2024-09-18

Write-Up

Triple Curves

Contents

Ectario

1 Chall informations

1.1 Script

From the `chall.py`:

```
from Crypto.Util.number import *
from flag import FLAG

FLAG = bytes_to_long(FLAG)

p = 2^222 - 117 # strong prime from https://safecurves.cr.yp.to/
F1 = FiniteField(p)
E1 = EllipticCurve(F1, [0, 1, 0, 1, 3])

q  = 2^221 - 3 # strong prime from https://safecurves.cr.yp.to/
F2 = FiniteField(q)
E2 = EllipticCurve(F2, [0, 1, 0, 1, 3])

r = 2^224 - 2^96 + 1 # strong prime from https://safecurves.cr.yp.
to/
F3 = FiniteField(r)
E3 = EllipticCurve(F3, [0, 1, 0, 1, 3])

P = E1.random_point()
Q = E2.random_point()
R = E3.random_point()

s = (int(P[0]) * int(Q[0]) * int(R[0]))

print(f"{F1(s)=}")
print(f"{F2(s)=}")
print(f"{F3(s)=}")

s += int(R[0])

assert s > FLAG

print(f"s XOR flag={s ^^ FLAG}")
print(f"{R[1]=}")
```

1.2 Known infos

Goal: The challenge generates three elliptic curve points P, Q, and R from three different finite fields. The goal is to recover the flag, which has been XORed with a masked value s, derived from the x-coordinates of these points.

Informations from `chall.py`:

```
p = 2^222 - 117
q = 2^221 - 3
r = 2^224 - 2^96 + 1

print(f"{F1(s)=}") # i.e. s % p
print(f"{F2(s)=}") # i.e. s % q
print(f"{F3(s)=}") # i.e. s % r

assert s > FLAG    # i.e. the xor will be a one-time pad

print(f"s XOR flag={s ^^ FLAG}")
print(f"{R[1]=}")
```

The x-coordinates of the points P, Q, and R are multiplied together to produce s, and then masked. Our goal is to reverse this process and recover the original flag.

2 Analyse

2.1 Key Points

1. Three Strong Primes: The primes p, q, and r used to define the finite fields are strong primes chosen from secure cryptographic sources. These primes are also pairwise co-prime, which is important for the next step in solving the challenge using the Chinese Remainder Theorem (CRT).

2. CRT Application: Since p, q, and r are pairwise co-prime, we can apply the Chinese Remainder Theorem to reconstruct s modulo $p * q * r$. This property allows us to uniquely determine s because the product $P[0] * Q[0] * R[0]$ will be smaller than $p * q * r$, making s recoverable.

3. XOR Masking: The script adds $R[0]$ to s and then XORs the result with the flag. Once s is recovered, reversing the XOR operation will allow us to recover the original flag.

3 How to solve

3.1 Recovering s

To solve this challenge, we need to reverse the process:

1. The first step is to recover s modulo p, q, and r. These values are given as `F1(s)`, `F2(s)`, and `F3(s)` in the output.

2. Since p, q, and r are prime and co-prime, we can apply the Chinese Remainder Theorem to reconstruct s modulo $p * q * r$. This works because the values of $P[0]$, $Q[0]$, and $R[0]$ are smaller than their respective field moduli, ensuring that s is fully recoverable.

3. Once we have s, we add $R[0]$ to match the value before the XOR operation, and then XOR the result with the value stored in `s XOR flag` to recover the flag.

3.2 Solve script

```
from Crypto.Util.number import *

p = 2^222 - 117
F1 = FiniteField(p)
E1 = EllipticCurve(F1, [0, 1, 0, 1, 3])
q  = 2^221 - 3
F2 = FiniteField(q)
E2 = EllipticCurve(F2, [0, 1, 0, 1, 3])
r = 2^224 - 2^96 + 1
F3 = FiniteField(r)
E3 = EllipticCurve(F3, [0, 1, 0, 1, 3])

s_xor_flag, F1_s, F2_s, F3_s, yR = None, None, None, None, None

with open("output.txt", 'r') as file:
    for line in file:
        line = line.strip()
        if line.startswith("s XOR flag="):
            s_xor_flag = int(line.split("=")[1])
        elif line.startswith("F1(s)="):
            F1_s = int(line.split("=")[1])
        elif line.startswith("F2(s)="):
            F2_s = int(line.split("=")[1])
        elif line.startswith("F3(s)="):
            F3_s = int(line.split("=")[1])
        elif line.startswith("R[_sage_const_1 ]="):
            yR = int(line.split("=")[1])

polR = E3.defining_polynomial()
xRs                                                                  =
polR(y=yR,z=1).univariate_polynomial().roots(multiplicities=False)

for x in xRs:
    s = int(CRT([F1_s, F2_s, F3_s], [p, q ,r]))
    s += int(x)
    flag = long_to_bytes(s_xor_flag ^^ int(s))
    print(flag) if b"}" in flag else ...
```

3.3 Explanation

Chinese Remainder Theorem: By leveraging the fact that p, q, and r are co-prime, we can apply the CRT to reconstruct s modulo $p * q * r$, which directly gives us the value of s. This is feasible because the product $P[0] * Q[0] * R[0]$ is smaller than $p * q * r$.

$$\begin{cases} s = \mathrm{F1}(s) \bmod p \\ s = \mathrm{F2}(s) \bmod q \\ s = \mathrm{F3}(s) \bmod r \end{cases}$$

$$\Rightarrow \{\mathrm{CRT}([\mathrm{F1}(s), \mathrm{F2}(s), \mathrm{F3}(s)], [p, q, r]) = s \bmod (p * q * r)$$

And finally:

$$P[0] * Q[0] * R[0] < p * q * r \Rightarrow s \bmod (p * q * r) = s$$

Recovering the Flag: After recovering s, we need to add $R[0]$ to reverse the addition in the original script. Once done, we can XOR the result with the masked flag value `s XOR flag` to recover the original flag.

Note: To recover R x-coordinate, (needed to recover s before `s += int(R[0])`), the code extracts the y-coordinate y_R of the elliptic curve point R and then computes the possible x-coordinates that satisfy the curve's equation. Mathematically, this is equivalent to solving the elliptic curve equation for x given $y = y_R$ and $z = 1$ in $-x^3 - x^2 * z + y^2 * z - x * z^2 + 26959946667150639794667015087019630673557916260026308143510066298878 * z^3 = 0$. So by substituting $y = y_R$ and $z = 1$ (the z value is used for projective elliptic curves), the code transforms the elliptic curve equation into an univariate polynomial in x and finds the roots (possible x-values) that correspond to y_R. These roots are stored in `xRs` for further iteration.

4 Conclusion

This challenge highlights the power of the Chinese Remainder Theorem in modular arithmetic, especially when working with cryptographic systems based on finite fields. By correctly reconstructing s through CRT and reversing the XOR operation, we are able to successfully recover the flag.