Shamir's Secret

overview

Shamir's Secret is an intermediate cryptography challenge from HTB.

The knowledge required to complete this challenge is focus on the calculation of modulus and its linear equation.

topic analysis

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

`server.py` The content is excerpted as follows

```python
from Crypto.Util.number import *
from secret import flag
import random
import os
def getrandbits(n):
    return bytes_to_long(os.urandom(n // 8))

N = 2**1024

# Generate random key(64-bit number of which 32 of those bits are 1)


key = 0
rem = list(range(64))
for _ in range(32):
    bitpos = random.choice(rem)
    rem.remove(bitpos)
    key |= 1 << bitpos

def doeval(poly, x):
    # Given polynomial and x value, generates y modulo N
    ans = 0
    for i, coeff in enumerate(poly):
        ans += x**i * coeff
        ans %= N
    return ans

def encrypt(msg, key):
    out = ()
    msg = bytes_to_long(msg)
    poly = [msg] + [getrandbits(1024) for _ in range(31)]

    for bitpos in range(64):
        if key & 1 << bitpos != 0:
            # Real
            x = getrandbits(1024)
            out += ((x, doeval(poly, x)),)
        else:
            # Fake
            x = getrandbits(1024)
            y = getrandbits(1024)
            out += ((x,y),)
    return out

def printenc(data, key):
    for pair in encrypt(data, key):
        print(pair)
```

```python
def menu():
    print("[1]: Get encrypted flag")
    print("[2]: Encrypt your own message")
    return int(input("> "))

doneflagenc = False
try:
    while True:
        try:
            option = menu()
        except:
            print("Invalid menu item")
            continue
        if option == 1:
            if doneflagenc:
                print("Nope")
                continue
            printenc(flag, key)
            doneflagenc = True
        elif option == 2:
            try:
                msg = bytes.fromhex(input("Input message as hex: "))
            except:
                print("Invalid message format")
                continue
            printenc(msg, key)
        else:
            print("Unknown option")
            continue
except:
    print("Unknown error ocurred")
```

The encryption algorithm implemented by the above code includes two steps.

1. The first step is to randomly generate a 64-bit bit `key`, of which 32 bits are 0, and the other 32 are 1.

2. The second step is to first generate an `poly` array containing 32 elements. Among them `poly[0]` is the plaintext, and the following 31 elements are random numbers, and then traverse `key` each bit in, if the bit is 1, then first generate a random number `x`, and then return `x` the `(x**0 * poly[0] + x**1 * poly[1] . . . +x**31 * poly[31]) mod N` result of the sum. And if the bit is 0, the two numbers returned are random numbers.

The running environment of the above code provides two input options.

- Option 1 `flag` encrypts the pair and returns the result, but it is only allowed to be used once.
- Option 2 encrypts the plaintext provided by the user and returns the result. There is no limit to the number of times this option can be used.

problem solving process

First of all, it must be obtained `key`.

According to the algorithm and value provided by the code, we can know that when the plaintext sum `x` is an even number, `(x^0 * poly[0] + x^1 * poly[1] . . . +x^31 * poly[31]) mod N` the result must be an even number, so we can use option 2 and input an even number, and then observe the returned result. If a certain value If the pair `x` is an even number and the other is an odd number, then it can be judged that the bit corresponding to the value pair is 0, and the cycle is repeated until 32 0 bits are found, and we get it `key`.

Then we can use `flag` the encryption result obtained by option 1, because `key` there are 32 bits in 1, so we can get 32 linear equations, and the unknowns are `poly` 32 elements in the value, using sage we can use `N` the modulus based on The numerical linear equation is solved, and in the result `poly[0]` is `flag` the plaintext of .

The solution code is as follows:

```python
from pwn import remote
from sage.all import *
from Crypto.Util.number import bytes_to_long, long_to_bytes

conn = remote('<instance IP>', <instance PORT>, level = 'error')

def tesBit():
  conn.recvuntil(">")
  conn.sendline("2")
  conn.recvuntil("Input message as hex: ")
  conn.sendline("00")
  pairs = []
  for i in range(64):
    line = conn.recvline()
    pairs.append(eval(line))
  return pairs

def getFlagPairs():
  conn.recvuntil(">")
  conn.sendline("1")
  pairs = []
  for i in range(64):
    line = conn.recvline()
    pairs.append(eval(line))
  return pairs

## recover key
bits = [1] * 64
count_of_zero_bits = 0

while (count_of_zero_bits != 32):
  print(".")
  key_pairs = tesBit()
  for i in range(64):
    x, y = key_pairs[i]
    if ((x % 2 == 0) and (y % 2 == 1)):
      bits[i] = 0

  count_of_zero_bits = sum(bits)

bits.reverse()

key_bits = ""
for i in range(64):
  key_bits = key_bits + str(bits[i])

print("key_bits =", key_bits)
```

```
key = int(key_bits, 2)

## recover flag
flag_pairs = getFlagPairs()

R = IntegerModRing(2**1024)

x_values = []
y_values = []

for bitpos in range(64):
  if key & 1 << bitpos != 0:
    # Real
    x, y = flag_pairs[bitpos]

    x_list = []
    for j in range(32):
      x_list.append(x**j)

    x_values.append(x_list)
    y_values.append(y)

x_matrix = Matrix(R, x_values)
y_vector = vector(R, y_values)

poly = x_matrix.solve_right(y_vector)

print("flag = ", long_to_bytes(int(poly[0])))
```