PWNME - Like a Whispering Entropy

ValekoZ

Description

> I don't trust those who wrote crypto implementations, so I had to write my own... I created everything from scratch so I know it is safe ! No one backdoored my Diffie-Hellman implementation ! Btw, I gave you an output of a connection to my service, but anyway you won't be able to get back the secret message

Code analysis

```
#!/bin/env sage

from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad

from Crypto.Util.number import getStrongPrime, long_to_bytes, bytes_to_long

import hashlib

from os import getenv, urandom
from random import getrandbits


FLAG = getenv("FLAG", "PWNME{dummyflag}").encode()


class PRNG(object):
    def __init__(self):
        self._seed = None
        self.q = 279622638356169037213136872013126932777
        self.n = 12
        self.A = Matrix(GF(self.q), 1, self.n, [
                19159356164385140466,
                19848194065535878410,
                33461959522325830456,
                12213590058439028697,
                35299014249932143965,
                13327781436808877193,
                20921178705527762622,
                9371898426952684667,
                9769023908222006322,
                28712160343104144896,
                32272228797175569095,
                14666990089233663894
            ])

        # LCG props
        self.a = getrandbits(32)
        self.c = getrandbits(32)
        self._lcgseed = getrandbits(32)
        self.mod = 2551431067

    @property
    def noise(self):
        self._lcgseed = (self.a * self._lcgseed + self.c) % self.mod

        return self._lcgseed

    @property
    def seed(self):
        if self._seed is None:
            self._seed = Matrix(GF(self.q), self.n, 1, [
                    getrandbits(102) for i in range(self.n)
                ])

        return self._seed

    def randint(self):
        b = (self.A * self.seed + self.noise)[0][0]

        self.A = Matrix(GF(self.q), 1, self.n, [
                int(x * b^(i+1)) % 2^65 for i, x in enumerate(self.A[0])
            ])

        return b



def encrypt(shared_secret: int, iv: bytes, msg: bytes):
    sha1 = hashlib.sha1()
    sha1.update(str(shared_secret).encode('ascii'))
    key = sha1.digest()[:16]

    cipher = AES.new(key, AES.MODE_CBC, iv)
    ciphertext = cipher.encrypt(pad(msg, 16))

    data = {}
    data['iv'] = iv.hex()
    data['encrypted_msg'] = ciphertext.hex()
    return data


def main():
    # Initialize PRNG:
    rng = PRNG()

    g = 2
    p = 155251809230070893513091813125848175563133404943451431320235119490296623994910210725

    n = 100

    ivs = [
        long_to_bytes(int(rng.randint())).ljust(16, b'\0') for _ in range(n)
    ]

    secret_keys = [
        int(rng.randint()) |
            int(rng.randint() << 128) |
            int(rng.randint() << 256) |
            int(rng.randint() << 384) |
            int(rng.randint() << 512) |
            int(rng.randint() << 640) for _ in range(n)
    ]

    public_keys = [
        pow(g, sk, p) for sk in secret_keys
    ]

    for i, e in enumerate(zip(ivs, secret_keys, public_keys)):
        iv, sk, pk = e
        print(f"Alice's public key: { pk }")

        pk2 = bytes_to_long(urandom(0x60))
        print(f"User #{i} public key: { pk2 }")

        shared_secret = pow(pk2, sk, p)
        print(f"{encrypt(shared_secret, iv, FLAG) = }")

        print("------------------------")


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

Here we see the main function, generating 100 ivs, then 100 secret keys, and then computing the cipher of the flag using different keys computed using a Diffie-Hellman algorithm.

Checking the constants used, we see that everything seems safe.

But then, analysing the PRNG algorithm, we can see that it seems pretty unusual.

Side note: The computation of the private keys are wrong because of type issues, I should have returned `int(b)` instead of just `b` in the `randint` function. Noticing that, we could have bypassed the whole challenge since the randomness of the private keys are really low

PRNG algorithm

The `randint` function is the following:

```
def randint(self):
    b = (self.A * self.seed + self.noise)[0][0]
    # ...
    return b
```

This looks like a LWE cryptosystem:

$$A \times s + e = b \mod q$$

Since all of this is done modulo q, which is big, and e (the noise), which is pretty small compared the other values, it might be possible to retrieve s without performing an exact **CVP**, having the values of A and b. This is referred as the "Hidden Number Problem", since we know the MSB of $A \times s$ and we want to retrieve s.

Since the first 100 random numbers generated are the IVs, we know them.

When we generate a new random number, we modify A in a deterministic manner. We can then compute the different values of A with:

```
nextA = [A[i] * b^(i+1) for i in range(12)]
```

LCG

Then, knowing all the values of A, s and b, we can compute the values of the noises and then deduce the missing parameters of the LCG algorithm.

Having lcg_1, lcg_2 and lcg_3 we have:

$$a = \frac{lcg_3 - lcg_2}{lcg_2 - lcg_1}$$

And then

$$c = lcg_3 - a \times lcg_2$$

Solving script

Since my script isn't so clean, here is the one used by remy_o during the competition:

```
# Reconstruction de seed

from sage.all import Matrix, ZZ, QQ, mod
from sage.matrix.berlekamp_massey import berlekamp_massey

B = [
    0x07dbf2ecf811a9eb23ca250dda08bfec,
    0x955543311a9d6ef1b088f92972d7c93b,
    0x90965f3b910c06d34bc18c761ddfee70,
    0xc93bde6abb33d8ee9a37465b9efa92ba,
    0x21a9215984fd9513654b880c89e3586a,
    0xaae7bf0d2678fb3ca606d1904cd5a5e7,
    0x0ef36ac5f5b69d9182b2e26400a2a93a,
    0x7008b5800ef4b18d3d95c6272c3bcd62,
    0x1d3811ee484ac900e004d6a5b6ec8693,
    0x1cd782e2a089127c41bee85467ae44c8,
    0x99dbd8974d66f6829ee82be9204ed874,
    0x9ebb68bb26ee609b1e83d39760f1ae72,
    0xcc067a2923a7b15b87634fe553b28478,
    0xa61579000c348474006c26a9d6d9c00b,
    0x687e5599bbc7a3663db5c418038bbe1e,
    0x8a3d64984b936919da65a64e98b12439,
    0xbc8a8eb4139eae18410447805b488d1b,
    0x1d773f7f32598c94d147149e450ea3ee,
]
B = B[:14]

p = 2551431067
```

```
q = 2796226383561690372131368720131269327777

A0 = [
        19159356164385140466,
        19848194065535878410,
        33461959522325830456,
        12213590058439028697,
        35299014249932143965,
        13327781436808877193,
        20921178705527762622,
        9371898426952684667,
        9769023908222006322,
        28712160343104144896,
        32272228797175569095,
        14666990089233663894
]

size = 12 + len(B) + 1
M = Matrix(ZZ, size, size)
for i in range(12):
    M[i,i] = 1
A = A0
for j, b in enumerate(B):
    # Equation somme(seedi Ai) = b + epsilon
    for i in range(12):
        M[i,12+j] = A[i]
    # Modulo q
    M[12+j,12+j] = q
    # Résultat (négatif décalé de 2^31)
    M[12 + len(B), 12+j] = -b - 2**31
    M[12 + len(B), 12 + len(B)] = 1
    A = [((a * pow(b, i+1, q)) % q) % 2**65 for i, a in enumerate(A)]

# 12 colonnes 2^102
# B colonnes 2^30
for i in range(len(B)):
    M.rescale_col(12 + i, 2**70)
# 1 colonne 1
M.rescale_col(12 + len(B), 2**102)

ML = M.LLL().change_ring(QQ)
for i in range(len(B)):
    ML.rescale_col(12 + i, 1/QQ(2**70))
ML.rescale_col(12 + len(B), 1/QQ(2**102))

r = []
```

```
for r in ML.rows():
    if abs(r[-1]) == 1:
        print(r)
        print([int(x).bit_length() for x in r])
        break

seed = []
for i in range(12):
    assert 0 < r[i] < q
    assert int(r[i]).bit_length() <= 102
    print(r[i])
    seed.append(int(r[i]))

A = A0
epss = []
for j, b in enumerate(B):
    epsilon = (b - sum(a*s for a,s in zip(A, seed))) % q
    A = [((a * pow(b, i+1, q)) % q) % 2**65 for i, a in enumerate(A)]
    print(epsilon)
    epss.append(mod(epsilon,p))

char = berlekamp_massey(epss)
print(char)
print(char.roots())
for a, _ in char.roots():
    if a != 1:
        break
for x, y in zip(epss, epss[1:]):
    print("c", (y - a * x) % p)
```