EDU CTF course - Computer Security 2022 Fall

Crypto Write up by B10832008 蔡芸軒

There Is No Such Thing As Absolute Security ...

Challenge Name : COR

Points : 10

Category : Crypto

Description : Although I'm not a statistician, the output is quite random, right?

Approach :

This is a triple LFSR (linear feedback shift register) encryption. We can get the ciphertext and the length of each initial bit, which is 27, 23, 25. However, we don't know the plaintext. We can solve this by enumerating all possible initial states, but it has 2^23 * 2^25 * 2^27 = 2^75 possibilities, which is not possible to solve by computers nowadays. However, we can solve this with the **LFSR correlation attack**.

The algorithm is, we create three LFSR chains, each time we get one bit from each, denoted by **x1, x2** and **x3**, and if x1 is 1, return x2, else return x3. If we enumerate all 2^3 = 8 possible values of (x1, x2, x3) and the output, we would find that **there is a 75% correlation between x2 and the output, as well as between x3 and the output**.

By this breakthrough, **we can separately enumerate x2, x3, and then x1 in order**. For x2 and x3, we enumerate all possible initial states and generate the possible x2 and x3. If there is about 75% correlation between it and the outputs, that means it is the original x2 and x3 we're looking for. After getting x2 and x3, then we can enumerate all possible initial states of x1 and check if the result is exactly the output we got.

By this approach, it cut down the time complexity from 2^75 to about 2^27 (2^23 + 2^25 + 2^27). It still costs lots of time though. So I write my code in C++ instead of Python.

Code :

```
vector<bool> lfsr1_init, lfsr2_init, lfsr3_init, lfsr1_list, lfsr2_list, lfsr3_list, flag;

void lfsr2_generate(vector<bool> list){
    if(list.size() == 23){
        LFSR lfsr({0, 5, 7, 22}, list);
        int counter = 0 ;
        for(int i=0; i<232; i++) lfsr.getbit();
        for(int i=232; i<432; i++) if(lfsr.getbit() == output[i]) counter++;
        // using 70% as threshold
        if(counter >= 140) lfsr2_init = list;
    }
    else{
        vector<bool> cp = list;
        cp.push_back(0);
        lfsr2_generate(cp);
        cp = list;
        cp.push_back(1);
        lfsr2_generate(cp);
    }
}

void lfsr3_generate(vector<bool> list){
    if(list.size() == 25){
        LFSR lfsr({0, 17, 19, 24}, list);
        int counter = 0 ;
        for(int i=0; i<232; i++) lfsr.getbit();
        for(int i=232; i<432; i++){
            if(lfsr.getbit() == output[i]) counter++;
        }
        // using 70% as threshold
        if(counter >= 140) lfsr3_init = list;
    }
    else{
        vector<bool> cp = list;
        cp.push_back(0);
        lfsr3_generate(cp);
        cp = list;
        cp.push_back(1);
        lfsr3_generate(cp);
    }
}

void lfsr1_generate(vector<bool> list){
    if(list.size() == 27){
        LFSR lfsr({0, 13, 16, 26}, list);
        int counter = 0 ;
        for(int i=0; i<232; i++) lfsr.getbit();
        for(int i=232; i<432; i++){
            bool x = lfsr.getbit()?lfsr2_list[i]:lfsr3_list[i];
            if(x==output[i]) counter++;
        }
        if(counter == 200) lfsr1_init = list;
    }
    else{
        vector<bool> cp = list;
        cp.push_back(0);
        lfsr1_generate(cp);
        cp = list;
        cp.push_back(1);
        lfsr1_generate(cp);
    }
}
```

```cpp

int main(){
    lfsr2_generate({});
    lfsr3_generate({});

    LFSR lfsr2({0, 5, 7, 22}, lfsr2_init);
    LFSR lfsr3({0, 17, 19, 24}, lfsr3_init);
    lfsr2_list.resize(0);
    lfsr3_list.resize(0);
    for(int i=0;i<432;i++){
        lfsr2_list.push_back(lfsr2.getbit());
        lfsr3_list.push_back(lfsr3.getbit());
    }

    lfsr1_generate({});
    LFSR lfsr1({0, 13, 16, 26}, lfsr1_init);
    lfsr1_list.resize(0);
    for(int i=0;i<432;i++){
        lfsr1_list.push_back(lfsr1.getbit());
        lfsr2_list.push_back(lfsr2.getbit());
        lfsr3_list.push_back(lfsr3.getbit());
    }

    flag.resize(0);
    for(int i=0; i<232; i++){
        bool bit = lfsr1_list[i]?lfsr2_list[i]:lfsr3_list[i];
        bit = bit ^ output[i];
        flag.push_back(bit);
    }
    for (int i=0; i<232; i+=8){
        int ascii = 0;
        for(int j=0; j<8; j++){
            ascii = ascii * 2 + flag[i+j];
        }
        cout << char(ascii);
    }
}
```

Result :

```
FLAG{W0W_you_Kn0w_COR_477ACK}
```

Challenge Name : POA

Points : 10

Category : Crypto

Description : A confidential channel for receiving messages from my friend.
nc edu-ctf.zoolab.org 10101

Approach :

This is a CBC encryption. CBC encryption does padding before encryption. In this case it pads an 0x08 and 0x00 to the rest bytes. If we send a ciphertext to the server, we'll know if the plaintext can be unpad successfully. Thus we can solve this with the padding oracle attack.

Each block in CBC has 16 bytes. For each block **i**, **pt[i] = decrypt(ct[i]) ^ ct[i-1]**, so we can solve each block separately. In each block, we start from the last byte to the first byte, enumerating all possible values of the byte from 0 to 256, excluding its original value.

Take the last byte for example, if there's no padding error occurs only when the last byte of **ct[i-1]** replaced by **x,** that means **decrypt(ct[i]) ^ x = 0x08**, then we can deduce that **pt = decrypt(ct[i]) ^ ct[i-1] = 0x08 ^ x ^ ct[i-1]**. After solving the last bit, now we have to modify it again to make the last bit of the plaintext 0x00 so that we can go on testing the rest of the bytes. The modified one is **ct[i-1] = x ^ 0x80 ^ 0x00**

Code :

```python
def byte_to_array(byte_string):
    return [[byte_string[i+j]for j in range(16)] for i in range(0, len(byte_string), 16)]

def array_to_byte(array):
    return bytes(bytearray([j for array in array for j in array]))

r = remote('edu-ctf.zoolab.org', 10101)
ct = byte_to_array(bytes.fromhex(r.readline()[:-1].decode()))

for block_i in range(1, len(ct)):
    block_pt = []
    mod_ct = copy.deepcopy(ct[:block_i+1])
    for byte_i in range(15, -1, -1):
        for test_val in range(256):
            if(ct[block_i-1][byte_i] == test_val): continue
            mod_ct[block_i-1][byte_i] = test_val
            r.sendline(array_to_byte(mod_ct).hex().encode('ascii'))
            res = r.readline()
            if res == b'Well received :)\n':
                block_pt = [test_val ^ 0x80 ^ ct[block_i-1][byte_i]] + block_pt
                mod_ct[block_i-1][byte_i] = test_val ^ 0x80
                break
        else:
            block_pt = [0x80] + block_pt
            mod_ct[block_i-1][byte_i] = 0x80 ^ ct[block_i-1][byte_i]

    print(array_to_byte([block_pt]))
```

Result :

```
[x] Opening connection to edu-ctf.zoolab.org on port 10101
[x] Opening connection to edu-ctf.zoolab.org on port 10101: Trying 60.248.184.73
[+] Opening connection to edu-ctf.zoolab.org on port 10101: Done
b'FLAG{ip4d_pr0_iS'
b'_r3ally_pr0_4Nd_'
b'f1aT!!!!}\x80\x00\x00\x00\x00\x00\x00'
```

Challenge Name : LSB

Points : 150

Category : Crypto

Description : I'll tell you the least significant information...

Approach :

This is RSA encryption. We know the public key N and e, the ciphertext ct. Also we can get the remainder of plaintext divide 3 by sending a ciphertext to the server. We can solve this with the **LSB (least significant bit) oracle attack**.

First we have ct = pt ^ e % N. We can multiply each side by 3 ^ e, means 3 ^ e * ct % N = (3 * pt) ^ e % N. Then we can send ct as ciphertext to server to get pt % 3, denoted by r, and then send 3 ^ e * ct % N as ciphertext to server to get (3 * pt) % N % 3, denoted by r3.

There are 3 possible value of r3 :
IF 0 < 3 * pt < N :
    r3 = (3 * pt) % 3
IF N < 3 * pt < 2 * N :
    r3 = (3 * pt - N) % 3
IF 2 * N < 3 * pt < 3 * N :
    r3 = (3 * pt - 2N) % 3

By this, we can find which range pt is in. And then we can set the value of current r3 to r to see if 9 * pt < N or N < 9 * pt < 2 * N and the like. Do this again and again until finding the value.

Remind that in order to accelerate, always mod N before doing multiplying computation.

Code :

```python
r = remote('edu-ctf.zoolab.org', 10102)
n = int(str(r.readline())[2:-3])
e = int(str(r.readline())[2:-3])
enc = int(str(r.readline())[2:-3])

# set the lower and upper bound for c
pt_lower_bound, pt_upper_bound = 0, n

# get pt % 3
r.sendline(str.encode(str(enc)))
pt_remainder = int(r.readline())
e3, multiple = pow(3, e), pow(3, e)

while pt_lower_bound+1 < pt_upper_bound:
    r.sendline(str.encode(str((enc * multiple) % n)))
    pt3_remainder = int(r.readline())
    range = pt_upper_bound - pt_lower_bound
    if (3 * pt_remainder) % 3 == pt3_remainder:
        pt_upper_bound = pt_lower_bound + range // 3 + 1
    elif (3 * pt_remainder - n % 3) % 3 == pt3_remainder:
        pt_upper_bound = pt_lower_bound + range * 2 // 3 + 1
        pt_lower_bound = pt_lower_bound + range // 3
    elif (3 * pt_remainder - 2 * (n % 3)) % 3 == pt3_remainder:
        pt_lower_bound = pt_lower_bound + range * 2 // 3

    pt_remainder = pt3_remainder

    # multiply the multiple of ct we test by 3^e each iteration
    multiple = (multiple * e3) % n

print(pt_lower_bound.to_bytes(math.ceil(math.log(pt_lower_bound, 256)), 'big'))
```

Result :

```
[x] Opening connection to edu-ctf.zoolab.org on port 10102
[x] Opening connection to edu-ctf.zoolab.org on port 10102: Trying 60.248.184.73
[+] Opening connection to edu-ctf.zoolab.org on port 10102: Done
b"FLAG{lE4ST_519Nific4N7_Bu7_m0S7_1MporT4Nt}\xc9\xbe\xbeEt\x92\xf4\x05_\xb3\x9b\xde\xcex\x88S\x08)iw\x90\x7f\x93\xf54
xbb\xca\xc66\x06\xfd\xfd\x1f\x15\xd4\xc15h\xbd\x068\r<\x0f\x85\xa9\xf0\xc5\xa2\xdd\x9a\xf0J\xde\xb3K\x07\x8b\x8fBV#\x
03\xdc\x1fW\rK]\xec\x86\xd0#L\xff\x1d\x82a\xe8\xa7\xda\x84\x84\xd6\xc1\x8e\xed\xe7\xa5A\x96\xea}\xf0p\xab0z\xf2\x00\x
ebj{\xed\x06oz'\xd1\x98\xdcL\x82\x9a\xab\xd5\x85\xdeB\xf1\x90\xd9\xdf\xdb\x17kz\xffTD\xc7\xa6\xf5\x8d?s\xa6D\x02 \x93
\xe0F^\xfd4\x803\xc5\xff\xb5\xc2\x86\xd7\xf0\xd9^>\x0f\x9c\xfb\xfc\xf6\xd0\xbe!\xed\x81\x91\xc61\xdf\x1d\xff\xee\x86\
x8c\x8f\xd2\xa0j\xee#!s\xe7p\xb3\xd4\xe4\x90\xd8l\x86_\xd6\x02x\xf7\xbaKa\xca\xd39"
[*] Closed connection to edu-ctf.zoolab.org port 10102
```

Challenge Name : XOR

Points : 300

Category : Crypto

Description : none

Approach :

The encryption program in this challenge is giving an random initial state at first, each time shifting the initial state 1 bit to the left in a 64-bit field, and XOR to the number *0x1da785fc480000001* if the leftmost bit is 1.

I first came up with separating the initial state into 4 parts and using brute force approaches in each to see how much it matches the output. Obviouslyit failed because when trying one part, bits in the other three parts are just too random.

Then I calculated the mapping function which can map the initial state to the final 70 bits of output, trying to find some regularity between them. For example, one bit can directly map to another. But I quickly found that the mapping function is so complex and completely irregular.

But one thing is true, which is that no matter what the initial state is and how many times the operation is executed, the final bit can be computed directly from the initial state. To be more specific, we can get any bit by computing the 64 bits of initial state, without computing the intermediate processes. For example, 10000 th bit = state[0] ^ state[3] ^ state[22]. we don’t have to do all 10000 iterations.

After thinking for a long while, I came up with an idea of transferring the mapping function into a matrix. In that way, we can get the initial state by multiplying its inverse matrix (mod 2). I wrote a script to check the feasibility and corroborate that it works !!

The coding part of this challenge is a little bit cumbersome. Since numpy doesn't have the function computing modular matrix inverse, we have to install *SageMath*. Moreover, the matrix’s size is 64 * 64, which is too large for *Matrix(IntegerModRing(2), mat)* in *SageMath*, so I used *Matrix(GF(2), matrix)* in *SageMath* instead.

Code :

```python
output = [1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1,
    0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1,
    1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0,
    1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1,
    1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0,
    0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1,
    0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0,
    0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0,
    1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0,
    1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0]

xor, xor_list = 0x1da785fc480000001, []

for i in range(64, 0, -1):
    if xor % 2 == 1:
        xor_list = [i] + xor_list
    xor //= 2

# construct the list for bits that each bit relate to
def construct_relate_list():
    global relate
    relate += [set({})]
    for idx in xor_list:
        for element in relate[len(relate)-65]:
            if element in relate[idx-65]:
                relate[idx-65].remove(element)
            else:
                relate[idx-65].add(element)

relate = [set({i}) for i in range(64)]
for _ in range(406 * 37):
    construct_relate_list()

# construct the matrix which can transfer initial states to (64 of) the last 70 bits of output
matrix = numpy.array([[1 if j in relate[i * 37 + 36] else 0 for j in range(64)] for i in range(336, 336+64)])

# get init state by multiplying the inverse matrix
init_state_list = numpy.dot(Matrix(GF(2), mat).inverse(), numpy.array([[output[i]] for i in range(336, 336 + 64)])) % 2
init_state = 0
for i in init_state_list:
    init_state = init_state * 2 + i[0]

def getbit():
    global init_state
    state <<= 1
    if state & (1 << 64):
        state ^= 0x1da785fc480000001
        return 1
    return 0

ori_output, flag = [], 0
for _ in range(336 + 70):
    for __ in range(36):
        getbit()
    ori_output.append(getbit())
for i in range(336):
    flag = flag * 2 + (output[i] ^ ori_output[i])

print(flag.to_bytes(math.ceil(math.log(flag, 256)), 'big'))
```

Result :

```
b'FLAG{Y0u_c4N_nO7_Bru73_f0RCe_tH15_TiM3!!!}'
```

Challenge Name : dlog

Points : 20

Category : Crypto

Description : nc edu-ctf.zoolab.org 10103

Approach :

This is a simple discrete logarithm problem and can be solved by the Pohlig–Hellman algorithm.

We have $b \hat{} flag \% p = ct$, while we can choose our own b and p and the server will return $b \hat{} flag \% p$ to us. Usually it is safe except that the factors of prime p are all small numbers.

First, we have to choose an unsafe prime for the Pohlig–Hellman algorithm. That is, $p - 1$ should be a multiple of some small primes. I set $p - 1$ to 2 at first, and then randomly choose small primes to multiply until it reaches 1024 bit. And then check if $x + 1$ is prime, if so, we have $p = x + 1$, if not, repeat until success. The reason why I set x 2 instead of 1 is that all 1024 bits primes are odd, so $p - 1$ must be even.

Code :

```python

def unsafe_p_gen():
    while True:
        ans = 2
        while ans.bit_length() < 1024:
            ans *= getPrime(7)
        if ans.bit_length() == 1024 and isPrime(ans + 1):
            return ans + 1

p = unsafe_p_gen()
b = 11

r = remote('edu-ctf.zoolab.org', 10103)
r.readuntil(b"give me a prime")
r.sendline(str(p).encode())
r.readuntil(b"give me a number")
r.sendline(str(b).encode())
r.readuntil(b"The hint about my secret:")
ct = int(r.readline().strip())

b = b % p
ct = ct % p
flag = int(discrete_log(p, ct, b))
print(flag.to_bytes(math.ceil(math.log(flag, 256)), 'big'))
```

Result :

```
[x] Opening connection to edu-ctf.zoolab.org on port 10103
[x] Opening connection to edu-ctf.zoolab.org on port 10103: Trying 60.248.184.73
[+] Opening connection to edu-ctf.zoolab.org on port 10103: Done
b"FLAG{D0_No7_SLiP!!!1t'5_SM0o7h_OwO}\xd0\x0b\xca#\x1a\xfa\x8e\x86\xc6\x89\x1f\xa9Y\x9c\x93\x9c\x15\xd2#\xefu\xb8]\x9
2\x89\x00i\x981y\xbcyV\xbfm#\x93;\xd7>\xe2\n\xc1t;\xa48K\xeb\xf5\xa5\x12\\5\x97]`\xea3`\xde\xf1zE}\xea\x1e~\x1dc\x90\
xc6\xf6\x93A[eo\xa3\xb7A\xc4\x7f\x89\x8a\x88#{\xa3j\x04\x1bu"
```

Challenge Name : DH

Points : 200

Category : Crypto

Description : nc edu-ctf.zoolab.org 10104

Approach :

If we check the given encryption program file carefully, we'll found that this is a simple formula :
*(((g ^ a % p) ^ b) % p * flag) % p = c*, where
g : provided by user,
a : random number which we don't know,
b : random number which we don't know,
p : random prime number which we know,
c : the calculation result which we can get,
and the only constraint is that g, a and b are all in the range from 2 to p - 2.

Since we want to get the flag, I came up with the idea of making *(g ^ a % p) ^ b = 1*. By this, c, which we can get, would equal to the flag. The equation, (g ^ a % p) ^ b = 1, can be simplified as *g ^ (a * b) % p = 1*.

Fermat's little theorem states that *x ^ (p - 1) = 1 (mod p)* if *p* is prime and *x* is not a multiple of *p*. We can not set *g = x ^ (p - 1)* due to the constraint of *g % p != 1*. Then I came up with letting *g = x ^ ((p - 1) / 9)*. Whenever *p - 1* is multiple of 9 as well as *a* and *b* are both multiple of 3, we got *g ^ (a * b) % p = 1*.

Since whether we can get the flag depends on the random number *a*, *b*, and the random prime *p*, we won't get the flag everytime by this approach. Because of this, I use a for loop to try until I get the flag. It won't cost much time since the probability we get the flag in one time is (1 / 9) * (1 / 3) * (1 / 3) = 1 / 81.

Code :

```python
for i in range(81): # 9*3*3, can do even less if lucky
    r = remote('edu-ctf.zoolab.org', 10104)
    # get the prime p
    p = int(str(r.readline())[2:-3])
    if (p-1) % 9 == 0:
        g = pow(3, ((p-1)//9), p)
        r.sendline(str.encode(str(g)))
        try:
            flag = int(str(r.readline())[2:-3])
            flag = flag.to_bytes(math.ceil(math.log(flag, 256)), 'big')
            # check if we get a correct flag
            if chr(flag[0]) == "F":
                print(flag)
                break
        except:
            print('Bad :(')
```

Result :

```
[x] Opening connection to edu-ctf.zoolab.org on port 10104
[x] Opening connection to edu-ctf.zoolab.org on port 10104: Trying 60.248.184.73
[+] Opening connection to edu-ctf.zoolab.org on port 10104: Done
[x] Opening connection to edu-ctf.zoolab.org on port 10104
[x] Opening connection to edu-ctf.zoolab.org on port 10104: Trying 60.248.184.73
[+] Opening connection to edu-ctf.zoolab.org on port 10104: Done
[x] Opening connection to edu-ctf.zoolab.org on port 10104
[x] Opening connection to edu-ctf.zoolab.org on port 10104: Trying 60.248.184.73
[+] Opening connection to edu-ctf.zoolab.org on port 10104: Done
[x] Opening connection to edu-ctf.zoolab.org on port 10104
[x] Opening connection to edu-ctf.zoolab.org on port 10104: Trying 60.248.184.73
[+] Opening connection to edu-ctf.zoolab.org on port 10104: Done
[x] Opening connection to edu-ctf.zoolab.org on port 10104
[x] Opening connection to edu-ctf.zoolab.org on port 10104: Trying 60.248.184.73
[+] Opening connection to edu-ctf.zoolab.org on port 10104: Done
[x] Opening connection to edu-ctf.zoolab.org on port 10104
[x] Opening connection to edu-ctf.zoolab.org on port 10104: Trying 60.248.184.73
[+] Opening connection to edu-ctf.zoolab.org on port 10104: Done
[x] Opening connection to edu-ctf.zoolab.org on port 10104
[x] Opening connection to edu-ctf.zoolab.org on port 10104: Trying 60.248.184.73
[+] Opening connection to edu-ctf.zoolab.org on port 10104: Done
[x] Opening connection to edu-ctf.zoolab.org on port 10104
[x] Opening connection to edu-ctf.zoolab.org on port 10104: Trying 60.248.184.73
[+] Opening connection to edu-ctf.zoolab.org on port 10104: Done
[x] Opening connection to edu-ctf.zoolab.org on port 10104
[x] Opening connection to edu-ctf.zoolab.org on port 10104: Trying 60.248.184.73
[+] Opening connection to edu-ctf.zoolab.org on port 10104: Done
b'FLAG{M4yBe_i_N33d_70_checK_7he_0rDEr_OF_G}'
```

Challenge Name : node

Points : 100

Category : Crypto

Description : none

Approach :

This is an elliptic curve encryption program. We have the curve's equation $y \hat{} 2 = x \hat{} 3 - 3 * x + 2$. I found the parameter a and b in this program satisfy $4 * a \hat{} 3 + 27 * b \hat{} 2 = 0$, which means this is a **singular curve**. Singular curve is not secure in elliptic curve encryption.
Since the equation
$y \hat{} 2 = x \hat{} 3 - 3 * x + 2$ can be rewritten in the form $y \hat{} 2 = (x - \alpha) \hat{} 2 * (x - \quad)$, where $\alpha =$ 1 and $\quad = -2$, we can solve this challenge by **node attack**.

In node attack, we have a mapping function
$\varphi(x, y) = (y + sqrt(a - b) * (x - a)) / (y - sqrt(a - b) * (x - a))$
it has been proved by utaha that, for any point P and Q on the curve, $\varphi(P+Q) = \varphi(P) * \varphi(Q)$, thus, $\varphi(P * d) = \varphi(P) \hat{} d$.

In this program, we know the point G and the point F, which is equal to $G * flag$, and we want to get the flag. First I calculate $\varphi(G)$ and $\varphi(G * flag)$, and want to get the $flag$ which satisfy
$\varphi(G) \hat{} flag \% p = \varphi(G * flag)$. In this way, we can use the discrete logarithm method used in ***dlog*** to get the flag.

Code :

```python
# This function is from https://gist.github.com/nakov/60d62bdf4067ea72b7832ce9f71ae079
# Can use Sage module instead
def modular_sqrt(a, p):
    def legendre_symbol(a, p):
        ls = pow(a, (p - 1) // 2, p)
        return -1 if ls == p - 1 else ls

    if legendre_symbol(a, p) != 1:
        return 0
    elif a == 0:
        return 0
    elif p == 2:
        return p
    elif p % 4 == 3:
        return pow(a, (p + 1) // 4, p)

    s = p - 1
    e = 0
    while s % 2 == 0:
        s //= 2
        e += 1

    n = 2
    while legendre_symbol(n, p) != -1:
        n += 1

    x = pow(a, (s + 1) // 2, p)
    b = pow(a, s, p)
    g = pow(n, s, p)
    r = e

    while True:
        t = b
        m = 0
        for m in range(r):
            if t == 1:
                break
            t = pow(t, 2, p)

        if m == 0:
            return x

        gs = pow(g, 2 ** (r - m - 1), p)
        g = (gs * gs) % p
        x = (x * gs) % p
        b = (b * g) % p
        r = m

def Phi(alpha, beta, x, y):
    ans = ((y + modular_sqrt(alpha-beta, p)*(x-alpha)) % p) * pow((y - modular_sqrt(alpha-beta, p)*(x-alpha)) % p, -1, p)
    return ans % p

Point = namedtuple("Point", "x y")
p = 143934749405770267808039109533241671783161568136679499142376907171125336784176335731782823029409453622696871327283737309148105009645408337908364715252952913
    322558857826125357939557272950776497159778396750983932456366682771945699642843910855001472647561367694613650577664546895409254178984894650442674939558 01
G = Point(101806057140780850544714530443644783825785167075147195900696966628348944447492085252540090679241301721340985975519224144331425477628386574016040358648 7
    523532638024005272501632977811897492853920871543776848902874510789376923805561921269716690690156736626355614257355937957438521412327110661815422506703872033333,
    210708770610471404482239943378636153064994127432885248474058869292952127649993188722507718459666305388324601532051592215665909425735595882197577670726340725646
    459999590846534514050370793114900897670107649554189296242764912800345781503635840129133375880350805094211392297105783422610174413530444370929771190 13)
F = Point(980154959329070768640962584079889620073763288498998102503220023256253597359229376865333594555703692919999004762976944455578453688028307880629767608154 6
    723966128315709442518533754057884285184349717778060241532270622642626555158466333792037445888294881760457946028588478644021371507519618265365242653081399349 71,
    871661360542992726585345929824303616755203192060994999925292376639352466175619447164478311625616042775683976309200483763928060475584208919228134751247189678 8
    907432206174734178036892242539606146885146018586196443239240856176958846852418786171386564578362923777824279396698093857550091931091983893092436864205)

alpha, beta = 1, -2
PhiF, PhiG = Phi(alpha, beta, F.x, F.y) % p, Phi(alpha, beta, G.x, G.y) % p

flag = int(discrete_log(p, PhiF, PhiG))
print(flag.to_bytes(math.ceil(math.log(flag, 256)), 'big'))
```

Result :

```
b'FLAG{I_h3arD_th47_ECDlp_1s_h4rDEr_THaN_dlp}5\xff\xe4\xd4\xb6\xd5m\x92\x86\nn\x1b&c\x85\x14\xeaBOWxZ/\x89\xbd}6JZ6t\x
9aF\xd3\x84\x0c\x1a\x95\xf6\xe5&\x0c\xa2\xfe\xaf\x9c:\x05\xe0sc9\xfb+\xf5v\xcd\xeb\x11l_\xec\xe5\xb9\xcaH\x95\xdd\xf5P
\xd7~0L\xe4\x17\xfb'
```

Challenge Name : AES

Points :

Category : Crypto

Description :

Approach :

This is an AES encryption program. AES is a secure encryption algorithm. Nevertheless, no encryption system is entirely secure. We can do Side-channel attack, which is any attack based on extra information that can be gathered because of the way the algorithm is implemented, rather than algorithm itself. For example, hardware information during a computation.

In this challenge, it provides an AES encryption file and a json file that record 50 data, each containing a 16 bytes plaintext, corresponding ciphertext, and a 1806 length power trace array, with the *ith* value representing the power consumption at time *i*.

To get the AES key, we can use the CPA (correlation power analysis, a kind of power analysis). The rationale behind his work is that the power consumption would be larger when machines try to complement more bits, which means that the Hamming distance is larger. For example, the power consumption would be larger when transformatting 0000 to 1111 than when transformatting 0000 to 0001.

The approach is to repeat computing one byte for 16 times. For each byte, firstly, I take the corresponding bytes in plaintext from the 50 data, and enumerate all possible bytes of the key, which is 0x00~0xFF in this case, and then compute Sbox(plaintext ^ key) to get **hypothetical intermediate values**, and compute Hamming distances of all hypothetical intermediate values. The **hamming distance matrix** is a 50 * 256 matrix, while the value on *ith* row and *jth* column represents Hamming distances of the hypothetical intermediate value when the plaintext is equal to the *ith* data and the key is equal to *j*. Secondly, I compute the correlation coefficient of hamming distance and power trace. This construct a 256 * 1806 matrix which represents correlation coefficient of ith column in hamming distance matrix and jth column in power trace matrix. Finally, we can guess the row of the first or second largest value in the correlation coefficient matrix is the **key**.

The reason why I choose the value after AES doing XOR and SubBytes() and before doing Mixcolumn() as intermediate value is that XOR and SubBytes() are both independent of each byte's location, and thus we can compute each of the 16 bytes separately, which makes the process simpler.

Though we successfully get the correct key by the below script, the situation is more complex in real attack. In a real attack, one might easily face a predicament. For example, there could be too much noise in the power trace, or it could be hard to locate the sample window. If we get most of the bytes of the key using CPA, brute-force attack could solve the remaining bytes.

Code :

```python
# s-box matrix from encryption file
sbox = [0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16]

def hamming_distance(num):
    return bin(num).count("1")

data = json.load(open('stm32f0_aes.json'))

# K is 256 since value of each byte is between 0 to 255, T is the number of time point we have traced
D, K, T = len(data), 256, len(data[0]['pm'])
trace_matrix = np.array([data[i]['pm'] for i in range(D)])

# test one byte at a time, since each byte is independent before MixColumns
for byte in range(16):
    # hamming_distance_matrix has shape D * K
    hamming_distance_matrix = np.array([[hamming_distance(sbox[data[i]['pt'][byte]^j]) for j in range(K)] for i in range(D)])

    # coef_matrix has shape K * T
    coef_matrix = np.array([[np.corrcoef(hamming_distance_matrix[:,i], trace_matrix[:,j])[0][1] for j in range(T)] for i in range(K)])

    row, col = np.unravel_index(coef_matrix.argmax(), coef_matrix.shape)
    print(chr(row), end='')
```

Result :

```
18MbH9oEnbXHyHTR
```