题目：

Our agents have gathered too many public keys that all of them were used to encrypt the secret flag. Can you decrypt the flag with a performant approach?

做题时间是在 2019.09，此时 CryptoCTF2019 比赛已经结束，比赛页面显示此题有 4 solves，但是没有搜到 Writeup。

代码：执行 python dec.py 即可得到 flag，部分前期工作用到的代码在 work.py 中。

附件中给了 15933 对公钥和密文，文件名分别是“pubkey_00000.pem”到“pubkey_15932.pem”和“flag_00000.enc”到“flag_19532.enc”。

用 python 的 RSA.importKey()函数读一下公钥文件，可以读到 n 和 e，其中 n 不少于 2048 位，e 全部是 65537。

这种情形第一想法是 CRT（中国剩余定理）搞一下就出来了。实际情况是：第一，一万多个大数全放进去的话，单是计算所有 n 的乘积就慢的很；第二，它解不出来。当然第二点是知道了正确解法之后知道的。

然后再尝试，这么多 n，试试 GCD 吧。把前 100 个 n 交叉 GCD 果然能出来很多公因数，是 256 位的。那除一下得到一堆一千七百多位的数是咋回事呢，那要不试试把这些也交叉 GCD 一下？结果又出来一些 256 位的。然后除完这些第二批 256 位的又可以得到一些新的大数。随即想到可以写个递归的程序，把它们继续这样分层 GCD 下去。

代码：

```
from Crypto.PublicKey import RSA
from Crypto.Util.number import long_to_bytes, bytes_to_long
import gmpy2
import random
import datetime

FILE_NUM = 100


def get_now():
    return datetime.datetime.now().strftime('%Y%m%d-%H%M%S')


def bit_len(n):
    return (len(hex(n))-2) * 4


def bit_len_list(a):
    num = len(a)
    if 1 > num:
        return 0
    N = 100
    tot = 0
    for i in range(N):
        tot += bit_len(list(a)[random.randint(0, num - 1)])
```

```python
    return tot/N


sessions_c = []
sessions_n = []
ancestor = dict()
divisor = dict()
for i in range(0, FILE_NUM):
    num = '%05d'%i
    with open('stuff\\keys\\pubkey_' + num + '.pem', 'r') as f:
        pubkey = RSA.importKey(f.read())
        n = getattr(pubkey, 'n')
        e = getattr(pubkey, 'e')
        if e != 65537:
            print(i)
    with open('stuff\\enc\\flag_' + num + '.enc', 'rb') as f:
        c = bytes_to_long(f.read())
    sessions_c.append(c)
    sessions_n.append(n)
    divisor[n] = set()
    ancestor[n] = set()
print("Reading files over.")
print(bit_len_list(sessions_c))
print(bit_len_list(sessions_n))


def full_divide(n, p):
    while 0 == n % p:
        n //= p
    return n


def link(n, p):
    if 1 == p:
        return
    if p not in ancestor.keys():
        ancestor[p] = set()
    if 0 == len(ancestor[n]):
        divisor[n].add(p)
        ancestor[p].add(n)
    else:
        for root in ancestor[n]:
            try: divisor[root].remove(n)
            except: pass
            divisor[root].add(p)
```

```python
        ancestor[p].add(root)


level = 0
def recursive_gcd(pool, level):
    if 0 == len(pool):
        return
    print("level =", level, " size =", len(pool), "now: ", get_now())
    new_pool1 = set()
    new_pool2 = set()
    d = 1
    for i in pool:
        for j in pool:
            if i == j:
                continue
            if bit_len(i) < 260 or bit_len(i) < 260:
                continue
            d = int(gmpy2.gcd(i, j))
            if d == 1:
                continue
            new_pool1.add(d)
            qi = full_divide(i, d)
            qj = full_divide(j, d)
            new_pool2.add(qi)
            new_pool2.add(qj)
            link(i, d)
            link(j, d)
            link(i, qi)
            link(j, qj)
            d = 1
    recursive_gcd(new_pool1, level + 1)
    recursive_gcd(new_pool2, level + 1)


recursive_gcd(sessions_n, 0)
```

递归函数是 recursive_gcd。

结果保存在 ancestor 和 divisor 两个字典中。字典 divisor 的 key 为所有的 n，value 是一个集合，其内容为 GCD 得到的 n 的因数；字典 ancestor 的 key 为得到的数，包括因数也包括原始的那些 n，value 也是一个集合，内容是 GCD 得到的这个数可以整除的所有的 n，如其名“ancestor”。

FILE_NUM 先设为 100，也就是说只对前 100 个 n 进行 GCD。很快就结束了，查看每个 divisor[n]里面的因数的位数。发现有 1540+256+256、256+256+256+1284、1796+256、256+256+256+256+256+768、256+256+256+256+1024 几种。但都没有完全被分解为 256 的。

随便挑了几个 768、1024 位的质数去 factordb 查，也没查到，也不知道它们是不是质数（但其实是可以检验的：计算所有(p-1)的乘积得到 phi，然后看一下是否满足 r==pow(r, e*invert(e, phi), n)，其中 r 要随机多选几个，n 以内就行，如果全都满足那么基本上可以确定它是质数，具体证明还没研究，也不知道

对不对)。猜想是因为只用了前 100 个 n，和后面的数一起 GCD 的话应该还能得到很多公因子。FILE_NUM 先设为 15932 跑了一下，实在太慢，觉得这个 recursive_gcd 其实可以优化，前面得到的 256 位数可以直接试除后面的大数。

然后写了个新算法，代码：

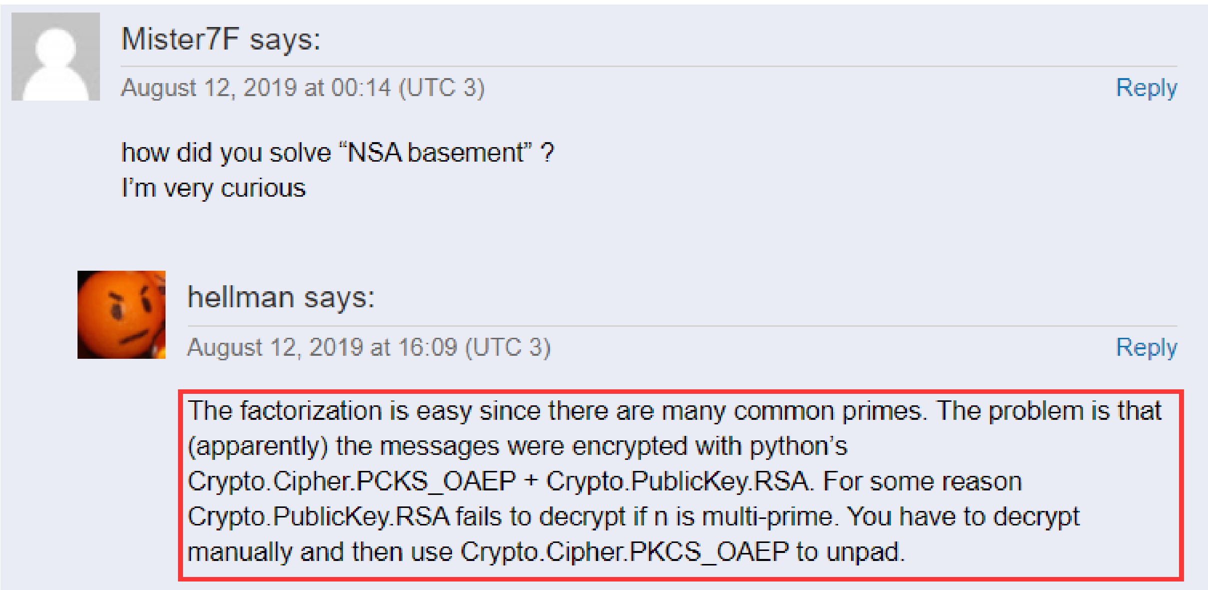
```
set256 = set()
setbig = set(sessions_n)
def new_gcd(set256, setbig):
    while True:
        width = bit_len_list(list(setbig))
        print("len(set256) =", len(set256), "\tlen(setbig) =", len(setbi
g), "now: ", get_now(),
              "\t", width, "bits")
        newset256 = set().union(set256)
        newsetbig = set().union(setbig)
        for p in set256:
            for big in setbig:
                if big not in newsetbig:
                    continue
                if 0 != big % p:
                    continue
                new_big = full_divide(big, p)
                link(big, p)
                link(big, new_big)
                for root in ancestor[p]:
                    divs = divisor[root]
                    if 7 <= len(divs):
                        f = open(get_now() + '_divisors.txt', 'w')
                        print('root =', root, '\n\t', )
                        for d in divs:
                            print(hex(d), end=',\n\t', file=f)
                        print()
                        f.close()
                newsetbig.remove(big)
                if 300 < bit_len(new_big):
                    newsetbig.add(new_big)
                else:
                    newset256.add(new_big)
            newset256.remove(p)
        set256 = set().union(newset256)
        setbig = set().union(newsetbig)

        print("Start gcd ", "now: ", get_now())
        finish_num = (width * 25 // 2048) + 2
        for i in sorted(list(setbig), reverse=True):
            for j in sorted(list(setbig), reverse=True):
```

```
            if i == j:
                continue
            if bit_len(i) < 300 or bit_len(i) < 300:
                continue
            d = int(gmpy2.gcd(i, j))
            if d != 1:
                if bit_len(d) < 300:
                    set256.add(d)
                else:
                    setbig.add(d)
        if len(set256) >= finish_num:
            break
    if 0 == len(set256):
        break
```

目的就是通过 GCD 找到一个能分解为 8 个 256 位数的 n。这个跑得也特别慢，好像是跑了十几个小时，但是依然没看到全分解的，也没有 6*256+512 的，最好的结果就是 5*256+768 了，这和前面那个 recursive_gcd 相比好像没什么进步。

对于一个 n，计算所有(p-1)的乘积得到 phi，然后看一下是否满足 r==pow(r, e*invert(e, phi), n)。试了几个发现 5*256+768 的 n 其实是满足的，那就先当作它只有 6 个因数吧，那意思就是 n 有多个质因子呗（所以 NSA 指的是这个意思吗）。然后 phi 和 d 都容易计算。但是发现直接把密文读进来的 c，用各自的 n 和计算得到的 d 计算 pow(c, d, n)得到的结果是不一样的，而且 long_to_bytes 之后也并不是可读的字节。

在这个链接（http://mslc.ctf.su/wp/1st-crypto-ctf-2019-least-solved-challenges/）得到了一点 hint。


Mister7F says:
August 12, 2019 at 00:14 (UTC 3) Reply

how did you solve “NSA basement” ?
I’m very curious

hellman says:
August 12, 2019 at 16:09 (UTC 3) Reply

The factorization is easy since there are many common primes. The problem is that (apparently) the messages were encrypted with python’s Crypto.Cipher.PCKS_OAEP + Crypto.PublicKey.RSA. For some reason Crypto.PublicKey.RSA fails to decrypt if n is multi-prime. You have to decrypt manually and then use Crypto.Cipher.PCKS_OAEP to unpad.

好的估计是 padding 的问题，那就去翻一下这个东西吧。其实如果直接搜索 rsa padding 可以看到主要有三种方式：RSA_PKCS1_PADDING， RSA_PKCS1_OAEP_PADDING， RSA_NO_PADDING。然后并没有找到单独 padding 的现成代码，还去翻了 rfc 文件（https://www.ietf.org/rfc/rfc3447.txt），差点就自己写了个 padding。

看了一下 Crypto.Cipher.PKCS1_OAEP 的用法（https://pythonhosted.org/pycrypto/Crypto.Cipher.PKCS1_OAEP-module.html），想了想可以算出 d 来之后放到 RSA 对象中传进 PKCS1_OAEP 就行了。代码：

```
with open('stuff\\enc\\flag_' + num_str + '.enc', 'rb') as f:
```

```
    c_bytes = f.read()
key = RSA.construct((n, e, d))
cipher = PKCS1_OAEP.new(key)
m = cipher.decrypt(c_bytes)
```

但是这样会报错。

```
  File "[redacted]\Crypto\Cipher\PKCS1_OAEP.py", line 227, in decrypt
    raise ValueError("Incorrect decryption.")
ValueError: Incorrect decryption.
```

报错的原因是解出来之后格式不对。

以下代码来自 Crypto\Cipher\PKCS1_OAEP.py，是解密的函数。

```
def decrypt(self, ct):
    """Decrypt a PKCS#1 OAEP ciphertext.

    This function is named ``RSAES-OAEP-DECRYPT``, and is specified in
    section 7.1.2 of RFC3447.

    :Parameters:
     ct : string
            The ciphertext that contains the message to recover.

    :Return: A string, the original message.
    :Raise ValueError:
        If the ciphertext length is incorrect, or if the decryption does n
ot
        succeed.
    :Raise TypeError:
        If the RSA key has no private half.
    """
    # TODO: Verify the key is RSA

    # See 7.1.2 in RFC3447
    modBits = Crypto.Util.number.size(self._key.n)
    k = ceil_div(modBits,8) # Convert from bits to bytes
    hLen = self._hashObj.digest_size

    # Step 1b and 1c
    if len(ct) != k or k<hLen+2:
        raise ValueError("Ciphertext with incorrect length.")
    # Step 2a (O2SIP), 2b (RSADP), and part of 2c (I2OSP)
    m = self._key.decrypt(ct)
    # Complete step 2c (I2OSP)
    em = bchr(0x00)*(k-len(m)) + m
    # Step 3a
    lHash = self._hashObj.new(self._label).digest()
    # Step 3b
```

```python
    y = em[0]
    # y must be 0, but we MUST NOT check it here in order not to
    # allow attacks like Manger's (http://dl.acm.org/citation.cfm?id=7041
43)
    maskedSeed = em[1:hLen+1]
    maskedDB = em[hLen+1:]
    # Step 3c
    seedMask = self._mgf(maskedDB, hLen)
    # Step 3d
    seed = strxor(maskedSeed, seedMask)
    # Step 3e
    dbMask = self._mgf(seed, k-hLen-1)
    # Step 3f
    db = strxor(maskedDB, dbMask)
    # Step 3g
    valid = 1
    one = db[hLen:].find(bchr(0x01))
    lHash1 = db[:hLen]
    if lHash1!=lHash:
        valid = 0
    if one<0:
        valid = 0
    if bord(y)!=0:
        valid = 0
    if not valid:
        raise ValueError("Incorrect decryption.")
    # Step 4
    return db[hLen+one+1:]
```

其中也提到了 RFC3447，具体流程还是看那个文件比较清晰。

然后其中解密的流程大概就是，padding-RSA-padding，encrypt 好像也差不多，除了 RSA 部分似乎是对称的。然后其中 RSA 的实现是在这一句：

```python
m = self._key.decrypt(ct)
```

暂且不看这个函数的代码，在构建 RSA 密钥的时候，如果我们传入 d=1，即：

```python
key = RSA.construct((n, e, 1))
```

那么报错内容为：

```
  File "[redacted]\Crypto\PublicKey\_slowmath.py", line 132, in rsa_construct
    raise ValueError("Unable to compute factors p and q from exponent d.")
ValueError: Unable to compute factors p and q from exponent d.
```

“ValueError: Unable to compute factors p and q from exponent d.”。也就是说它会根据 d 算出两个质因数 p 和 q，这对于普通 RSA 当然是完全正确的，但是我们这里的 n 有多于 2 个质因子。在这个“_slowmath.py”中可以看到这样一段注释：

```python
        # Compute factors p and q from the private exponent d.
        # We assume that n has no more than two factors.
        # See 8.2.2(i) in Handbook of Applied Cryptography.
```

“We assume that n has no more than two factors.”

所以在 RSA.construct 函数我们传入 n、e、d 的时候，它会把 n 分解，但分解的结果是不可能正确的，所以 RSA 解密这一步会有问题，最终导致了格式不对，Incorrect decrypton。

然后我把 PKCS1_OAEP 的解密函数复制下来进行了一些修改。主要修改包括：把中间 RSA 那一步换掉；把其中带有“self.”的变量改掉；某些相关变量按照执行“PKCS1_OAEP.new(key)”时进行设置。保证执行流程除了 RSA 那步都没有改变。

```
_hashObj = Crypto.Hash.SHA
_mgf = lambda x,y: Crypto.Signature.PKCS1_PSS.MGF1(x,y, _hashObj)
def decrypt_fromPKCS(ct, d, n):
    # See 7.1.2 in RFC3447
    modBits = Crypto.Util.number.size(n)
    k = ceil_div(modBits, 8)  # Convert from bits to bytes
    hLen = _hashObj.digest_size

    # Step 1b and 1c
    if len(ct) != k or k < hLen + 2:
        raise ValueError("Ciphertext with incorrect length.")
    # Step 2a (O2SIP), 2b (RSADP), and part of 2c (I2OSP)
    # m = self._key.decrypt(ct)
    m = long_to_bytes(pow(bytes_to_long(ct), d, n))
    # Complete step 2c (I2OSP)
    em = bchr(0x00) * (k - len(m)) + m
    # Step 3a
    lHash = _hashObj.new('').digest()
    # Step 3b
    y = em[0]
    # y must be 0, but we MUST NOT check it here in order not to
    # allow attacks like Manger's
(http://dl.acm.org/citation.cfm?id=704143)
    maskedSeed = em[1:hLen + 1]
    maskedDB = em[hLen + 1:]
    # Step 3c
    seedMask = _mgf(maskedDB, hLen)
    # Step 3d
    seed = strxor(maskedSeed, seedMask)
    # Step 3e
    dbMask = _mgf(seed, k - hLen - 1)
    # Step 3f
    db = strxor(maskedDB, dbMask)
    # Step 3g
    valid = 1
    one = db[hLen:].find(bchr(0x01))
    lHash1 = db[:hLen]
    if lHash1 != lHash:
        valid = 0
```

```
    if one < 0:
        valid = 0
    if bord(y) != 0:
        valid = 0
    if not valid:
        raise ValueError("Incorrect decryption.")
    # Step 4
    return db[hLen + one + 1:]
```

然后选了两个 n 被分解成 5*256+768 的密文，解密。

```
factors_97 = [
    0x9924a29bc79f3cda657327b37b96c679542ffa9aa5193ac447d9d320e0c94faf,
    0xce1a2b1f6f9baa2ab43c796c7ca14113aace4a02e6e31ecd97cbb9471b700ef7,
    0xe1e1d6e65c575bb597040a83d85f193ee4a35fff6edc3ef300cf566201e76413,

0x2f16205d466405b4d631b3c5e177dce85d62d09482a65234707b76f8e29220d1181fed9a9343f9359d4bc1692c1b5f13cb4000873db2a8c8b7381ce3a8656d77734f67314ed70c987fc41376560e674133f465afda1f9ea42b5a44117866d051d,
    0x94c21c264f65fd298bad48e528bd882d8d4cd7fb436739ad7585178c6c204d95,
    0xbb5794796906fa730aed54e02880f9092e6d488bfc8f020acf0cb3d60446d88f,
]
factors_87 = [
    0xe5b4b1b65784b253f37e2677033aaebebdcf5eef0c52f4845c240e8a19aa91f3,
    0xe8ced45445808b9486982387 97a76b92c36a6490b22c260def7403963b6f0fe3,

0x3bebbf0541192edc741ccb522d6888ee0701ecb5f2723b3cd59ffd09fc385a51f4ebb0af83c934362778b5c7ac3df8a0157bdb0b0d3b72d7ef4cadae5f4bcb99f8d6af4da4172d2df55a4a328a2f9ec133fad44c6af0f8845a75227b40ff63809,
    0xccbccc3ac19ec7b472311f3c9fe1acad2196093ba196104c2ad894818e1eca49,
    0xf7deac53ee40907cd4b186d88d04e96d4740a0e37678d4abffb405574ad9de21,
    0xfb6b690d83c20cc08fd16bcaa9277e412b7b855f6b7e59b40386d08a5589d7ab,
]


def decrypt(num_str, factors):
    with open('stuff\\keys\\pubkey_' + num_str + '.pem', 'r') as f:
        pubkey = RSA.importKey(f.read())

    n = getattr(pubkey, 'n')
    e = getattr(pubkey, 'e')
    print('e =', e)
    i = 1
    for p in factors:
        i *= p
    assert i == n
```

```python
    with open('stuff\\enc\\flag_' + num_str + '.enc', 'rb') as f:
        c_bytes = f.read()
        c = bytes_to_long(c_bytes)
    print("n:", bit_len(n), " bits")
    print("c:", bit_len(c), " bits")
    print("c =", hex(c))

    phi = 1
    for p in factors:
        phi *= (p - 1)

    d = int(gmpy2.invert(e, phi))
    for _ in range(10):
        test_m = random.randint(2, n - 1)
        assert test_m == (pow(test_m, e * d, n))

    # key = RSA.construct((n, e, d))
    # cipher = PKCS1_OAEP.new(key)
    # m = cipher.decrypt(c_bytes)
    m = decrypt_fromPKCS(c_bytes, d, n)
    print(m)


decrypt('00097', factors_97)
decrypt('00087', factors_87)
```

可以正确得到 FLAG。

总结：
有“too many public keys”的时候尝试交叉 GCD；
Padding；
多素数；
python RSA 的密钥构建细节。