z5aZ4z0+z
aT098S3+n9P9jMiquLlVM+QU4/wMN39O5UgnEYsdMFYaPHQb6nx2iZeJtRdD4HYJ
LfnrBdyX6xUFzp6xK1q54Qq/VvkgpY5+AOzwWXfocoNN2FhM9KyHy33FAVm9lix1
y++2xqw6MadOfY8eTBDVAgMBAAE=
-----END PUBLIC KEY-----
```


2017强网杯线上赛 RSA 费马分解（p&q相近时）

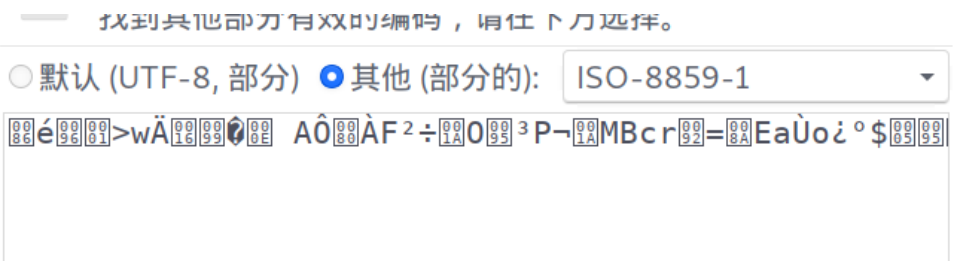
python2 solve.py --verbose -i examples/closed_p_q.txt

```
N is 966808932627497190635859236054960349099463975227350564
e is 65537
c is 168502910088858295634315070244377409556567637139736308
```

Common factor between ciphertext and modulus attack

python2 solve.py --verbose -k examples/common_factor.pub --decrypt examples/common_factor.cipher

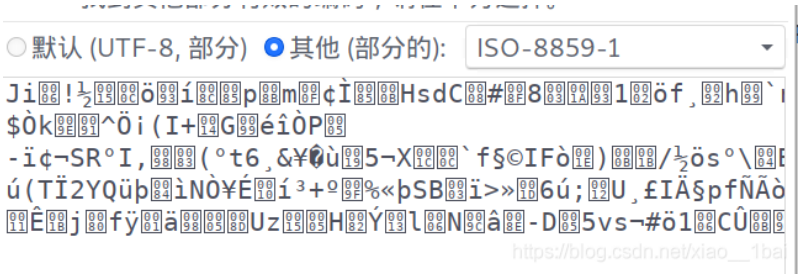
```
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCr5jPOwufsEKhRknkFplffThBB
YCPAw0/GTWS9i4JXt78get0EewrfIcUlsFIGjHApXHRsOxvhQ2857Yv3qBPkuEXO
DKicqCi0V2PUaxiYx6L6X4/nhCjKts33Dvhx25cbMjKEGhziRZzmUKFUNi+Az7ZB
Y8PKY61yvPvb8BVP9wIDAQAB
-----END PUBLIC KEY-----
```



small e

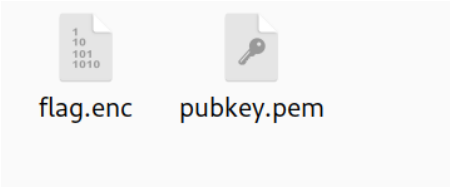
python2 solve.py --verbose -k examples/small_exponent.pub --decrypt examples/small_exponent.cipher

```
-----BEGIN PUBLIC KEY-----
MIICIDANBgkqhkiG9w0BAQEFAAOCAg0AMIICCAKCAgEAsKHzkKzT1DtH058TJmL2
nBWJJdkoceR4aeKEGpF8INUQJDG5qXgUWNhA/SlXeBWkFhLWh6NIfSb7riVvFdR0
DDRZG2RqvMyxonrN4pmz5xYAhXtFXCg2YOBFXGj/RcBkTP7CEdf1GhbILpHXhtks
eZ+zy0j5LeNCunDdghMFazFKjVHalJPPG4bsFf3wPgRudtPxoa0Kq7aEzl0VfjmY
KKY6WvWSAii7XqHma4/qo8y7r/VV40Z5dzDd/BxM9KndQGWIYpNIxMKSZd+eLD0C
VYvl41yyd/TnrHtRWO85A6OWSGNxAp5Uo0UpKrpHSZ8cJn5oCuc4GV/VryqAdZOY
kPXWnms+lOPlYIYapsbGnagkBduiGC5m7P9qipzfWtUibwc+fVJeBQ/dd+C7GJGk
nv7C02emk9KmeZ0NRmeVPU893sFqwVu0z2Al6ljstt+lcjFtoI0xBgc5czIq51l0
RvL9MEPfbh1gTGofDllHPZvBgtTsb8RYjxxrKqR2h2qEstTg1FkQOZEY1+HiDcwn
cD8r0+mvci83p2c7FdZ0kihiyE0A/C/H3d3JFcRpP8sLF4np3L1yrARlnnwY3PNi
VHYAQEAr/O8RuKPvnIvduqqNFMbo9RinCwNtIGuAnNmztRoewBMtrOltypRR80w4
q4TtR159lPvp/8AH8tFIYL0CAQM=
-----END PUBLIC KEY-----
```

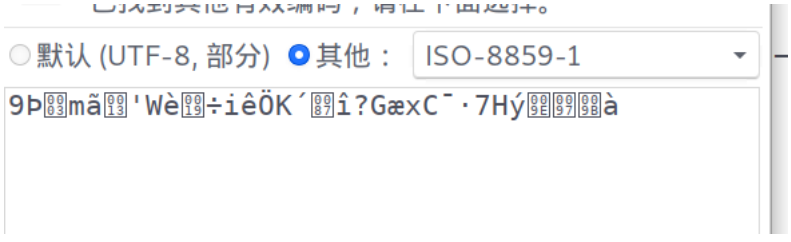



rabin method when e == 2

python2 solve.py --verbose -k examples/jarvis_oj_hardRSA/pubkey.pem --decrypt examples/jarvis_oj_hardRSA/flag.enc




```
-----BEGIN PUBLIC KEY-----
MDowDQYJKoZIhvcNAQEBBQADKQAwJgIhAMJjauXD2OQ/+5erCQKPGqxsC/bNPXDr
yigb/+l/vjDdAgEC
-----END PUBLIC KEY-----
```



Small fractions method when p/q is close to a small fraction

python2 solve.py --verbose -k examples/smallfraction.pub --private

```
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQEoAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAnjpIotEijRoXZUwKirN3QAAAAAJMWAAA
AAAAAAAAAAAAAAAAAAAAAAAABUlMDN0AghvxOWxg4490Zh+eqRW4+BCSOZMEEkM
NS9XgPPYCmAAAADsSQIDAQAB
-----END PUBLIC KEY-----
```


Known High Bits Factor Attack

python2 solve.py --verbose -i examples/KnownHighBitsFactorAttack.txt

```
hbop = 0xf3a5f928e11c5901f9f4289e513f046748efb99d4f8e706e207a943e1d2c9df43feab38e20c2106d87167e5501ac41adfc4
n = 0x7e7007c7c85788b9b77cda64c9b3f5d2a795fe1b1f8d3f120288a30a168c3ea932c7574700ff0f596c5ad04a703756aedc66b9
e = 65537
```

需要多组密钥的

第三届上海市大学生网络安全大赛--rrrsa d泄漏攻击

python2 solve.py --verbose -i examples/d_leak.txt

```
N is 203872343041197070988331406754084460184035797431363253379911752970643927197089590754
e is 46957
d is 100253769899360725050398460577945019275285273728892580100848484525100388076495426456
e is 56167
c is 91174026432228072347362717897275685291913109679763305989422324622813788291728363437
```


模不互素

（模不互素是给你两个单独分解不了的大数级n，方便让你求公因子。然后，只用一个n来加密，对，只用一个n。）

python2 solve.py --verbose -i examples/share_factor.txt

```
n = 208233691145562607629135888444718697257629858122159879938677836300514202410579123850554827880163279784683180670782338
n = 190838216137364299584320249800744053754089532692768396963192655968554261892568656506514604600798193689235761097230795
e = 65537
c = 132340339973043167780377237555402951765664171675831253347481153138562724618734859751767696399005566727346172733590195
```

共模攻击

python2 solve.py --verbose -i examples/share_N.txt

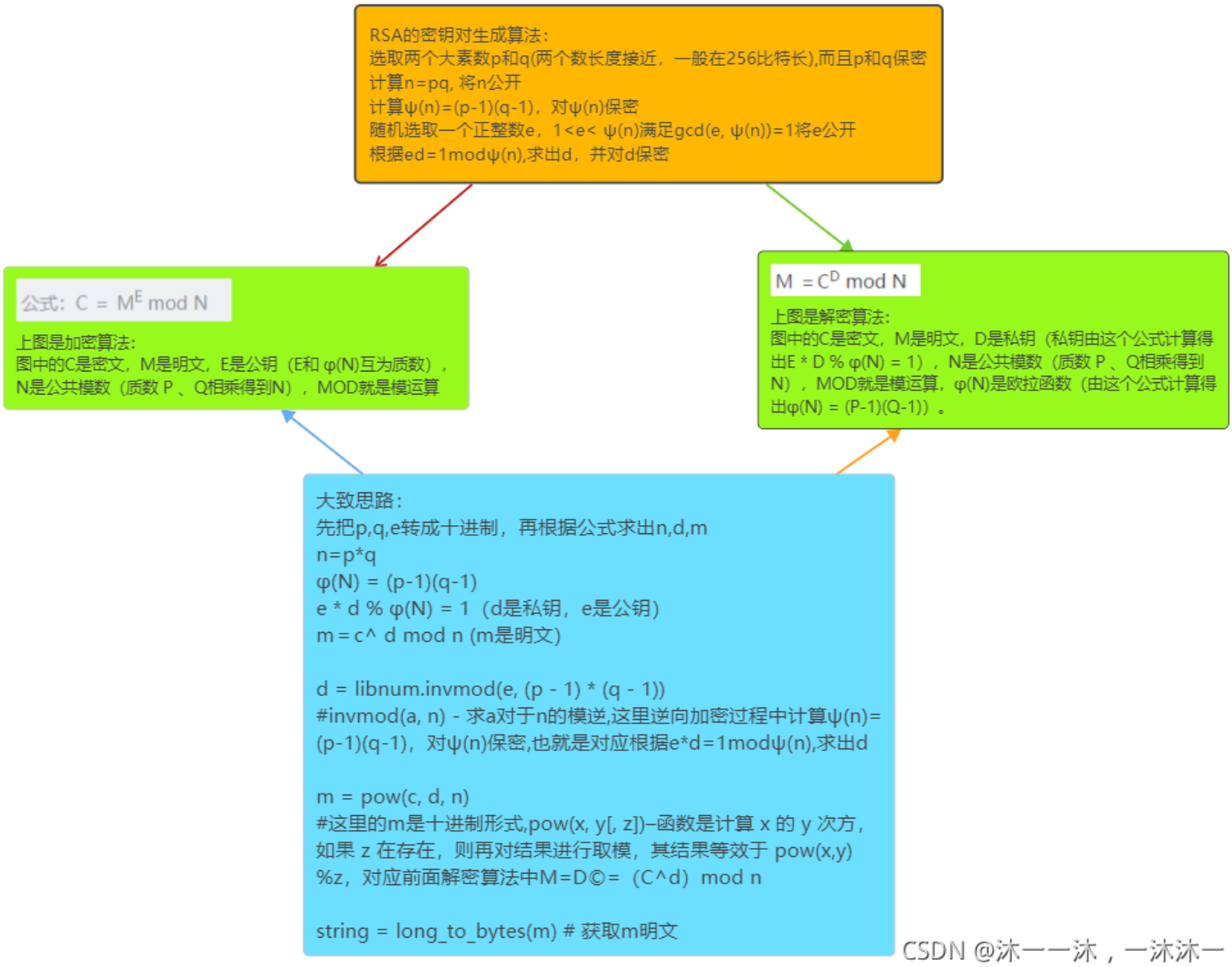
```
n = 497173522389038132581679656348726441223635460077646001242117112508529043004662639934569016294650189660
e = 137203702513055021984537223031886297150200318545270430281942345877211867503923385688407419809136028288
c = 339959282219630878273388034415899015715280698706451230817010044015469192892656193972643297879339983976

n = 497173522389038132581679656348726441223635460077646001242117112508529043004662639934569016294650189660
e = 135219279794171758254630633472228032676723381262369190976856395464196406491376318790865260550404406007
c = 206637277523316507362139933927937155550022837331244504843965961476614019383602527135957853427917306492
```


Basic Broadcast Attack(低加密指数广播攻击)

python2 solve.py --verbose -i examples/Basic_Broadcast_Attack.txt

```
c : 68679409966918700315304117144859068445528181933255289063193054014288151083466807594332167633810967321824633144 4
e : 7
n : 24810910852704603048663349011054669655631146433543459534796438815331335687309113943583212235150241971378068933 1

c : 11179052201843296851154916696601291938235676417475847173247453892471333698920515860207084422135916963942522752 2
e : 7
n : 47127839105299361033791208737798899776781255381503030381686909082155757361019104103280620540716894699133142173 1

c : 40118076943735337559537379692982070943921515348000211097599356263330760075906748374129727526740438883695094503 1
e : 7
n : 43134291711046821358455351358884087777021003839470296505990450581706219379356272391794220129036895199873385802 5

c : 12649076592222649371192164869044025408231371717627780046219346377852024544337050152652676577122342534868958091 7
e : 7
n : 19300838921149221007298944887478599082800229045219271606272038103970656559943914197281654158587468730541828306 4

c : 28899089935435267588235897519846120393433214114341521238696384122507316899457327055029546972333281452563984838 4
e : 7
n : 30754121488827635692971849599267749375077949182550303145729325375314926401905783830931628738658879320179944880 0

c : 14086629413855672403639830676118042465846020320143823318815048070368505684208141652603596234960968703217788472 9
e : 7
n : 30430477983470426195631142659668071772256641205525929891985872996115858010744648779370983539942187689192406517 4

c : 20049299207588955907155276095787913402589652379134151403340360498371893119855957833576443856534037886298731701 9
e : 7
n : 35489275126536805974281635942907480463916089663069129771420548612817030902692423639961709000309976531819984030 3
```

RSA通用的简单脚本：（已知p、q、e、c值）




```
import libnum
```

```
from Crypto.Util.number import long_to_bytes

q = int(

"0xa6055ec186de51800ddd6fcbf0192384ff42d707a55f57af4fcfb0d1dc7bd97055e8275cd4b78ec63c5d592f567

16)

p = int(

"0xfa0f9463ea0a93b929c099320d31c277e0b0dbc65b189ed76124f5a1218f5d91fd0102a4c8de11f28be5e4d0a

16)

e = int(

"0x6d1fdab4ce3217b3fc32c9ed480a31d067fd57d93a9ab52b472dc393ab7852fbcb11abbebfd6aaae8032db131

16)

c =
0x7fe1a4f743675d1987d25d38111fae0f78bbea6852cba5beda47db76d119a3efe24cb04b9449f53becd43b0b46

n = q * p

d = libnum.invmod(e, (p - 1) * (q - 1)) #invmod(a, n) - 求a对于n的模逆,这里逆向加密过程中计算ψ(n)=(p-1)(q-1)，对ψ(n)保密,也就是对应根据ed=1modψ(n),求出d

m = pow(c, d, n) # pow(x, y[, z])--函数是计算 x 的 y 次方，如果 z 在存在，则再对结果进行取模，其结果等效于 pow(x,y) %z，对应前面解密算法中M=D(C)=C^d(mod n)

#print(m) #明文的十进制格式

string = long_to_bytes(m) # m明文，用长字节划范围

print(string.decode())
```

ECC加密：

介绍：

椭圆曲线密码学（英语：Elliptic curve cryptography，缩写为 ECC），一种建立公开密钥加密的算法，基于椭圆曲线数学。

ECC的主要优势是在某些情况下它比其他的方法使用更小的密钥——比如RSA加密算法——提供相当的或更高等级的安全。ECC的另一个优势是可以定义群之间的双线性映射，基于Weil对或是Tate对；双线性映射已经在密码学中发现了大量的应用，例如基于身份的加密。其缺点是同长度密钥下加密和解密操作的实现比其他机制花费的时间长 [1] ，但由于可以使用更短的密钥达到同级的安全程度，所以同级安全程度下速度相对更快。一般认为160比特的椭圆曲线密钥提供的安全强度与1024比特RSA密钥相当。

关键总结：

设私钥、公钥分别为k、K，即K = kG，其中G为G点。

公钥加密：

选择随机数r，将消息M生成密文C，该密文是一个点对，即：

C = {rG, M+rK}，其中K为公钥

私钥解密：

M + rK - k(rG) = M + r(kG) - k(rG) = M

其中k、K分别为私钥、公钥。

题目样式举例：

已知椭圆曲线加密Ep(a,b)参数为

p = 15424654874903

a = 16546484

b = 4548674875

G(6478678675,5636379357093)

私钥为

k = 546768

求公钥K(x,y)

ECC脚本积累(解出最后的公钥和私钥即可)：

```
import collections
import random
EllipticCurve = collections.namedtuple('EllipticCurve', 'name p a b g n h')
curve = EllipticCurve(
'secp256k1',
# Field characteristic.
p=int(input('p=')),
# Curve coefficients.
a=int(input('a=')),
b=int(input('b=')),
# Base point.
g=(int(input('Gx=')),
int(input('Gy='))),
# Subgroup order.
n=int(input('k=')),
# Subgroup cofactor.
h=1,
```

```
)
# Modular arithmetic ########################################################
def inverse_mod(k, p):
"""Returns the inverse of k modulo p.
This function returns the only integer x such that (x * k) % p == 1.
k must be non-zero and p must be a prime.
"""
if k == 0:
raise ZeroDivisionError('division by zero')
if k < 0:
# k ** -1 = p - (-k) ** -1 (mod p)
return p - inverse_mod(-k, p)
# Extended Euclidean algorithm.
s, old_s = 0, 1
t, old_t = 1, 0
r, old_r = p, k
while r != 0:
quotient = old_r // r
old_r, r = r, old_r - quotient * r
old_s, s = s, old_s - quotient * s
old_t, t = t, old_t - quotient * t
gcd, x, y = old_r, old_s, old_t
assert gcd == 1
assert (k * x) % p == 1
return x % p
# Functions that work on curve points #########################################
def is_on_curve(point):
"""Returns True if the given point lies on the elliptic curve."""
if point is None:
# None represents the point at infinity.
return True
```

```
x, y = point
return (y * y - x * x * x - curve.a * x - curve.b) % curve.p == 0
def point_neg(point):
"""Returns -point."""
assert is_on_curve(point)
if point is None:
# -0 = 0
return None
x, y = point
result = (x, -y % curve.p)
assert is_on_curve(result)
return result
def point_add(point1, point2):
"""Returns the result of point1 + point2 according to the group law."""
assert is_on_curve(point1)
assert is_on_curve(point2)
if point1 is None:
# 0 + point2 = point2
return point2
if point2 is None:
# point1 + 0 = point1
return point1
x1, y1 = point1
x2, y2 = point2
if x1 == x2 and y1 != y2:
# point1 + (-point1) = 0
return None
if x1 == x2:
# This is the case point1 == point2.
m = (3 * x1 * x1 + curve.a) * inverse_mod(2 * y1, curve.p)
else:
```

```
# This is the case point1 != point2.
m = (y1 - y2) * inverse_mod(x1 - x2, curve.p)
x3 = m * m - x1 - x2
y3 = y1 + m * (x3 - x1)
result = (x3 % curve.p,
-y3 % curve.p)
assert is_on_curve(result)
return result
def scalar_mult(k, point):
"""Returns k * point computed using the double and point_add algorithm."""
assert is_on_curve(point)
if k < 0:
# k * point = -k * (-point)
return scalar_mult(-k, point_neg(point))
result = None
addend = point
while k:
if k & 1:
# Add.
result = point_add(result, addend)
# Double.
addend = point_add(addend, addend)
k >>= 1
assert is_on_curve(result)
return result
# Keypair generation and ECDHE ################################################
def make_keypair():
"""Generates a random private-public key pair."""
private_key = curve.n
public_key = scalar_mult(private_key, curve.g)
return private_key, public_key
```

```
private_key, public_key = make_keypair()

print("private key:", hex(private_key))

print("public key: (0x{:x}, 0x{:x})".format(*public_key))
```