Pimeys CTF Challenge Writeup

Cubic Pell Curve Cryptosystem Attack

Cryptography Challenge Solution

Abstract

This writeup presents the solution to the `Pimeys` cryptography challenge. The challenge implements a public-key encryption scheme based on a twisted cubic (Pell) curve. We exploit a small private exponent vulnerability using Wiener's attack to recover the factorization of n, then solve a multivariate polynomial system to invert the encoding function and recover the flag.

Contents

1 Challenge Overview

1.1 Cryptosystem Description

The challenge implements a public-key cryptosystem with the following components:

1.1.1 Key Generation

- Generate two 512-bit primes p, q with $p \equiv q \equiv 1 \pmod 3$
- Compute $\phi = (p-1)(q-1)$
- Generate a small private exponent d of $\frac{512}{4} = 128$ bits
- Choose random $u, v \in \{1, 2\}$ and compute $n = p^u \cdot q^v$
- Choose random $x \in [1, n)$ and compute $y = x^3 \mod n$
- Compute bound $D = \lfloor n^{1/(2u+2v)} \rfloor$ and verify $d < D$
- Compute public exponent: $e = d^{-1} \mod \left(\phi^2 \cdot \frac{n^2}{p^2 q^2}\right)$

The public key is (n, x, e) and the private key is d.

1.1.2 Group Operations

The cryptosystem operates on a twisted cubic curve structure. Points are represented as triples (a, b, c) with operations:

Addition: For points $A = (a_1, b_1, c_1)$ and $B = (a_2, b_2, c_2)$:

$$u = a_1 a_2 + y(b_1 c_2 + b_2 c_1) \mod n \tag{1}$$
$$v = a_1 b_2 + a_2 b_1 + y c_1 c_2 \mod n \tag{2}$$
$$w = b_1 b_2 + a_1 c_2 + a_2 c_1 \mod n \tag{3}$$

Doubling: For point $A = (a, b, c)$:

$$u = a^2 + 2ybc \mod n \tag{4}$$
$$v = 2ab + yc^2 \mod n \tag{5}$$
$$w = b^2 + 2ac \mod n \tag{6}$$

Scalar Multiplication: $\mathrm{MUL}(t, A)$ computed via double-and-add.

1.1.3 Encoding Function

The message (m_1, m_2) is encoded to a curve point via:

1. Set $(u, v, w) = (m_1, m_2, 1)$
2. Compute:

$$a = u^3 + 2x^2 u(v^2 + xvw + x^2 w^2) + x^4 vw(v + xw) \mod n \tag{7}$$
$$b = x^2 v^3 + 2v(u^2 + x^2 uw + x^4 w^2) + xuw(u + x^2 w) \mod n \tag{8}$$
$$c = x^5 w^3 + 2xw(u^2 + xuv + x^2 v^2) + uv(u + xv) \mod n \tag{9}$$
$$r = u^3 + yv^3 + y^2 w^3 - 3yuvw \mod n \tag{10}$$

3. Output: $(X, Y, Z) = \left(\frac{a}{r}, \frac{b}{r}, \frac{c}{xr}\right) \mod n$

1.1.4 Encryption

$$C = \text{MUL}(e, \text{encode}(m_1, m_2))$$

1.2 Given Values

From the challenge output:

- n: 1536-bit modulus
- x: curve parameter
- e: 3069-bit public exponent
- $C = (C_0, C_1, C_2)$: ciphertext

2 Vulnerability Analysis

2.1 Small Private Exponent

The critical vulnerability is the small private exponent d (only 128 bits). The relationship:

$$e \cdot d \equiv 1 \pmod{M}$$

where $M = \phi^2 \cdot n^2/(p^2q^2)$, allows for a Wiener-style attack.

2.2 Determining u and v

Since n is 1536 bits and p, q are each 512 bits:

$$n = p^u \cdot q^v \approx 2^{512(u+v)}$$

This gives $u + v = 3$, so either $(u, v) = (1, 2)$ or $(u, v) = (2, 1)$.

3 Attack Strategy

3.1 Step 1: Wiener's Attack

We apply continued fraction expansion to e/n^2:

Listing 1: Wiener's Attack Implementation

```
def continued_fraction_expansion(num, den, max_terms=5000):
    cf = []
    while den != 0 and len(cf) < max_terms:
        q = num // den
        cf.append(q)
        num, den = den, num - q * den
    return cf

def convergents_from_cf(cf):
    n0, n1 = 0, 1
    d0, d1 = 1, 0
    for a in cf:
        n0, n1 = n1, a * n1 + n0
        d0, d1 = d1, a * d1 + d0
        yield n1, d1

```

```
# Attack: try convergents of e/n^2
cf = continued_fraction_expansion(e, n*n)
for k, d in convergents_from_cf(cf):
    if d == 0 or k == 0:
        continue
    ed_1 = e * d - 1
    if ed_1 % k != 0:
        continue
    quot = ed_1 // k
    g = gcd(quot, n)
    if 1 < g < n:
        # Found factor!
        factor = g
        break
```

This yields:

$$d = 244802100224204260927379249278011895877 \tag{11}$$

$$\text{factor} = p^2 = 115036392984235839663121648437024316404... \tag{12}$$

Taking the square root gives us p, and $q = n/p^2$:

$$p = 107255019921790066144201786965188669500 6... \quad (512 \text{ bits}) \tag{13}$$

$$q = 107391546112975051020595582453083903023 8... \quad (512 \text{ bits}) \tag{14}$$

Thus $n = p^2 \cdot q$ (so $u = 2, v = 1$).

3.2 Step 2: Decryption

With d recovered, we decrypt:

$$D = \text{MUL}(d, C) = (X, Y, Z)$$

This gives us the encoded point $D = \text{encode}(m_1, m_2)$.

3.3 Step 3: Inverting the Encoding

We need to recover (m_1, m_2) from (X, Y, Z).

3.3.1 Setting Up the Polynomial System

From the encoding, we have:

$$X = \frac{a}{r}, \quad Y = \frac{b}{r}, \quad Z = \frac{c}{xr}$$

This gives us the relations:

$$\frac{X}{Y} = \frac{a}{b}, \quad \frac{Y \cdot x}{Z} = \frac{b \cdot x}{c/x} = \frac{xb}{c/x}$$

Cross-multiplying:

$$Y \cdot a = X \cdot b \tag{15}$$

$$Y \cdot c = xZ \cdot b \tag{16}$$

Substituting the polynomial expressions for a, b, c in terms of m_1, m_2 (with $w = 1$):
Equation 1: $Y \cdot a(m_1, m_2) = X \cdot b(m_1, m_2)$
Equation 2: $Y \cdot c(m_1, m_2) = xZ \cdot b(m_1, m_2)$
Both equations are polynomial in m_1 and m_2 with:

- Equation 1: degree 3 in both m_1 and m_2
- Equation 2: degree 2 in m_1, degree 3 in m_2

3.3.2 Resultant Computation

We eliminate m_1 by computing the resultant:

$$\text{Res}_{m_1}(\text{eq}_1, \text{eq}_2) = P(m_2)$$

This yields a univariate polynomial of degree 9 in m_2.

3.3.3 Factorization over $\mathbb{F}_p$

Working modulo p, we factor $P(m_2)$:

Listing 2: Polynomial Factorization

```
res_poly = Poly(resultant(eq1, eq2, m1), m2, modulus=p)
factors = res_poly.factor_list()
# Result: 5 factors including 4 linear factors
```

The factorization reveals linear factors, each giving a candidate for $m_2 \mod p$.

3.3.4 Back-Substitution

For each candidate m_2, we substitute back into Equation 1 (now univariate in m_1) and solve.

3.3.5 Verification

We verify each (m_1, m_2) pair by checking:

$$\text{encode}(m_1, m_2) \stackrel{?}{=} D$$

3.4 Step 4: Message Reconstruction

The valid solution found:

$$m_1 = 360311846268017093125589856460176847703864055049682522277322611339255117135741508320757081 \tag{17}$$

$$m_2 = 942072153180164204618986201625891938402661669129163548840323978275828258306900488980914394 \tag{18}$$

The original message was split as decimal strings:

$$m = \text{bytes\_to\_long}(\text{flag})$$

$$m_1 = \text{int}(\text{str}(m)[:L/2]), \quad m_2 = \text{int}(\text{str}(m)[L/2:])$$

Reconstructing:

$$m_{\text{str}} = \text{str}(m_1) \| \text{str}(m_2)$$

$$\text{flag} = \text{long\_to\_bytes}(\text{int}(m_{\text{str}}))$$

4 Solution

Listing 3: Final Flag Recovery

```
from Crypto.Util.number import long_to_bytes

m1 = 360311846268017093125589856460176847 7...93072
m2 = 942072153180164204618986201625891938 4...49085

m_str = str(m1) + str(m2)
m_int = int(m_str)
flag = long_to_bytes(m_int)
print(flag)
```

5 Flag

```
ASIS{puBl!c_K3y_enCrYpT10n_5ch3m3_84s3d_0n_tH3_cu8!c_P3Ll_cuRv3_Us1ng_3nc0d!n9_fUnc7!0ns!!}
```

6 Summary

1. **Vulnerability:** Small private exponent ($d \approx 128$ bits) enables Wiener's attack
2. **Factorization:** Continued fractions on e/n^2 reveal d and a factor of n
3. **Decryption:** Scalar multiplication $D = \text{MUL}(d, C)$ recovers the encoded point
4. **Inversion:** Resultant computation and polynomial factorization over $\mathbb{F}_p$ solve for (m_1, m_2)
5. **Reconstruction:** Decimal string concatenation recovers the original flag

7 References

- Wiener, M. J. (1990). Cryptanalysis of short RSA secret exponents. *IEEE Transactions on Information Theory*, 36(3), 553-558.
- Boneh, D., & Durfee, G. (2000). Cryptanalysis of RSA with private key d less than $N^{0.292}$. *IEEE Transactions on Information Theory*, 46(4), 1339-1349.
- Washington, L. C. (2008). *Elliptic Curves: Number Theory and Cryptography*. CRC Press.