Twisted Entanglement

overview

Twisted Entanglement is an intermediate cryptography challenge.

The knowledge points required to complete this challenge lie in the elliptic curve algorithm and the random number generation mechanism in Python.

topic analysis

Associated mission files include `server.py` source `util.py` code and an online environment.

`server.py` The content is excerpted as follows

```python
from util import *

assert private_key < 8748541127929402731638

p =
115792089237316195423570985008687907853269984665640564039457584007908834671663
a = 0
b = 7
E = {"a": a, "b": b, "p": p}

def main(s):
    G = [
        0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798,
        0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8
    ]
    Q = multiply(private_key, G, E)
    sendMessage(s, f"Public Key: {Q}")

    while True:
        sendMessage(s, menu)
        try:
            option = receiveMessage(s, "\n> ")
            if option == "1":
                user_point = receiveMessage(s, "\nEnter your point x,y: ")
                point = parseUserPoint(user_point)
                public_key = multiply(private_key, point, E)
                sendMessage(s, f"\nHere's your new Public Key: {public_key}")
            elif option == "2":
                user_basis = receiveMessage(
                    s, "\nChoose your 256 basis for the KEP: ")

                basis = parseUserBasis(user_basis)
                q_server_key, q_user_key = generateKeys(basis, private_key)
                ciphertext = encrypt(FLAG, q_server_key)

                sendMessage(s, f"\nThe Quantum key: {q_user_key}")
                sendMessage(s, f"\nFlag Encrypted: {ciphertext.hex()}")

            elif option == "3":
                sendMessage(s, "\nQuantum Goodbye!")
                break
            else:
                sendMessage(s, "\nInvalid option!")
        except Exception as e:
            sendMessage(s, f"\nOoops! something strange happen X__X: {e}")
```

`util.py` The content is excerpted as follows

```python
import netsquid as ns
from random import seed, randint
from hashlib import sha256
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad

O = "Origin"

ns.sim_reset()


def eea(r0, r1):
    if r0 == 0:
        return (r1, 0, 1)
    else:
        g, s, t = eea(r1 % r0, r0)
        return (g, t - (r1 // r0) * s, s)


def add(P, Q, a, m):
    if (P == O):
        return Q
    elif (Q == O):
        return P
    elif ((P[0] == Q[0]) and (P[1] == m - Q[1])):
        return O
    else:
        if (P[0] == Q[0] and P[1] == Q[1]):
            S = ((3 * (pow(P[0], 2)) + a) * eea(2 * P[1], m)[1]) % m
        else:
            S = ((Q[1] - P[1]) * eea((Q[0] - P[0]) % m, m)[1]) % m
        x3 = (pow(S, 2) - P[0] - Q[0]) % m
        y3 = (S * (P[0] - x3) - P[1]) % m
        Q[0], Q[1] = x3, y3
        return [x3, y3]


def multiply(s, P, E):
    s = list(int(k) for k in "{0:b}".format(s))
    a, p = E["a"], E["p"]
    del s[0]
    T = P.copy()
    for i in range(len(s)):
        T = add(T, T, a, p)
        if (s[i] == 1):
            T = add(P, T, a, p)
    return T
```

```python
def parseUserPoint(user_point):
    return [int(c) for c in user_point.split(",")]


def parseUserBasis(user_basis):
    if len(user_basis) != 256:
        raise Exception("Input must be of length 256")
    basis = []
    for b in user_basis:
        if b == "Z":
            base = ns.Z
        elif b == "X":
            base = ns.X
        else:
            raise Exception(
                "Incorrect base, must be Standard (Z) of Hadamard (X)")
        basis.append(base)
    return basis


def measure(q, obs):
    res, _ = ns.qubits.measure(q, obs)
    return res


def randomBasis():
    r = randint(0, 1)
    return ns.Z if r else ns.X


def bitsToHash(bits):
    bit_string = ''.join([str(i) for i in bits])
    blocks = bytes(
        [int(bit_string[i:i + 8], 2) for i in range(0, len(bit_string), 8)])
    return sha256(blocks).digest()


def bitsToHex(bits):
    bit_string = ''.join([str(i) for i in bits])
    blocks = bytes(
        [int(bit_string[i:i + 8], 2) for i in range(0, len(bit_string), 8)])
    return blocks.hex()


def generateKeys(basis, private_key):
    seed(private_key)
    q_server_key = []
    q_user_key = []

    for i in range(256):
        qubits = ns.qubits.create_qubits(2)
        q1, q2 = qubits[0], qubits[1]
        ns.qubits.operate(q1, ns.X)
        ns.qubits.operate(q1, ns.H)
        ns.qubits.operate(q2, ns.X)
        ns.qubits.combine_qubits([q1, q2])
        ns.qubits.operate([q1, q2], ns.CX)
        q_server_key.append(measure(q1, randomBasis()))
        q_user_key.append(measure(q2, basis[i]))

    q_server_key = bitsToHash(q_server_key)
    q_user_key = bitsToHex(q_user_key)
    return q_server_key, q_user_key


def encrypt(message, key):
    cipher = AES.new(key, AES.MODE_ECB)
    ciphertext = cipher.encrypt(pad(message, 16))
    return ciphertext
```

The encryption algorithm implemented by the above code includes multiple parts. The first is the given elliptic curve `private_key`, which is used as the seed of the Python random number to generate the quantum operator (Z or X), and secondly, it is used to `netsquid` simulate the realization of the quantum Observe that the observed result value (0 or 1) is used to generate the sum `q_server_key`, `q_user_key` and `q_server_key` the sha256 hash value is used as the key for AES encryption.

The operating environment provides three input options. Option 1 allows the user to input coordinate values (x, y), and then returns the `[private_key]` scalar product result with the modified point. Option 2 allows the user to input 256 quantum operators, and returns the values of the quantum observations made by these operators, along with the AES-encrypted ciphertext. Option 3 is useless.

problem solving process

First of all, we must obtain the elliptic curve `private_key`. The elliptic curve code provided does not check whether the coordinates entered by the user are on the given curve, and only uses the given `a` and `p` curve parameters, but `b` not used. So we can use the same `a` sum `p`, but different `b` to create another elliptic curve. `b` The selection requirement is that the discrete logarithm on the corresponding curve conforms to the requirements of the Pohlig-Hellman algorithm.

The following code uses `b=6` the curve and or takes the order of the points using sarge.

```
p =
115792089237316195423570985008687907853269984665640564039457584007908834671663
a = 0
b = 6
EC = EllipticCurve(GF(p), [a, b])

Gx =
97739641136662608657079256755827419133838433889311376347497047878595450848685
Gy =
98100600220769146147883276184268394981687000350669426476581029710371895499142

G = EC(Gx, Gy)
G.order()
```

Perform a prime factorization of this order and `private_key` confirm that it meets the requirements of the Pohlig-Hellman algorithm according to the upper bound given in the code.

```
>>
factor(82708635169511568159693560720491362752815226084374474059483336146146842
78506)

***factors found***

P1 = 2
P1 = 7
P5 = 10903
P7 = 5290657
P11 = 10833080827
P14 = 22921299619447
P41 = 41245443549316649091297836755593555342121
```

IMPORTANT: From this point it's needed to install Netsquid

I was unable to install netquid on Kali, seems not compatible so I install and use REMnux in a VM then able to install netquid.

After inputting the coordinate pair, obtain the scalar product result, and then use the Pohlig-Hellman algorithm to obtain `private_key`

```
Qx =
3857225661745020856873269956141698742872251158780186082433874002180145459209
Qy =
6544502763778813556431537492609375417205638644494698005245711242038528944385

Q = EC(Qx, Qy)

dlogs = []
for fac in primes:
  t = int(G.order()) // int(fac)
  dlog = discrete_log(t*Q, t*G, operation="+")
  dlogs += [dlog]

private_key = crt(dlogs, primes)
```

After obtaining `private_key` it, we can obtain the generated `q_server_key` sequence of quantum operators. The principle is that the sequence generated by Python random numbers is determined by its seed. Using the same seed value, the generated sequence of random numbers is also the same.

```
seed(private_key)

server_basis = ""

for i in range(256):
  b = randomBasis()
  if (b == ns.Z):
    server_basis = server_basis + "Z"
  else:
    server_basis = server_basis + "X"

print("server_basis=", server_basis)
```

Through experiments, we can find that in `generateKeys` the method, when the same calling method is used for `q1` and , the returned result is always opposite. Therefore, we can use option 2 to input , then restore the obtained result to binary, and then invert bit by bit, we can get `q2``basis``measure``server_basis``q_server_key`

```
#q_user_key
hex = "a425ec07feabe32f689e7bf2322f171217a1549d2ee00f54622d99ea26dcf27d"

blocks = bytes.fromhex(hex)

bits = []
for block in blocks:
    s = bin(int(block))[2:].zfill(8)
    for i in s:
        bits.append(int(i))

server_bits = []
for b in bits:
    if (b == 0):
        server_bits.append(1)
    else:
        server_bits.append(0)

q_server_key = bitsToHash(server_bits)
```

After getting `q_server_key` it, you can perform AES decryption.

```
cipher_text =
bytes.fromhex("0842dbf2337a3be8b1a03ba2692ce7ed046902d537cc99613b73a372e280229a
4f4f6caca4e827a952ee88426702f1dcd0f03b9fcee64d5729d46d15954bbf6a234222058295fd2
c257eceab1fd9e5b0")

cipher = AES.new(q_server_key, AES.MODE_ECB)
plain_text = cipher.decrypt(cipher_text)
print(f"plain_text = {plain_text}")
```

Get the flag:

```
python3 Twisted_solution.py
plain_text =
b'HTB{E***T_W**_s***_b*****G_**_1**1_**D_*_h***_P**b4*****y_s*_I_**3_*_E**_S***
*}'
```