【CTF WriteUp】2020电信和互联网行业赛个人赛部分Crypto题解

原创

零食商人 于 2020-12-21 11:17:41 发布

Crypto

（话说题目做一半就当答案是什么鬼）

Crypto-bacon

题目

```
flag{AAAABAAAAAAAABAABBBAABBABABAAABAABAAAABBAABAAABABBABAAAAAABAABAAAABBBABABAABAABA}
```

解答

简单的培根密码，略

Crypto-黄金分割RSA

题目

encryption

```
[1,28657,2,1,3,17711,5,8,13,21,46368,75025,34,55,89,610,377,144,233,1597,2584,4181,6765,10946,987]
```

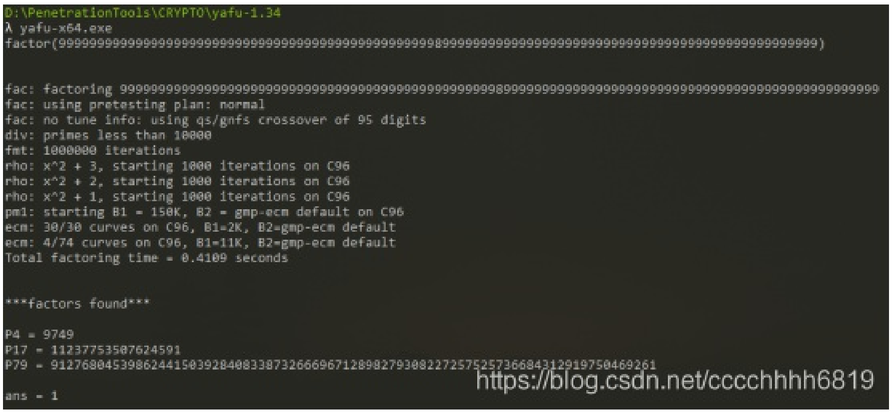
output

```
publickey=[0x1d42aea2879f2e44dea5a13ae3465277b06749ce9059fd8b7b4b560cd861f99144d0775ffffffffffff,5]
c=421363015174981309103786520626603807427915973516427836319727073378790974986429057810159449046489151
```

解答

（本题为2020 GACTF-da Vinci after rsa，然后河南天安杯又考一遍，这是第三遍）

首先使用yafu分解n，得到三个质因数



注意到e为5，与其中的q = P17-1和r = P79-1均不互质，所以需要尝试另外的解法。假设明文为m，那么m应该满足

```
m^5 = c (mod p)
m^5 = c (mod q)
m^5 = c (mod r)
```

在线上赛的情况下，我们可以利用网站www.wolframalpha.com求解m模p、q、r的可能值，并利用中国剩余定理求出m的所有可能值，进而判断其中哪一个是flag。

```
p_roots = [7361]
q_roots = [2722510300825886, 6139772527803903, 6537111956662153, 8415400986072042, 9898464751509789]
r_roots = [1809664152256324651202082723661084756679340824052388089580482942870112436452816114411493328258682873357893989007684496552202823306045771363205185148674391, 1369135259891793292334345751773139388112378132927363770631732500241630990458667, 5570877862584063114417410584640901580756179707042774516590562822938385811269597, 849905240758807800288593176516613730839707423236108768297444863394635053929222]
```

写出解题代码如下：

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from Crypto.Util.number import *
import gmpy2

def GCRT(mi, ai):
    assert (isinstance(mi, list) and isinstance(ai, list))
    curm, cura = mi[0], ai[0]
    for (m, a) in zip(mi[1:], ai[1:]):
        d = gmpy2.gcd(curm, m)
        c = a - cura
        assert (c % d == 0)  # 不成立则不存在解
        K = c / d * gmpy2.invert(curm / d, m / d)
        cura += curm * K
        curm = curm * m / d
    return (cura % curm, curm)  # (解,最小公倍数)

n = 0x1d42aea2879f2e44dea5a13ae3465277b06749ce9059fd8b7b4b560cd861f99144d0775ffffffffffff
c = 421363015174981309103786520626603807427915973516427836319727073378790974986429057810159449046489151
p = 9749
q = 11237753507624591
r = n / p / q
e = 5

p_roots = [7361]
q_roots = [2722510300825886, 6139772527803903, 6537111956662153, 8415400986072042, 9898464751509789]
r_roots = [180966415225632465120208272366108475667934082405238808958048294287011243645, 281611441149332825868287
3357893989007684496552202823306045771363205185148674391, 1369135259891793292334345751773139388112378132927363770
631732500241630990458667, 5570877862584063114417410584640901580756179707042774516590562822938385811269597, 84990
52407588078002885931765166137308397074232361087682974448633946350539292222]

m_list = []
for pp in p_roots:
    for qq in q_roots:
        for rr in r_roots:
            res = GCRT([p, q, r], [pp, qq, rr])[0]
            if pow(res, e, n) == c:
                print long_to_bytes(res)
```

得到一个字符串flag{weadfa9987_adwd23123_454f}，但是我们还有一个条件没有使用，即一串数

```
[1,28657,2,1,3,17711,5,8,13,21,46368,75025,34,55,89,610,377,144,233,1597,2584,4181,6765,10946,987]
```

仔细观察发现，这些数就是斐波那契数列的一个排列，且其总数为25，恰好与flag{}内的字符数相同，因此 **将字符按照同样方式打乱后** 可以得到真实flag。

	A	B	C	D	E
1	w	1		1	w
2	e	1		28657	5
3	a	2		2	a
4	d	3		1	e
5	f	5		3	d
6	a	8		17711	4
7	9	13		5	f
8	9	21		8	a
9	8	34		13	9
10	7	55		21	9
11	_	89		46368	4
12	a	144		75025	f
13	d	233		34	8
14	w	377		55	7
15	d	610		89	_
16	2	987		610	d
17	3	1597		377	w
18	1	2584		144	a
19	2	4181		233	d
20	3	6765		1597	3
21	_	10946		2584	1
22	4	17711		4181	2
23	5	28657		6765	3
24	4	46368		10946	_
25	f	75025		987	2
26					

（不要问我为什么原题需要按加密方式打乱而不是按照加密方式进行恢复，问GACTF出题人）

Crypto-Corrupted Keys

题目

ciphertext.txt

```
c = 0x6f9c3479883b414030032610a0831089ea2d5f6598d16f8b3415dbb7ff88e6214c7704dbaf1f0f0fe8243468b203b0c128933ab45f406109d234ab94457aa4ff81de3e0c1dda55b95344683e7cfef4e39dedd1203120af89e14702ac54a1a21adb500dadc67033deb2dcf844aa10c5b6425aca0a756ee5e5ce5b583de68d7dfa675b8142c4b175b347bd1c3b2d2cd32aa2e03356ecf4821704d7b7542a22d09ebb239e382fc5b72ea051b65596e41d228fb7b0f7acf5686d05b8d6807a26c1a1d92c8b116c6f27e2b21ded5f1f3b8f9a88e45ec7b14aee18e74454fefb1a482a9eafc9550d16f6683e2f7cbd0d9ce9a474f4db01e2f97d0d3d23fad566489e1e
```

private_key.pem

```
-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEAANAL4ECQAIAAsACAUJBfAA0NIAADdwAIAGAAAAALoAAAAABw
PwBQAAAAnwAAcP4NAAE3QFAAcJMKwJbGANcAADCjDhC7DgAAANIAMAoAXgAAMQAA
AOAAIAYQIFEJAAgCYA8GwAHgAw1KcFhAAHzwAKAAAAA2BgMAYEBZIDcnAAEMMAAE
QcAAUVfQAD3VEABAsQAAAAAMAAegilfw0A8ALWABBAIKAAAAAIQAwApwkArfyDwG
ALAAAAQA9FBAoBqJAKAAgKUABzOQawlnoGAgUMYAoAAAACAocAHowABAAwUHAJCy
AgUFEA4AAAA0AIAOAOAAjwDADQDUeAAKIA7DAAIDAQABAoIBAAAAAAAAAAAAADgAA
APvgAAEAAAAAAAAAAAAACgAAhAAAAABgAAAJAAAAAAAAAAAADAAAAA0AAAAA4A4A
AAAAAGAAAAAAAEAAAAAAAAAAAAAAAAAACwAAgAAAAAIAAAAA0AAAAAAAABAACBAA
AA8AAAAABgAAkAAAANAAAAAABgAAAAAAwAAIAAAAAADgAAAAAAAAAAADAACAAAAA
AAAAAAAAACgAAAAAAAAAAAAAABgABAAAAAAsAACAAgAAAAAAAAAAAABAAAAwcACQ
AAAOAN0AAAEHAAAAAAAAAAAAAAAAAAAAAAAcAAAAAAAAAAAQAAAAAAAAAAAAA
AAAFAAACgYEAAAAAAAAAQAAAAAAAAAAAAAFAAAAAAgAMAcACgAKAAAArAAAAAAAA
AAAAAAAAAACQADAAAAAAAAAAAAAAAACwAAAACwAAgAAAAAAAANAAAAAAAAAAAAAAg
AAAAAAAMAGAAAAAAAAAAAsAAAAQACAAADAAAAAAAAAAAAFAAAAAAAAAAACgYEAAACw
AKAAAAAA0AAAEAAHAAAAAAAAAAAAPQAAAAAAAAAAAAACAAAA4AALMAAAAABQAAoAAA
AAAAAAAAAABQAAAAAKAOAAACAAAAAAAAAAwIAAAAAAA4AMAAAAAAAAAGQAAGAAAAAA
AAADAACAAAAJAAAAUAUAAAAAAAAAAAAAAAAAAAAACgYAFOW7M5UcyswjtKNXo783B
hDUPHTPG49nzxsU33eLi8pxZ6hFaFPaEO8NBkBHMqPI6lPn/wuisNQvWna5igQEA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAg0Fr7+S3oAktME9GMkvPYiqk1qeDYLZX
1mkrXy1v2aJ74q+J41UdYwKBgAAg8QsAAACgBQCQAAAAAAAAAAgAAqQAABgDQAAAA
AAAAAAAAcAoAAAABAAwAAAAwBgAAsw0gCAAAAAAAAAgAAAAA8AAAAAAAAAEDwAAAA
AACgAAAAAAAGAAAAAAAAACVAAUAAAAADgAAAAAAAAkAAAgAAKAAAAAAAALAAAAAADQ
5AAAAoGBAAAAAAAAAoAUA0AwCDwAABADJAAAAAHAAAAAAAAADgAAAAAAAAAAAAAI8A
AAYPAMAAADAAAAIAAAAACQAAgAAAoAcAAAAAAAAAAAAAAAAAAADAAAAAAA0AAAAAAZ
......
```

public_key.pem

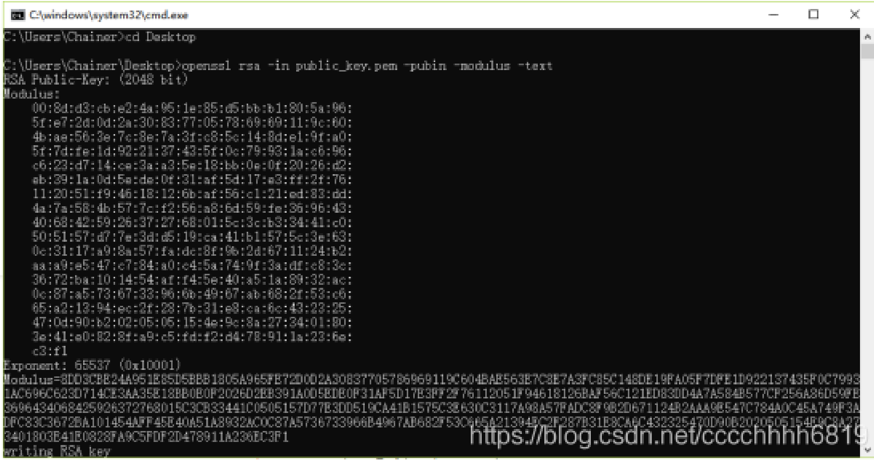
```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAjdPL4kqVHoXVu7GAWpZf
5y0NKjCDdwV4aWkRnGBLrlY+fI56P8hcFI3hn6Bfff4dkiE3Q18MeZMaxpbGI9cU
zjqjXhi7Dg8gJtLrORoNXt4PMa9dF+P/L3YRIFH5RhgSa69WwSHtg91KelhLV3zy
VqhtWf42lkNAaEJZJjcnaAFcPLM0QcBQUVfXfj3VGcpBsVdcPmMMMRepilf63I+b
LWcRJLKqqeVHx4SgxFp0nzrfyDw2croQFFSv9F5ApRqJMqwMh6VzZzOWa0lnq2gv
U8ZlohOU7C8oezHoymxDIyVHDZCyAgUFFU6ciic0AYA+QeCCj6nF/fLUeJEaI27D
8QIDAQAB
-----END PUBLIC KEY-----
```

解答

这道题本身设置了两个考点：一是如何从残缺的私钥中提出信息，二是如何利用中间被挖去一段的dp来分解n。前者参见前两年0ctf的一道题，后者参见今年11月辽宁祥云杯。但是这道题私钥直接给全了，导致使用openssl命令直接可以提出数据，第一个考点直接失效。应该如同上边一样给出。

开始解体。首先从公钥中提出n和e

openssl rsa -in public_key.pem -pubin -modulus -text



但是仅有n、e、c是无法做题的，因为n不能直接分解，所以我们还需要看看私钥给我们留下了什么。根据资料，我们得知RSA的私钥通常以PKCS#1的模式进行存储，简单地说如下所示：

```
RSAPrivateKey ::= SEQUENCE {
  version           Version,
  modulus           INTEGER,  -- n
  publicExponent    INTEGER,  -- e
  privateExponent   INTEGER,  -- d
  prime1            INTEGER,  -- p
  prime2            INTEGER,  -- q
  exponent1         INTEGER,  -- d mod (p-1)
  exponent2         INTEGER,  -- d mod (q-1)
  coefficient       INTEGER,  -- (inverse of q) mod p
  otherPrimeInfos   OtherPrimeInfos OPTIONAL
```

我们将现在已经被污染的私钥base64解码后，按照上边的格式展开，得到如下内容：

```
308204a3020100
0282010100
(n)00d00be04090008000b0008050905f000d0d20000377000800600000000ba000000000703f00500000009f000070fe0d0001374050007
0930ac096c600d7000030a30e10bb0e000000d200300a005e000031000000e00020061020510900802600f06c001e0030d4a705840007cf
000a00000003606030060405920372700010c30000441c0005157d0003dd5100040b1000000000c0007a08a57f0d00f002d600104020a000
000008400c00a70900adfc83c0600b000000400f45040a01a8900a00080a5000733906b0967a0602050c600a000000020287001e8c000400
305070090b2020505100e0000003400800e00e0008f00c00d00d478000a200ec300
0203
(e)010001
02820100
(d)0000000000000000e000000fbe000010000000000000000000000a0000840000000060000009000000000000000000c0000000d00000
000e00e000000000060000000000400000000000000000000000000000b00008000000002000000000d000000000000001000081000000f000
0000006000090000000d00000000000600000000000c00008000000000000e000000000000000000300008000000000000000000000000a000000
000000000000000006000100000000b00002000800000000000000000000004000003070009000000e00dd000001070000000000000000000000
0000000000000007000000000000000000100000000000000000000000000000050000
02818100
(p)000000000100000000000000000000005000000000800300700a000a000000ac00000000000000000000000000000090003000000000
00000000000b00000000b000080000000000000d0000000000000000000000002000000000000c00600000000000000b0000001000200000
300000000000000000050000000000000000
02818100
(q)0000b000a000000000d000001000070000000000000000000f400000000000000000000002000000e0000b30000000000050000a000000000000
0000000005000000000a00e000002000000000000000c08000000000e0030000000000000064000060000000000000000300008000000900000
05005000000000000000000000000000000
028180
(dp)05396ecce54732b308ed28d5e8efcdc184350f1d33c6e3d9f3c6c537dde2e2f29c59ea115a14f6843bc3419011cca8f23a94f9ffc2e8
ac350bd69dae628101000000000000000000000000000000000000000000000000083416befe4b7a0092d304f46324bcf622aa4d6a78360
b657d6692b5f2d6fd9a27be2af89e3551d63
028180
(dq)0020f10b000000a005009000000000000000020000a900000600d0000000000000000000000700a00000001000c00000030060000b30d2008
0000000000000800000000f0000000000000040f00000000000a000000000000600000000000000950005000000000e0000000000000000900000
80000a0000000000b0000000000d0e40000
02818100
(coeff)0000000000a00500d00c020f00000400c90000000007000000000000000e000000000000000000000008f0000060f00c0000030000002
0000000090000800000a00700000000000000000000000000c0000000000d00000000019（残缺）0
```

可以看到，除了dp以外其他内容基本已经无法使用，dp只缺少其中的200位，因此可以尝试使用coppersmith方法去解。由于coppersmith方法需要未知变量系数为1，我们这里尝试推导一下：

我们知道

$$dp * e = k * (p-1) + 1$$

由于dp <= p-1，所以k<e，coppersmith方法通过遍历1~e寻找k。现在假设k已知，那么有

$$dp * e + k - 1 = 0 \pmod{p}$$

设

$$dp = s_1 * 2^{520} + x * 2^{320} + s_2$$

带入原式得

$$(s_1*2^{520}+x*2^{320}+s_2)*e+k-1=0 \pmod p$$

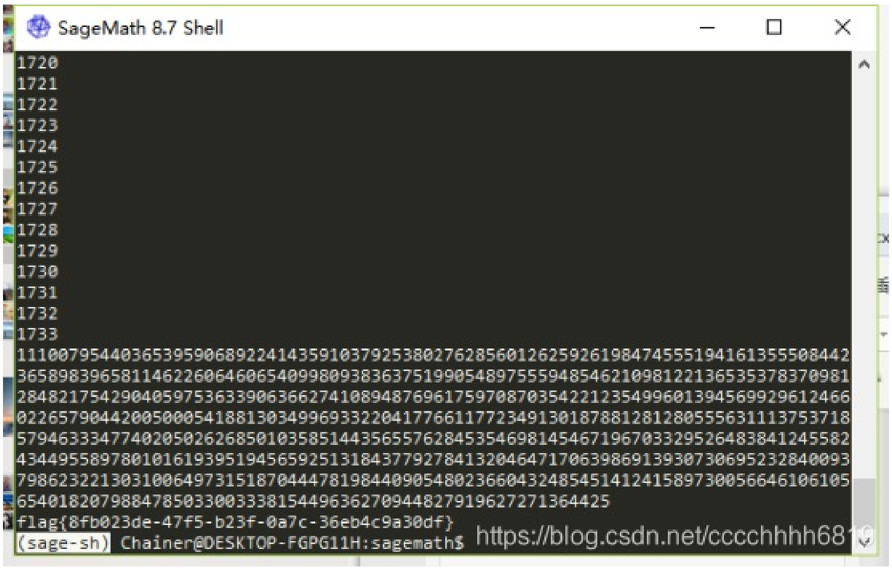
$$s_1*2^{520}+x*2^{320}+s_2+(k-1)*inv(e,n)=0 \pmod p$$

$$s_1*2^{520}*inv(2^{320},n)+x+s_2*inv(2^{320},n)+(k-1)*inv(e,n)*inv(2^{320},n)=0 \pmod p$$

这样就构造出了系数为1的同余方程，可以使用coppersmith方法解题了。解题脚本如下：

```
from sage.all import *
from Crypto.Util.number import long_to_bytes

n = 0x8dd3cbe24a951e85d5bbb1805a965fe72d0d2a30837705786969119c604bae563e7c8e7a3fc85c148de19fa05f7dfe1d922137435f
0c79931ac696c623d714ce3aa35e18bb0e0f2026d2eb391a0d5ede0f31af5d17e3ff2f76112051f94618126baf56c121ed83dd4a7a584b57
7cf256a86d59fe369643406842592637276801 5c3cb33441c0505157d77e3dd519ca41b1575c3e630c3117a98a57fadc8f9b2d671124b2aa
a9e547c784a0c45a749f3adfc83c3672ba101454aff45e40a51a8932ac0c87a5736733966b4967ab682f53c665a21394ec2f287b31e8ca6c
432325470d90b2020505154e9c8a273401803e41e0828fa9c5fdf2d478911a236ec3f1
e = 0x10001
c = 0x6f9c3479883b414030032610a0831089ea2d5f6598d16f8b3415dbb7ff88e6214c7704dbaf1f0f0fe8243468b203b0c128933ab45f
406109d234ab94457aa4ff81de3e0c1dda55b95344683e7cfef4e39dedd1203120af89e14702ac54a1a21adb500dadc67033deb2dcf844aa
10c5b6425aca0a756ee5e5ce5b583de68d7dfa675b8142c4b175b347bd1c3b2d2cd32aa2e03356ecf4821704d7b7542a22d09ebb239e382f
c5b72ea051b65596e41d228fb7b0f7acf5686d05b8d6807a26c1a1d92c8b116c6f27e2b21ded5f1f3b8f9a88e45ec7b14aee18e74454fefb
1a482a9eafc9550d16f6683e2f7cbd0d9ce9a474f4db01e2f97d0d3d23fad566489e1e
s1 = 0x5396ecce54732b308ed28d5e8efcdc184350f1d33c6e3d9f3c6c537dde2e2f29c59ea115a14f6843bc3419011cca8f23a94f9ffc2
e8ac350bd69dae628101
s2 = 0x83416befe4b7a0092d304f46324bcf622aa4d6a78360b657d6692b5f2d6fd9a27be2af89e3551d63
invE = 607246348005222277448444018894068784893261339674776416517053926300280106832311997118643400122356432947960
3123138991852830747459341281298547464945329258352716978884436175214634055317578153997659440696241077620973159496
0444089673635388970465490762413236424460315237883100650480470692160869167806717582595930549357426298413851694659
3961101500800318974736058847108299210208951476938115767279186343649800059109625400737095596094353444034846045686
1334522885919455014210659853726203177705141103038754371809891816964608801118434914236230355247754146610175756211
785759304159230507837819605093784920450130362742047348692105621
invpow = 1116443150207074751150053002174551090668533485688766572101159798693207919778016196338123839488045291036
2678853293436976134225721195690734856639343780389327693213031686657799035703254166242337351340806493525437022390
0703723931988947315436601244964166984345494128713782600950862786002028205007904344114955555693973765288323596742
9254789631975028782111078451934403051031125206688037746177755118530359200184630061799358019685860767135508139753
8545831351265009506760347727474227285668988817907365295771630760269871647111667154970535637451277321913843646237
831493800721780046762737829851933071451680344164843758329746830379

def coppersmith(k):
    F.<x> = PolynomialRing(Zmod(n))
    f = (s1 << 520) * invpow + x + s2 * invpow + (k - 1) * invE * invpow   # make monic
    x0 = f.small_roots(X=2 ** 200, beta=0.44, epsilon=1/32)
    return x0

for k in range(1, e):
    print k
    x0 = coppersmith(k)
    if len(x0) != 0:
        x = Integer(x0[0])
        dp = (s1 << 520) + (x << 320) + s2
        p = (e*dp - 1) // k + 1
        if p != -1:
            q = n // p
            assert n == p * q
            phi = (p-1)*(q-1)
            d = inverse_mod(e,phi)
            print d
            print long_to_bytes(pow(c,d,n))
            break
```

其中invE、invpow分别是计算好的invert(e, n)和invert(2 ** 320,n)，最终求出k=1733。



Crypto-strange_GSW

题目

strange_GSW.sage

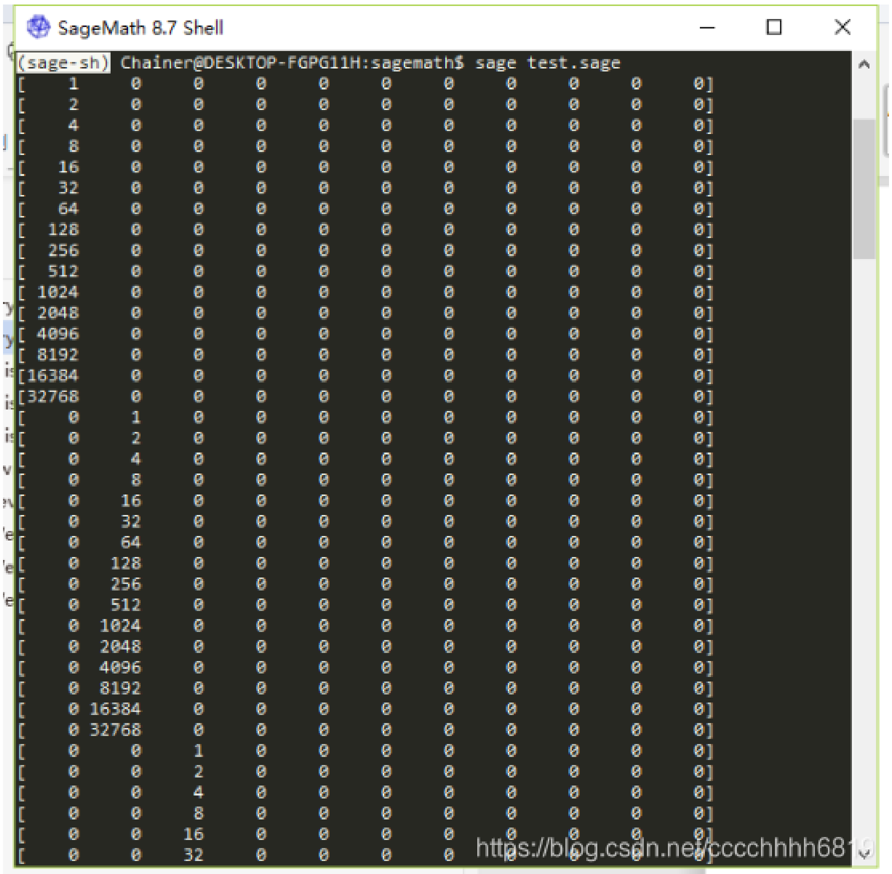
```
from Crypto.Util.number import *
from random import randrange
from hashlib import md5
from secret import FLAG


def GenerateG(_n, _q):
    _len = int(round(log(_q, 2)))
    _G = Matrix(ZZ, _len * _n, _n)
    for i in range(_len):
        for j in range(_n):
            _G[j * _len + i, j] = 2 ** i
    return _G


def BinaryExpansion(_C, _q):
    expansion_C = Matrix(ZZ, _C.nrows(), _C.nrows())
    log_q = _C.nrows() // _C.ncols()
    for i in range(_C.nrows()):
        for j in range(_C.ncols()):
            bits = bin(_C[i, j] % _q)[2:].rjust(log_q, '0')[::-1]
            for index in range(log_q):
                expansion_C[i, j * log_q + index] = int(bits[index])

    return expansion_C


def KeyGen(degree, _q, _B):
    s = random_matrix(ZZ, degree, 1, x=_q // 4, y=4 * _q // 3)
    return Matrix(s.list() + [-1]).transpose(), s


def BitEncrypt(plain_bit, _n, _q, _B, _G, _s):
```

```
    _m = int(round(log(_q, 2)) * _n)
    A = random_matrix(ZZ, _m, _n - 1, x=0, y=_q)
    e = random_matrix(ZZ, _m, 1, x=0, y=_B)
    _C = block_matrix([A, A * _s + e], ncols=2, subdivide=False) + plain_bit * _G
    _C = BinaryExpansion(_C, _q)
    return _C


def Encrypt(plain, _n, _q, _B, _G, _s):
    plain_bits = bin(plain)[2:]
    Cipher = []
    for bit in plain_bits:
        Cipher.append(BitEncrypt(int(bit), _n, _q, _B, _G, _s))
    return Cipher


if __name__ == '__main__':
    n = 11
    q = 65537
    B = q // 256
    G = GenerateG(n, q)
    decrypt_key, encrypt_key = KeyGen(n - 1, q, B)
    flag = FLAG + md5(long_to_bytes(randrange(2 ** 127, 2 ** 128))).hexdigest().encode()[:8] + b'}'

    _cipher = Encrypt(bytes_to_long(flag), n, q, B, G, encrypt_key)
    str_cipher = ' '.join([str(i.list()) for i in _cipher])
    with open('flag', 'wb') as f:
        f.write(flag)
    with open('cipher', 'w') as f:
        f.write(str_cipher)
```

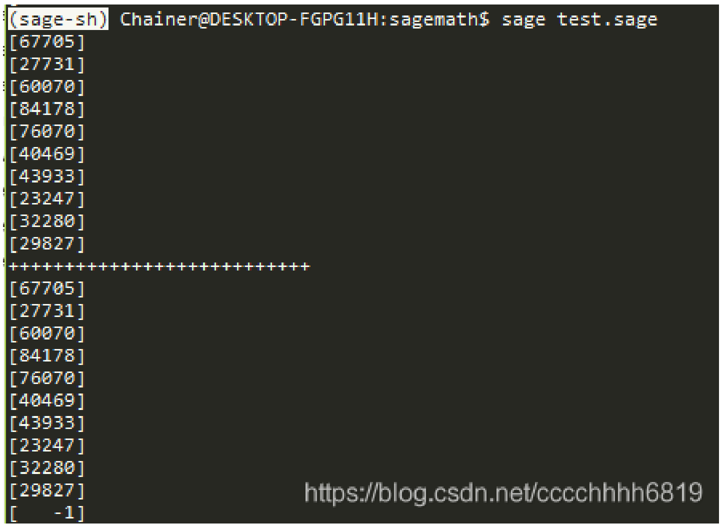
解答

（比赛好像没人做出来，确实非常费劲）

题目看起来比较复杂，所以我们跟着程序走，看看具体流程是什么样的。首先看矩阵G



可知矩阵G是一个固定的矩阵。然后生成一组加密密钥和解密密钥看看



可以看到加密密钥为一个10 * 1的矩阵，其中每项的值在(q/4, q*4/3)之间，q = 65537。解密密钥就比加密密钥多一个-1。

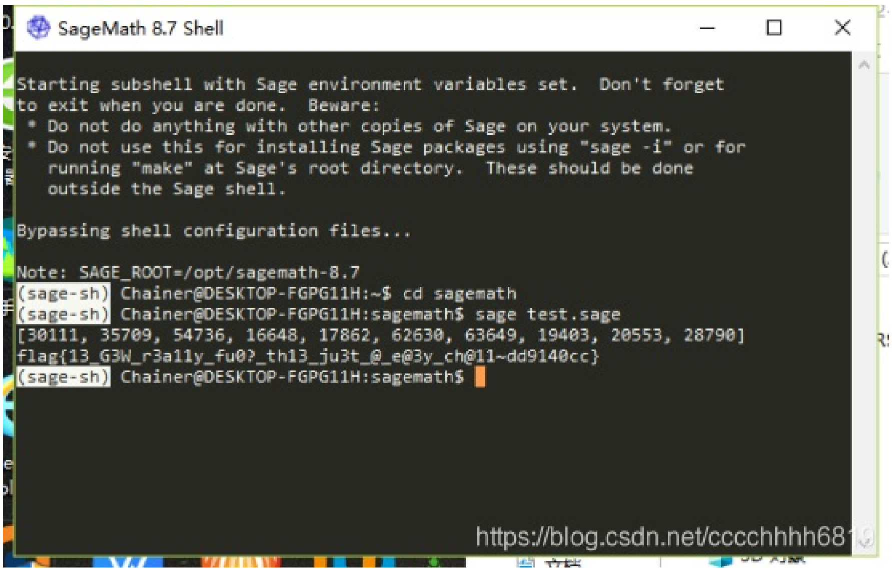
最后看加密过程，首先将密文转为整数，再转为二进制，然后对于二进制表示的每一位（0或1）进行BitEncrypt加密操作，将这个0或者1加密成一个176 * 176的由0和1构造成的矩阵。加密的过程中参数生成随机，所以相同明文加密结果不固定。加密生成的_C矩阵为一个176 * 11的矩阵，再经过BinaryExpansion函数扩展到一个176 * 176的矩阵。这个扩展很简单，就是把这些数每一个按照其二进制表示横向展开成16位，然后倒序写入，因此我们可以简单将其收起得到_C。

```python
def recoverMatrix(text):
    text = text.replace(',','').replace(' ','').replace('[','')
    Mdata = []
    for i in range(0, len(text), 16):
        Mdata.append(int(text[i:i+16][::-1],2))
    _C = Matrix(ZZ, 176, 11, Mdata)
    return _C
```

在得到_C之后，我们来尝试求解key。_C的结构是矩阵A拼接一列A * _s + e的结果，其中e是0~256以内的随机值，再根据明文该位置是0还是1来决定是否加上矩阵_G。考虑到大部分参数取值范围在(q/4, q*4/3)之间，因此e的(0, 256)区间可以视为小值，可以将本题当作一个格上的最近向量问题（CVP, Closet vector problem）进行求解。对明文首位进行求解时，针对结果可能为1和0的两种情况，求出的_C矩阵需要减去_G 或者 不发生变化。当存在解时，首位明文正确，且求解出的key就是加密所用key，后续每个明文字符直接用该key还原即可。完整解题代码如下：

```python
from sage.modules.free_module_integer import IntegerLattice
from Crypto.Util.number import long_to_bytes

def GenerateG(_n, _q):
    _len = int(round(log(_q, 2)))
    _G = Matrix(ZZ, _len * _n, _n)
    for i in range(_len):
        for j in range(_n):
            _G[j * _len + i, j] = 2 ** i
    return _G

def recoverMatrix(text):
    text = text.replace(',','').replace(' ','').replace('[','')
    Mdata = []
    for i in range(0, len(text), 16):
        Mdata.append(int(text[i:i+16][::-1],2))
    _C = Matrix(ZZ, 176, 11, Mdata)
    return _C

def CVP(lattice, target):
    gram = lattice.gram_schmidt()[0]
    t = target
    for i in reversed(range(lattice.nrows())):
        c = ((t * gram[i]) / (gram[i] * gram[i])).round()
        t -= lattice[i] * c
    return target - t

def BitDecrypt(_C, _G, _s, _q):
    _C = _C * _G * _s
    return 0 if abs(_C.list()[0] % _q - _q) < _q // 4 else 1

def Decrypt(Cipher, _G, _s, _q):
    plain_bits = ''
    for cipher in Cipher:
        plain_bits += str(BitDecrypt(cipher, _G, _s, _q))
    return int(plain_bits, 2)

if __name__ == '__main__':
    n = 11
    q = 65537
    B = q // 256
    G = GenerateG(n, q)

    with open('cipher', 'r') as f:
        cipher_str = f.read()
```

```
    cipher_str = cipher_str.split(']')[:-1]

    # Store Ciphertext
    _cipher_str = [eval(i.strip()+']') for i in cipher_str]
    _cipher_matrix = [Matrix(176, 176, i) for i in _cipher_str]

    # Recover Matrix _C
    _C = recoverMatrix(str(cipher_str[0]))
    _C = _C - G
    _A = _C[:176, :10]
    res = _C[:176, 10:].list()

    # Make a Matrix for CVP
    M = Matrix(ZZ, 186, 176)
    for i in range(176):
        for j in range(10):
            M[176 + j, i] = int(_A[i][j])
        M[i, i] = 65537

    lattice = IntegerLattice(M, lll_reduce=True)
    target = vector(ZZ, res)
    res = CVP(lattice.reduced_basis, target)

    # Recover Key
    R = IntegerModRing(65537)
    M = Matrix(R, _A[:176])
    key = M.solve_right(res)
    key = [int(i) for i in key]
    print key

    # get flag
    flag = long_to_bytes(Decrypt(_cipher_matrix, G, Matrix(key + [-1]).transpose(), q))
    print(flag)
```



Crypto-random_fault

（我真不会国密。。。）