SCTF Finals Write up

BaaaaaaaaaaaaaaaaarkingDog

1. Crypto – LCG(15 solve)

<server.py>

```
import signal
import random


class LCG:
    def __init__(self, s, k):
        self.init_state = s

        k1, k2, k3 = k
        self.x = (k1 - 0xdeadbeef) % k3
        self.y = (k1 * 0xdeadbeef) % k3
        self.z = k2
        self.m = k3

    def __iter__(self):
        self.index = 0
        self.size = 32
        self.s0, self.s1 = self.init_state
        return self

    def __next__(self):
        if self.index >= self.size:
            raise StopIteration

        self.index += 1
        s0, s1 = self.s0, self.s1
        self.s0, self.s1 = s1, (self.x * s1 + self.y * s0 + self.z) % self.m
        return self.s1


if __name__ == '__main__':
    signal.alarm(60)

    s0, s1, k1, k2, k3 = [random.getrandbits(64) for i in range(5)]

    s = (s0, s1)
    k = (k1, k2, k3)

    cnt = 16
    for i, v in enumerate(LCG(s, k)):
        guess = int(input())
        if guess == v:
            cnt += 1
        else:
            cnt -= 1

        if i <= 16:
            print(v)

    if cnt >= 16:
        with open('flag.txt', 'r') as f:
            print(f.read())
```

문제의 이름인 LCG는 Linear Congruential Generator를 의미합니다. 소스 코드를 살펴보면 랜덤으로 정해지는 `k1`, `k2`, `k3` 값을 가지고 `s0`, `s1` 값을 매 질의마다 바꿔나가고, 32번의 질의 중에서 `s1` 값을 24번 이상 맞춘다면 flag를 획득하게 됩니다. 바꿔말하면 8번 까지는 틀려도 괜찮습니다. 그리고 초반 16번의 경우에는 `s1` 값을 알려줍니다.

우리는 `k1`, `k2`, `k3` 으로부터 유도되어 LCG에서 쓰이는 `x`, `y`, `z`값을 알아낼 경우 답을 알아맞출 수 있음을 쉽게 알 수 있는데, 일단 `z`를 알아내고 나면 `x`와 `y`는 연립이차방정식을 풀어냄으로서 구할 수 있습니다. 또한 `z`는 수식을 마구 변형하다보면 두 상수의 합동 관계를 알아낼 수 있고 이 두 상수의 차는 `z`의 배수이기에 이런 쌍을 여럿 구해 gcd를 취해서 `z`를 알아낼 수 있습니다.

solver는 링크에서 확인할 수 있습니다. (solver.py)

2. Crypto – MQ(8 solve)

<Server.py>

```
from random import randint
from binascii import hexlify
import sys


def wline(msg=''):
    sys.stdout.write(msg + '\n')
    sys.stdout.flush()


class MQpoly:
    def __init__(self, terms, p, n):
        self.terms = terms
        self.p = p
        self.n = n

    def apply(self, vals):
        if len(vals) != self.n:
            raise ValueError('Values for n variables should be given')

        quad, uni, c = self.terms

        ret = 0
        for i in range(self.n):
            for j in range(i, self.n):
                ret += quad[i][j] * vals[i] * vals[j]
                ret %= self.p

        for i in range(self.n):
            ret += uni[i] * vals[i]
            ret %= self.p

        ret += c
        ret %= self.p

        return ret

    def __str__(self):
        quad, uni, c = self.terms

        st_terms = []
        for i in range(self.n):
            for j in range(i, self.n):
                v = quad[i][j]
                if v != 0:
                    st_terms.append('%dx%dx%d' % (v, i+1, j+1))

        for i in range(self.n):
            v = uni[i]
            if v != 0:
                st_terms.append('%dx%d' % (v, i+1))

        if c != 0:
```

```
            st_terms.append('%d' % (c,))

        return ' + '.join(st_terms)

    @classmethod
    def random_element(cls, p, n):
        quad = [[randint(0, p-1) if j <= i else None for j in range(n)]
                for i in range(n)]

        for i in range(n):
            for j in range(i+1, n):
                quad[i][j] = quad[j][i]

        uni = [randint(0, p-1) for i in range(n)]
        c = randint(0, p-1)

        terms = (quad, uni, c)
        return cls(terms, p, n)


if __name__ == '__main__':
    p = 131
    n = 32

    with open('flag.txt', 'rb') as f:
        flg = f.read()

    with open('/dev/urandom', 'rb') as f:
        flg += f.read(n - len(flg))

    mq = MQpoly.random_element(p, n)
    wline(str(mq))
    for i in range(n+1):
        lst_1 = sys.stdin.buffer.read(n)
        lst_2 = [(b1 + b2) % p for b1, b2 in zip(lst_1, flg)]
        clue = [mq.apply(lst_1), mq.apply(lst_2)]

        wline(hexlify(bytes(clue)).decode())
```

소스 코드를 살펴보면 MQpoly class는 입력으로 받은 `x1, x2, .. , x31`에 대해 랜덤으로 정해지는 이차함수에 대입한 결과를 반환해줍니다. 이 때 해당 이차식의 계수는 공개를 해줍니다. 우리는 각 질의에서 flag의 값을 모른채로 flag의 변화량을 지정해줄 수 있습니다. (예를 들어 `x1`을 3 증가시키고, `x5`를 9 증가시키고...) 그리고 flag를 해당 변화량만큼 변화했을 때 이차함수에 대입한 결과가 얼마인지를 알 수 있습니다. 우리는 33번의 질의 내로 flag를 알아내야합니다.

이 문제에서 중요한 것은 다른 모든 값을 그대로 둔 채 `f1`를 1 증가시켰을 때 이차함수의 결과가 얼마나 변하는지를 계산하는 것입니다. `f1`를 1 증가시킬 경우 이차함수의 결과의 변화량은 `2 · quad[1][1] · f1 + quad[1][2] · f2 + quad[1][3] · f3 +... + quad[32][3] · f32 + 2 · uni[1] + 1` 입니다.

(`f_i`는 flag의 각 바이트를 의미) `quad`, `uni`는 모두 공개된 값이므로 결국 변화량은 `f_i`에 대한 연립일차방정식을 이루게 되고 맨 처음엔 flag를 변화시키지 않은 질의를 보낸 이후, 32번에 걸쳐 `f1`, `f2`, `...` , `f31` 각각을 1 증가시켜 질의를 보내고 나면 32차 연립 방정식을 해결함으로서 답을 구할 수 있습니다.
solver는 링크에서 확인할 수 있습니다. (해당 solver에는 연립 방정식을 찾아내는 부분까지만 있고 이후에는 sage로 계수를 옮겨 간단하게 결과를 바로 확인했습니다.) (solver.py)

3. Crypto – polyPKC(7 solve)

<problem.py>

```
from Crypto.Util import number

class PubKey(object):

    def __init__(self, poly, order, n):
        self.poly = poly
        self.order = order
        self.N = n

    def poly_rand(self):
        return [number.getRandomRange(0, self.N) for i in range(0, self.order)]

    def poly_mul(self, p, q):
        res = [0 for _ in range(self.order)]
        for i in range(self.order):
            for j in range(self.order):
                k = (i + j) % self.order
                res[k] = (res[k] + (p[i] * q[j])) % self.N
        return res

    def encrypt(self, m):
        x = self.poly_mul(self.poly, self.poly_rand())
        x[0] = (x[0] + m) % self.N
        return x


class PrivKey(object):

    def __init__(self, t, N):
        self.t = t
        self.N = N

    def poly_apply(self, p, x):
        res = 0
        for i in range(len(p)):
            res += p[i] * pow(x,i,self.N)
        return res % self.N


    def decrypt(self, c):
        return self.poly_apply(c, self.t)

class KeyGenerator(object):

    def __init__(self, order, bits):
        q = 2
        p = 2
        N = p * q
        r = number.getRandomRange(1, N)
        t = pow(r, (p-1) * (q-1) / order, N)
```

```
        while (pow(t,order,N) != 1) or t == 1 or t == N -1:
            p = number.getPrime(bits)
            q = number.getPrime(bits)
            N = p * q
            r = number.getRandomRange(1, N)
            t = pow(r, (p-1) * (q-1) / order, N)

        pub_poly = [number.getRandomRange(0, N) for i in range(0, order)]

        fin = 0
        for i in range(1,order):
            fin += pub_poly[i] * pow(t, i, N) % N
            fin %= N
        pub_poly[0] = N - fin

        self.pubKey = PubKey(pub_poly, order, N)
        self.privKey = PrivKey(t, N)

key = KeyGenerator( 51, 512 )

with open('flag','r') as fp:
    m = int(fp.read().encode('hex'), 16)
with open('enc','w') as fp:
    fp.write('order : %d\n' % key.pubKey.order)
    fp.write('public key : %s\n' % str(key.pubKey.poly))
    fp.write('N : %d\n' % key.pubKey.N)
    fp.write('encrypted message : %s\n' % str(key.pubKey.encrypt(m)))
```

polynomial, order, N이 public key이고 t는 private key입니다. t는 51승을 할 경우 mod N에서 1과 합동입니다. t를 복원하고 나면 encrypted message를 t진법으로 읽어들여 수로 변환한 후 N으로 나눈 나머지를 취해 원래의 message를 찾아낼 수 있습니다.

$t^{51} - 1 = (t - 1)(t^2 + t + 1)(\ldots) \equiv 1 \pmod{N}$ 이므로 t-1이 p 혹은 q의 배수, 즉 t가 1일 때의 식의 값(= 그냥 계수의 합)이 p 혹은 q의 배수일 확률이 충분히 있음을 알 수 있습니다. 이 추론을 바탕으로 계수의 합과 N의 gcd를 취해보면 바로 소인수를 찾아낼 수 있고, 이후에는 t의 후보 51개에 대해 message를 확인해보면 됩니다. solver는 링크에서 확인할 수 있습니다.

(solver.py)