[HKUST Firebird CTF 2024]
Random Secure Algorithm (I)

Description:

According to chall-1.py, we can find that:

```
e = 0x10001 # given
rsa.n # is given
secret = [os.urandom(8) for _ in range(10)] # not given
iv = os.urandom(16) # is given
enc_flag = iv + AES.new(key =
hashlib.sha256(b''.join(secret)).digest(), mode = AES.MODE_CBC, iv =
iv).encrypt(pad(FLAG, 16))
# encrypted flag is given
enc_secret = []
for block in secret:
    enc_secret.append(rsa.encrypt(int.from_bytes(block, 'big')))
enc_s = ':'.join(hex(c)[2:].rjust(1024 // 8 * 2, '0') for c in
enc_secret)
# 10 encrypted secrets are given
```

AND:

```
def dot_mod(c1, c2, n):
    assert len(c1) == len(c2)
    return sum(i * j for i, j in zip(c1, c2)) % n

ct = input('🔒 ').strip().replace('-', '').split(':')
pt = [rsa.decrypt(int(c, 16)) for c in ct]

weights = list(range(1, 11))
random.shuffle(weights)

res = dot_mod(pt, weights, rsa.n)
print(f'📄 {hex(res)[2:]}')
```

Which means we can perform chosen ciphertext attack as we are allowed to decrypt our ciphertext and receive the plaintext.

However, we need to send exactly 10 ciphertexts and they would be decrypted then multiplied by random weights which is from 1 to 10, and added together then modulo n

Solution:

Our target is to find the 10 secret keys from cracking this challenge's RSA cryptosystem.

Let's take a look at `enc_secret` and the decryption first, we can get these formula:

$$\varepsilon(s) = [{s_1}^e, {s_2}^e, {s_3}^e, \dots, {s_{10}}^e]\ (mod\ n)$$

$$res = sum([{c_1}^d, {c_2}^d, {c_3}^d, \dots, {c_{10}}^d](mod\ n) * RandOrder([1,2,3, \dots,10]))(mod\ n)$$

It is obvious that we can easily apply **Fermat's little theorem** with chosen ciphertext attack, let's see what happen if we send `enc_secret` directly:

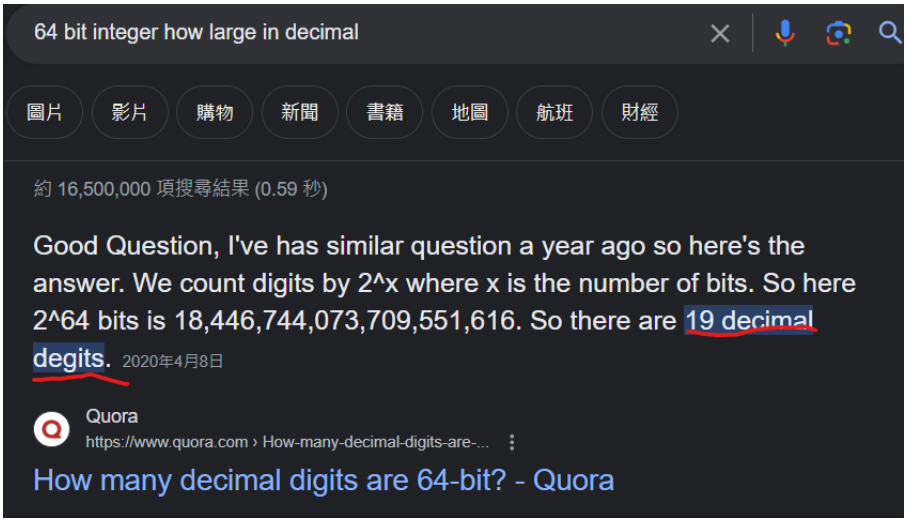
$$sent = [{s_1}^e, {s_2}^e, {s_3}^e, \dots, {s_{10}}^e](mod\ n)$$

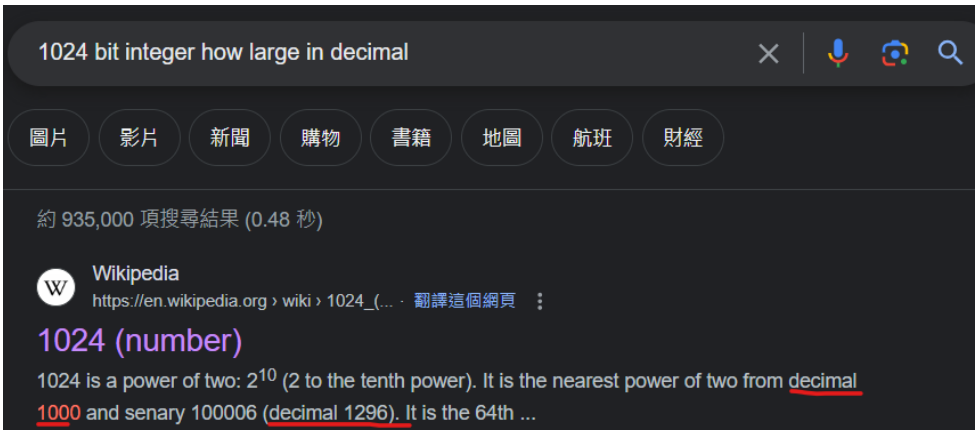
$$res = sum([{s_1}^{ed}, {s_2}^{ed}, {s_3}^{ed}, \dots, {s_{10}}^{ed}](mod\ n) * weights)(mod\ n)$$
$$\rightarrow res = sum([s_1, s_2, s_3, \dots, s_{10}](mod\ n) * weights)(mod\ n)$$

However, the secret keys are mixed up since they are added together.

After looking more carefully, you should also find that n is multiplied by 2 512 bits primes, and secret keys are 8 bytes large (64 bits), and value of weights are not larger than 10. Which means we can multiply the secret keys with different weights we picked and then send them together, we should get res which the secret keys split by many zeros if the sum of product is not larger than n !!!

BTW this is how could I get this idea:





Again, we also need to apply **Fermat's little theorem** on the weights we picked to bypass, this time I chose 10^30*(count-1) which count stands for $count^{th}$ secret key as weights, which the 10^30 is much larger than each value of secret key to do significantly splitting and the worst case (10^270 * 2^64 * 10 << 2^(1024-1))) is still much smaller than n so won't be filtered by mod n.

```
heheboi = []
count = 1
for block in enc_s:
    heheboi.append(pow(count,e,n)*block)
    count *= 10**30
tosend = ':'.join(hex(c)[2:].rjust(1024 // 8 * 2, '0') for c in heheboi)
io.sendlineafter('🔒 '.encode(),tosend.encode())
```

$$sent = [{s_1}^e, 10^{30*e}{s_2}^e, 10^{30*2e}{s_3}^e, \dots, 10^{270e}{s_{10}}^e](mod\ n)$$

Here is the result of res:

$$res = sum([s_1, 10^{30}s_2, 10^{30*2}s_3, \dots, 10^{270}s_{10}](mod\ n) * weights)(mod\ n)$$

```
145548104578389913600000000000062087102212529102406000000000065587664165463898668000000000139837815107094851901000000000027349169177680406408000000000030431031980520075676000000
0002365019097971457396600000000000404919739510915767700000000001344919219825655519440000000000018408035021122918 24
```

Now we have the 10 secret keys multiplied by weights (RandOrder([1 to 10])) on hand, so what to do next is to match each secret key to each item in enc_secret, and then do aes.decrypt the flag with cracked key and given iv and given enc_flag (I did it manually as I m lazy to split the key by writing more codes :P)

Solve Scripts:

```python
from pwn import *
e = 0x10001
io = remote('ash-chal.firebird.sh', int(36008))
io.recvuntil('🏁 '.encode())
enc_flag = int(io.recvline().decode().strip(),16)
io.recvuntil('📣 '.encode())
n = int(io.recvline().decode().strip(),16)
io.recvuntil('🔧 '.encode())
enc_s = [int(i,16) for i in io.recvline().decode().strip().split(':')]

heheboi = []
count = 1
for block in enc_s:
    heheboi.append(pow(count,e,n)*block)
    count *= 10**30
tosend = ':'.join(hex(c)[2:].rjust(1024 // 8 * 2, '0') for c in heheboi)
io.sendlineafter('🔒 '.encode(),tosend.encode())
io.recvuntil('📜 '.encode())
what7usaid = io.recvline().decode().strip()
print(int(what7usaid,16))
print(hex(n)[2:])
print("flag: ",enc_flag)
print("enc_s: ",enc_s)
```

```python
from Crypto.Util.number import getPrime
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
import signal
import hashlib
import os


x1 = 14554810457838991360
x2 = 62087102212529102406
x3 = 65587664165463898668
x4 = 139837815107094851901
x5 = 27349169177680406408
x6 = 30431031980520075676
x7 = 23650190979714573966
```

```
x8 = 40491973951091576770
x9 = 134491921982565551944
x10 = 1840803502112291824
x_list = [x10,x9,x8,x7,x6,x5,x4,x3,x2,x1]
e = 0x10001
n =
0xbf1b0cbd82cb9b199733d4eb6a2291dfc5d38925c19c7312ac4a640d0dc653d5fba0e46a6a6f07974d1de3
1b1ff71e22d026dc39c51e2c20fc64a84d32d46a44613aaef81ff8294cfb4faa93db0eae4f0ee6d25b101c68
ee012ae1cf605a2849f12a244742cfdac196ac9edd97aaa382f5c7f371464e571199c6d57fc96b4c81
iv =
int.to_bytes(int('59730648394cd0fae8db927d64b56c7929660996f77aeaf0501b9d7909226e0e350abd
74dd22b4e8dfd599a55ce87d609da4170d0ae769bf50708608d25c795c1be650b3a8233ccd5447442ed00965
5c'[:32],16),16,'big')
enc_s=  [576887213473525856428745187953981100200532984573341807829991735754397578646524940
401080759719298067219997754255547073943148503813408599080815865294790074662604471580581697
978344117785716880761846291526017102780716308191273624794768836165929435832942711033097383
836483820440820278714412101308204871755190380972010922,
409384446145016238322353941945360977893264493637126487877524986763998335533163783581140107
079368869985020845210947809278885255155873905110334741492264357450937188273697087193731627
273492574581588098340713439790310842361834916237947147302838046851637783016559436064960625
253403146294088118281208197507895287968428 36,
321260488534818554976248532001168703276555210873777938710274015364102405517720307860514427
774535716844924542109655985998297283485188955927409460985046616509123615620693355054296005
193018194514022088041436358581505555396193631791394131746063989481951655761929863790693
371602755297159502545428225155205663324580,
289434269529258412825679175462440594449306276314561216142502837257416378712587465508242393
790323156396406908317001910874780226679425561057242801802556571276898990091267014421310282
836321406837325044712927307571283877979751710878675805574737538299931143922555751973537
296625871461418040303528469426360588475 99,
914565013871882466399248983040523490988701314189021486128010631168161179639103314153355361
025370727799905014690051226008250150898528497404316198184995113120652613197447174892682280
419808983152772606881664100699466652935703144015586350185811404276444202823005631732491
151493759551050277005482808102348506735 58,
834292659847595652823747097597333034162043089884032368713451615991505380601891730837449298
252452843267697386411273744994823111417122855670568656593775518654853821325314029196132318
170476058252082929828976546370633122172695511626949856990238861101105488181869811121328
627527266494646901256081419282481524123 62,
248037484762763315435221762396292412195819661860747687664041993643342908653021095335288994
482714788371861078039707975122089701568845828877842796774094602203865566732212832723897538
580181971349835206980545246968570709280999132518477475015434073001247075097716144966416392
707785028875835363983068323394459621 52,
761044898357626298028710943175416927421862185955873608629344888352839806213214489260622176
```

```
156472881101493321823606199150451635636379676370379637490415876032919319140989239535137731945051552273222218839539541239679889960665888657389561471709953434713238075227935575735239315837404002814432300639945555254806388 0,
169395695871801649075999760069160976830342261511383253345388005421781270079665223142996103219331269552166860793377545899643104339684357958780627590593877913827813903300001457181203869557822824047537568380069136672499048341546169696760642291929246772254114783683419931402209964444375920716294225937497214473930,
488052954540089191315737987446669717396874450650306380369355589419748763091002728303951935693528336940760754057165395431053958458505461892272799537092080280834325889157548135775863237377749936722295970629229595488778437695457716069537664723963942157461918525139744013988565677964892754201869336802187952224680]
weights = list(range(1,11))[::-1]
print(weights)
secret = [0,0,0,0,0,0,0,0,0,0]

def check(item):
    for i in weights:
        if item % i == 0 and pow((item // i),e,n) in enc_s:
            print("ok! ", i)
            secret[enc_s.index(pow((item // i),e,n))] = int.to_bytes(item//i,8,'big')
            return
    print("bruh")
    return
for x in x_list:
    check(x)
the_flag = AES.new(key = hashlib.sha256(b''.join(secret)).digest(), mode = AES.MODE_CBC, iv =
iv).decrypt(int.to_bytes(int('59730648394cd0fae8db927d64b56c7929660996f77aeaf0501b9d7909226e0e350abd74dd22b4e8dfd599a55ce87d609da4170d0ae769bf50708608d25c795c1be650b3a8233ccd5447442ed009655c'[32:],16),64,'big'))
print(the_flag)
```

Thank you for watching!