Modular_Encryption1 writeup

Info challenge

Context

[FR]

L'arithmétique modulaire est au cœur de nombreuses techniques cryptographiques, notamment dans les algorithmes de chiffrement et de signature numérique. Elle permet de travailler avec des résidus de division, simplifiant ainsi les calculs complexes sur de grands nombres.
Maîtriser cette branche des mathématiques est essentiel pour comprendre et appliquer des méthodes de cryptographie sécurisées. Et en bon hacker l'arithmétique est un jeu pour vous, n'est-ce pas :) ?. Trouvez le Flag.
Bonne Réflexion.

[EN]

Modular arithmetic is at the heart of many cryptographic techniques, particularly encryption and digital signature algorithms. It makes it possible to work with division residues, simplifying complex calculations on large numbers. Mastering this branch of mathematics is essential to understanding and applying secure cryptographic methods.
And as a good hacker, arithmetic is a game for you, isn't it :) ?. Find the Flag.
Good thinking.

Author: `h1s0k4`

Catégorie: Cryptographie

Points: 500

Méthodologie de solve

output.txt

```
n1 = 146020941211024523247996231798612150946538898260780187478
n2 = 112802478032416783969806265651772823106256344591369382400
s = [82307081786160191830039031659863275303796978633298707661
w1 = 114169768790989127660179905317133317526247199731730905738
w2 = 127107983881093850500626668328501111346826711676584086911
z2 = 109990909833507224362850690143580918339166504444487261530
z3 = 37404128569367453183605356992315188248807573391557788041
enc_flag = c26aa70d1812e652c188119a699b6d424d9a20706322f323de
#https://teamrocketist.github.io/2019/03/31/Crypto-VolgaCtf201
```

chall.py

```
from Crypto.Util.number import getPrime, long_to_bytes, inver
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
from hashlib import sha256
from gmpy2 import next_prime
from secret import FLAG

class TRNG:

    def __init__(self, seed):
        self.e = 0x10001
        self.s = seed
        self.r = getPrime(768)
        self.k = getPrime(1024)
        self.l = getPrime(1024)
        self.w1 = getPrime(1024)
        self.w2 = getPrime(1024)
        self.n1 = self.k * self.l
```

```
        while True:
            self.p, self.q = getPrime(768), getPrime(768)
            if self.p < self.r and self.q < self.r:
                break
        self.n2 = self.p * self.q * self.r
        phi = (self.p - 1) * (self.q - 1) * (self.r - 1)
        self.d = inverse(self.e, phi)
        self.z1 = pow((4*self.k+3*self.l),self.w1,n1)
        self.z2 = pow((self.k-2*self.l),self.w2,n1)
        self.z3 = next_prime(self.z1)

    def next(self):
        self.s = (self.s * self.p + self.q) % self.r
        return self.s

def main():
    trng = TRNG(bytes_to_long(random.randbytes(26)))
    trns = [trng.next() for _ in range(5)]

    key = sha256(long_to_bytes(trng.d)).digest()
    cipher = AES.new(key, AES.MODE_ECB)
    enc_flag = cipher.encrypt(pad(long_to_bytes(pow(bytes_to_
    16)).hex()

    with open('output.txt', 'w') as f:
        f.write(f'n1 = {trng.n1}\n')
        f.write(f'n2 = {trng.n2}\n')
        f.write(f's = {trns}\n')
        f.write(f'w1 = {trng.w1}\n')
        f.write(f'w2 = {trng.w2}\n')
        f.write(f'z2 = {trng.z2}\n')
        f.write(f'z3 = {trng.z3}\n')
        f.write(f'enc_flag = {enc_flag}\n')


if __name__ == "__main__":
    main()
```

Ce challenge comporte 2 principaux noeuds, une partie **LCG** et une partie **RSA.**

Tout d'abord, décryptons le code de chiffrement

Nous avons la classe `TRNG` qui se charge de générer un nombre aléatoire à partir de la méthode `LCG`. Ainsi, à chaque appel de la fonction next, un nouveau nombre aléatoire est généré à partir des paramètres `p`, `q`, `r` et `s`. Ce procédé de génération se base un peu sur le principe des suites numériques ou il faut connaître le résultat précédent pour calculer le suivant.

Dans le `main`, la classe `TRNG` est instanciée à partir d'un nombre aléatoire de 26 bytes qui est le seed et en même temps notre `s0` dans la suite.

Ensuite, une série de 5 nombres est générée avec la fonction next de la classe pour donner des valeurs `s1`, `s2`, `s3`, `s4` et `s5`.

```
def main():
    trng = TRNG(bytes_to_long(random.randbytes(26)))
    trns = [trng.next() for _ in range(5)]
```

Après, une clé de chiffrement est généré à partir du hash de `d`

`d` étant une clé de déchiffrement RSA.

```
self.n2 = self.p * self.q * self.r
phi = (self.p - 1) * (self.q - 1) * (self.r - 1)
self.d = inverse(self.e, phi)
```

La clé de chiffrement `key` est par la suite utilisée pour encrypter `pow(bytes_to_long(FLAG),e,n1)`

Avec l'algorithme AES CBC

```
key = sha256(long_to_bytes(trng.d)).digest()
cipher = AES.new(key, AES.MODE_ECB)
enc_flag = cipher.encrypt(pad(long_to_bytes(pow(bytes_to_long
16)).hex()
```

Enfin, il nous est donné les valeurs `n1`, `n2`, `s`, `w1`, `w2`, `z2`, `z3` et `enc_flag` dans le fichier output.txt:

```
with open('output.txt', 'w') as f:
     f.write(f'n1 = {trng.n1}\n')
     f.write(f'n2 = {trng.n2}\n')
     f.write(f's = {trns}\n')
     f.write(f'w1 = {trng.w1}\n')
     f.write(f'w2 = {trng.w2}\n')
     f.write(f'z2 = {trng.z2}\n')
     f.write(f'z3 = {trng.z3}\n')
     f.write(f'enc_flag = {enc_flag}\n')
```

Pour résoudre ce challenge, il nous faut en premier lieu déterminer `d` pour déchiffré `pow(bytes_to_long(FLAG),e,n1)), 16)` .

Pour trouver `d` il nous faut trouver `p` , `q`, et `r` . Pour cela, nous allons utilisé les valeurs généré par le LCG.

Du code ci-dessus, nous pouvons déjà établir ces suites d'équations:

```
s0 = seed
s1 = s0 * p + q [r]
s2 = s1 * p + q [r]
s3 = s2 * p + q [r]
s4 = s3 * p + q [r]
s5 = s4 * p + q [r]
```

```
s2 - s1 = (s1 - s0) * p [r]
s3 - s2 = (s2 - s1) * p [r]
s4 - s3 = (s3 - s2) * p [r]
s5 - s4 = (s4 - s3) * p [r]
```

```
(s5 - s4)(s2 - s1)p = (s3 - s2)(s4 - s3)p [r]
p[(s5 - s4)(s2 - s1) - (s3 - s2)(s4 - s3)]= 0 [r]
```

`r` ne divisant pas `p` donc `(s5 - s4)(s2 - s1) - (s3 - s2)(s4 - s3)` est un multiple de `r` .

Ainsi

```
gcd((s5 - s4)(s2 - s1) - (s3 - s2)(s4 - s3), n2) = r
```

Car `n2 = p * q * r` les trois facteurs étant des nombres premiers.

Ainsi avec ce code, on arrive à déterminer `pow(bytes_to_long(FLAG),e,n1)), 16)`

```
from math import gcd
from Crypto.Util.number import getPrime, long_to_bytes, inver
from hashlib import sha256
from gmpy2 import next_prime
from Crypto.Cipher import AES
from binascii import unhexlify


n1 = 146020941211024523247996231798612150946538898260780187478
n2 = 112802478032416783969806265651772823106256344591369382400
s = [82307081786160191830039031659863275303796978633298707661
w1 = 114169768790989127660179905317133317526247199731730905738
w2 = 12710798388109385050062666832850111134682671167658408691
z2 = 10999090983350722436285069014358091833916650444448726153
z3 = 37404128569367453183605356992315188248807573391557788041
enc_flag = "c26aa70d1812e652c188119a699b6d424d9a20706322f323d


r = gcd(((s[4] - s[3]) * (s[1] - s[0])) - ((s[2] - s[1]) * (s
print(r)

p = (s[4] - s[3]) * inverse(s[3] - s[2], r) % r
q = n2 // (p * r)
e = 0x10001
```

```
phi = (p - 1) * (q - 1) * (r - 1)
d = inverse(e, phi)

key = sha256(long_to_bytes(d)).digest()
cipher = AES.new(key, AES.MODE_ECB)
dec_flag = cipher.decrypt(unhexlify(enc_flag))
```

La deuxième phase consiste à déterminer `FLAG` . Pour cela, il nous faut déterminer `k` et `l` .

Nous allons nous servir ici de z1 et z2

```
self.z1 = pow((4*self.k+3*self.l),self.w1,n1)
self.z2 = pow((self.k-2*self.l),self.w2,n1)
```

le premier soucis est que on ne connais pas z2 mais on sait que:

```
self.z3 = next_prime(self.z1)
```

Ainsi `z3` est le nombre premier qui suit immédiatement `z1` on peut donc brute forcer sa valeur en utilisant cette fonction:

```
def find_z1_value():
    z1 = []
    i = 1
    while not isPrime(z3 - i):
        z1.append(z3 - i)
        i = i + 1
    return z1
```

Ensuite pour chaque valeur de z1 nous allons essayer de résoudre le **Modular Binomials problem** entre z1 et z2.

```
z1_vals = find_z1_value()
z1 = 0
```

```
for z in z1_vals:
    print(f"index-{j}")
    D = -(pow(z,w2, n1) * pow(2, w1w2, n1)) - (pow(z2, w1, n1
    k = gcd(D, n1)
    l = n1 // k
    if l * k == n1 and k != 1:
            z1 = k
            break
    j = j + 1
```

Et enfin on peut déduire le flag:

```
ct = bytes_to_long(dec_flag)
phi = (l - 1) * (k - 1)
d = inverse(e, phi)
print(long_to_bytes(pow(ct, d, n1)))
```