Weekly Notes for CTF 🚩

Week *[REDACTED]*

Astrageldon

2023-11-24

1 Crypto

1.1 Discrete logarithm problems with smooth number factors

设 $n=\prod_i p_i$，而 p_i 是光滑数（smooth number, or N-smooth number），也即 $p_i=\prod_j^{n_i} p_{ij}+1\in\mathbb{P}$，其中 p_{ij} 是一些较小的质数（或小于 N 的质数）。现在，已知 g,h,p_i，要求出整数 x 使得 $g^x\equiv h \pmod n$，可以采取 Pohlig Hellman 提出的方法：

1. 求出 $m=\varphi(n)$ 的每一个因子 $q_i^{e_i}$。
2. 对于每个 i，在 0~$\frac{m}{q_i^{e_i}}$ 之间遍历 x_i，找到一个 x_i 使得 $(g^{x_i})^{\frac{m}{q_i^{e_i}}}\equiv h^{\frac{m}{q_i^{e_i}}} \pmod n$。
3. 使用中国剩余定理在模 $\prod_i q_i^{e_i}$ 的意义下求出满足同余方程组 $x\equiv x_i \pmod{q_i^e}$ 的一个 x。

上述算法的正确性是十分显然的，而在第二步中，可以利用大步小步算法（Shanks' Baby-step giant-step, BSGS），将时间复杂度优化到 $O(\sqrt{q_i^{e_i}})$，具体算法如下：

1. 计算出 $\alpha\equiv g^{\frac{m}{q_i^{e_i}}},\beta\equiv h^{\frac{m}{q_i^{e_i}}},N=q_i^{e_i}$是 α 在 $\mathbb{F}_n^*$ 中的阶数$,M=\lceil\sqrt{N}\rceil$。
2. 在 0~$M-1$ 之间遍历 j，用哈希表（Hash table）保存每个 $\alpha^j \mod n$ 对应的 j。
3. 令 $t=\alpha^{-M} \mod n,\gamma=\beta$。
4. 在 0~$M-1$ 之间遍历 i，如果 γ 是哈希表的一个键值，那么停止遍历并直接返回 $iM+j$ 作为找到的 x_i，否则将 γ 在模 n 的意义下乘以 t，并继续遍历。

上述算法的正确性是基于 $\alpha^{iM+j}\equiv\beta\Leftrightarrow\alpha^j\equiv\beta(\alpha^{-M})^i$。

图 1: Hash browns :) on Hash's table

```python
import gmpy2

def bsgs(alpha,beta,N,n):
    if N == 2:
        for j in range(3):
            if pow(alpha,j,n) == beta:
                return j
    tmp = {}
    M = gmpy2.iroot(N,2)
    M = M[0] if M[1] else M[0] + 1
    for j in range(M):
        tmp.update({pow(alpha,j,n): j})
    t = pow(alpha,-M,n)
    for i in range(M):
        if beta in tmp:
            return i*M + tmp[beta]
        else:
            beta *= t

def dlp(g,h,n,qe):
    # base, target result, modulus, (factor, exponent) pairs of phi(n)
    m = prod([pow(q,e) for q,e in qe])
    crt_remain = []
    crt_moduli = []
    for q,e in qe:
        N = pow(q,e)
```

```python
        k = m // N
        xi = bsgs(pow(g,k,n),pow(h,k,n),N,n)
        if xi is None or xi <= 1:
            continue
        crt_remain.append(xi)
        crt_moduli.append(N)
    x = crt(crt_remain,crt_moduli)
    return x
```

优化：上述算法当 $q_i^{e_i}$ 比较小时是管用的，反之，$O(\sqrt{q_i^{e_i}})$ 依然需要消耗大量的时间。

为此，考虑一个新的 DLP 问题：$g^x \equiv h \pmod n$，其中 g 的阶是 $q^e, e \geq 2$。我们可以将 x 写为 $x = \sum\limits_{k=0}^{e-1} x_k q^k$，其中 $0 \leq x_k < p$，计算 h^{q^r}，当 $r = e-1$ 时：

$$
\begin{aligned}
h^{q^{e-1}} &\equiv (g^x)^{q^{e-1}} \\
&\equiv g^{x_0 q^{e-1}} g^{q^e(x_1 + x_2 q + \cdots + x_{e-1} q^{e-2})} \\
&\equiv g^{x_0 q^{e-1}} \\
&\equiv (g^{q^{e-1}})^{x_0} \pmod n
\end{aligned}
$$

仿照前文，使用大步小步算法即可在 $O(\sqrt{\operatorname{ord}(g^{q^{e-1}})}) = O(\sqrt{q})$ 的时间复杂度内求出 x_0。而当 $r = e-2 \geq 0$ 时：

$$
\begin{aligned}
h^{q^{e-2}} &\equiv (g^x)^{q^{e-2}} \\
&\equiv g^{x_0 q^{e-2} + x_1 q^{e-1}} g^{q^e(x_2 + \cdots + x_{e-1} q^{e-3})} \\
&\equiv g^{x_0 q^{e-2} + x_1 q^{e-1}} \\
&\equiv (g^{x_0 q^{e-2}})(g^{q^{e-1}})^{x_1} \pmod n
\end{aligned}
$$

于是 $(g^{q^{e-1}})^{x_1} \equiv (h \cdot g^{-x_0})^{q^{e-2}} \pmod n$，照样可以在在 $O(\sqrt{q})$ 的时间复杂度内求出 x_1。最终，我们枚举 r，即可获得 e 个微型的 DLP 问题，它们都可以在 $O(\sqrt{q})$ 内解决，因此，我们在 $O(e\sqrt{q})$ 的时间复杂度内解决了上述 DLP 问题。经验告诉我们，Pohlig-Hellman 算法的第二步中 $g^{\frac{m}{q_i^{e_i}}}$ 的阶数很大概率上是 $q_i^{e_i'}$，其中 e_i' 是 $p-1, q-1, r-1$ 含有 q_i 因子个数的最大值，

且 $q_i \neq 2$。将 m 重新赋值 $m = \prod_i p_i^{e'_i}$ 后，该步骤中 x_i 的求解也可以通过上述方法进行优化。

```python
import gmpy2

def bsgs(alpha,beta,N,n):
    if N == 2:
        for j in range(3):
            if pow(alpha,j,n) == beta:
                return j
    tmp = {}
    M = gmpy2.iroot(N,2)
    M = M[0] if M[1] else M[0] + 1
    for j in range(M):
        tmp.update({pow(alpha,j,n): j})
    t = pow(alpha,-M,n)
    for i in range(M):
        if beta in tmp:
            return i*M + tmp[beta]
        else:
            beta *= t

def dlp(g,h,n,qe):
    # base, target result, modulus, (factor, exponent) pairs of
        phi(n)
    m = prod([pow(q,e) for q,e in qe])
    crt_remain = []
    crt_moduli = []
    for q,e in qe:
        N = pow(q,e)
        k = m // N
        xi = []
        G = pow(g,k,n)
        for r in range(e-1,-1,-1):
            H = pow(pow(h,k,n),pow(q,r),n)
            for l in range(e-r-1):
                H *= pow(G,-xi[l]*pow(q,r+1),n)
```

```
            xi.append(bsgs(pow(G,pow(q,e-1),n),H,q,n))
            if None in xi: break
        if None in xi: continue
        xi = sum([x*pow(q,l) for l,x in enumerate(xi)])
        if xi <= 1: continue
        crt_remain.append(xi)
        crt_moduli.append(N)
    x = crt(crt_remain,crt_moduli)
    return x
```

另：SageMath 中的 discrete_log 函数貌似就是使用 Pohlig-Hellman 算法与大步小步算法实现的，但是并不能直接手动输入因数，上述算法就权当是对其的一种改进吧。:)

用一个例子来说明下该算法的使用方法。

1.1.1 An Example :o

举个🌰，魔改自 Geek Challenge 2023 的 Diligent_Liszt：

```
import gmpy2 as gp
import random
from Crypto.Util.number import *

from secret import flag

DEBUG = 1

nbits = 1024
g = 3

gcd = GCD
```

```python

primes = []
pri = 1
while(len(primes)<100):
    pri = gp.next_prime(pri)
    primes.append(int(pri))

def gen_p_1(digit):
    while True:
        count = 2
        while count < 2**digit:
            p = random.choice(primes)
            count *= p**random.randint(1,10)
        count += 1
        if(gp.is_prime(count)):
            return count

def gen_p_3(digit):
    while 1:
        p,q,r = [gen_p_1(nbits) for _ in "pqr"]
        return p,q,r


p,q,r = gen_p_3(nbits)
n = p*q*r
x = bytes_to_long(flag)
y = gp.powmod(g,x,n)


print("p = {}".format(p))
print("q = {}".format(q))
print("r = {}".format(r))
print("y = {}".format(y))
print("n = {}".format(n))

if DEBUG:
    print("x0 = {}".format(x))
```

此问题在 $x < \min\{p,q,r\} = M$ 时可以通过求解

$$\begin{cases} 3^{x_p} \equiv y \pmod{p} \\ 3^{x_q} \equiv y \pmod{q} \\ 3^{x_r} \equiv y \pmod{r} \end{cases} \tag{1.1}$$

以及

$$\begin{cases} x \equiv x_p \pmod{p-1} \\ x \equiv x_q \pmod{q-1} \\ x \equiv x_r \pmod{r-1} \end{cases} \tag{1.2}$$

得到 x $(x < M)$。（事实上，求得的 $x_p, x_q, x_r = x$ 或 0）

反之，在 $n > x \geq M$ 时，方程组 (1.2) 的模数 $p-1, q-1, r-1$ 并不互质，不能直接使用中国剩余定理。那么，我们取其中两条同余方程，比如 $x \equiv x_p \pmod{p-1}, x \equiv x_q \pmod{q-1}$，将它们写为：$x = k_1(p-1) + x_p, x = k_2(q-1) + x_q$。即 $k_1(p-1) - k_2(q-1) = x_q - x_p$，由直觉可知，当 $\gcd(p-1, q-1) \nmid x_q - x_p$ 时这两个方程组成的方程组无解，否则可以用扩展欧几里得算法求出一组解。因此方程组 (1.2) 有解的条件是 x_p, x_q, x_r 全为奇数或者全为偶数。然而，遗憾的是，这种条件出现的概率并不是很高，对于同一组数据而言，能否成功解出方程组 (1.2) 是需要靠运气的。

我们尝试使用之前提到的优化后的 Pohlig-Hellman 算法：

```python
factors = {}
for i in sum([list(factor(xx)) for xx in [p-1,q-1,r-1]],[]):
        factors.update({i[0]: max(i[1],factors.get(i[0],0))})
factors = list(factors.items())
x = dlp(3,y,n,factors)
print(long_to_bytes(int(x)))
```

经验主义地来说，一般情况下，如果在 0~n 间不存在比明文 x_0 更小的 x 使得 $3^x \equiv y$，那么我们就能将 x_0 恢复出来。而有的时候 $3^{\frac{\varphi(n)}{q_i^{e_i}}}$ 的阶数不是 $q_i^{e'_i}$，如果可以将正确的阶数枚举出来，那么也能提高求解出正确的 x 的概率。

唔，折腾到这里已经是鄙人的极限了，就这样吧，改天研究下别的 DLP 算法。

1.2 Linear congruential generators

线性同余方法是用来生成伪随机数的一种很差劲的算法，为什么说它差劲呢？设置初值 X_0, a, b, m 以后，它给出一个序列 X_n：

$$X_{n+1} = (aX_n + b) \pmod{n}$$

那么我们只要获得连续的几项，就有可能直接恢复出 a, b, m 这三个参数，如果还知道这几项的序号，那么种子 X_0 也可以恢复出来。具体算法是：

$$t_n = X_{n+1} - X_n$$
$$m = \gcd(t_{n+1}t_{n-1} - t_n t_n, t_{n+2}t_n - t_{n+1}t_{n+1})$$
$$a = ((X_{n+2} - X_{n+1})(X_{n+1} - X_n)^{-1}) \mod m$$
$$b = (X_{n+1} - aX_n) \mod m$$
$$X_n = (a^{-1}(X_{n+1} - b)) \mod m$$

注意到我们恢复 m 的方法是求取最大公因数，因此得到的 m 与实际的 m 很可能差了一个倍数，但一般这个倍数往往不太大，因此一般可以通过枚举的方式将倍数消去得到真正的 m。我们于是得到了随机数生成器的全部状态，如果使用该生成器来分配扑克牌的话，可以通过计算预测出接下来将生成的所有牌面。