2024-09-18

Write-Up

Lifting Secrets from Elliptic Curves

Contents

Ectario

1 Chall informations

1.1 Script

From the `chall.py`:

```python
from sage.all import *
from Crypto.Util.number import bytes_to_long, inverse
from flag import FLAG
from random import randrange

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

assert E.order().is_prime()

texts = bytearray(FLAG)
mask = 0x1fffffffffffff
r = randrange(1, p)
for i, x in enumerate(texts):
    lifted = False
    added = 0
    while not lifted:
        try:
            P = E.lift_x(F(x))
            Q = r*P
            print(f"{added}: {int(Q.xy()[0])^mask}")
            lifted = True
        except:
            x+=1
            added+=1
```

1.2 Known infos

Goal: The flag is being encoded through elliptic curve point multiplications and masking. We need to break the encoding to retrieve the original flag.

Informations from `chall.py`:

```
p = 2^222 - 117
mask = 0x1fffffffffffff
r = randrange(1, p)
```

The challenge involves an elliptic curve defined over the finite field F_p, where p is supposed to be a "strong prime." The flag bytes are converted to elliptic curve points and then obfuscated using a random scalar r and a mask. Our goal is to recover the original flag from this transformation.

2 Analyse

2.1 Key Points

1. Elliptic Curve: The curve used is defined by the equation $y^2 = x^3 + x^2 + x + 3$. The challenge gives the assurance that the curve's order is a prime number (`assert E.order().is_prime()`), which implies that all non-zero elements of the group are invertible. This property will be crucial when trying to undo the multiplication by r.

2. Prime Field used: The value of p is written as `2^222 - 117`. This suggests a prime number used as the field modulus. However, a common pitfall in Python is that `^` denotes bitwise XOR, not exponentiation. Therefore, the value of p as written in the code is not as large as expected. Instead of being around 2^{222}, it becomes a much smaller number, making discrete logarithm attacks feasible.

3. Point Lifting and Masking: For each byte of the flag, the script attempts to "lift" the byte into an elliptic curve point using `E.lift_x()`. After that, the point is multiplied by r and the x coordinate is masked using `0x1ffffffffffff`. The process is repeated until the byte can successfully be lifted.

3 How to solve

3.1 Recovering r

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

1. The first step is to retrieve the random scalar r. Since we know that the order of the curve is prime and every element is invertible, we can compute the inverse of r if we can determine it.

2. We also know a part of the flag structure, such as that it begins with `"3C{"`. This information helps us guess one of the bytes and use it to compute r. Specifically, by lifting the known byte and comparing the result to the output from the script, we can perform a discrete logarithm to recover r.

3. Once we have r, we can reverse the point multiplication and retrieve the original flag bytes.

3.2 Solve script

```
from sage.all import *
from Crypto.Util.number import long_to_bytes, inverse, bytes_to_long
from random import randrange

p = 2^222 - 117
F = FiniteField(p)
E = EllipticCurve(F, [0, 1, 0, 1, 3])

assert E.order().is_prime()

outputs = open("output.txt", "r").readlines()
ciphered = [int(x.split(": ")[1]) for x in outputs]
added = [int(a.split(": ")[0]) for a in outputs]

mask = 0x1ffffffffffff

# # Perform discrete log to recover r using known part of the flag
xP = bytes_to_long(b"3") + added[0]
unmasked_x = F(ciphered[0]^mask)
unmasked_Q = E.lift_x(unmasked_x)
r2 = discrete_log(unmasked_Q, E.lift_x(F(xP)), operation="+")

# Brute-force method to recover r and reconstruct the flag
for r in range(1, p):
    flag = b""
    try:
        for i, x in enumerate(ciphered):
            unmasked_x = F(x^mask)
            unmasked_Q = E.lift_x(unmasked_x)
            r_inv = inverse(r, E.order())
            P = F(r_inv) * unmasked_Q
            x_value = F(P[0]) if F(P[0]) > 40 else int(P[0]) + p
            flag += long_to_bytes(int(x_value) - int(F(added[i])))
        if b"}" in flag:
            # Verify that both methods give the same result
            assert r == r2
            print(flag)
            exit()
    except ZeroDivisionError:
        continue
```

3.3 Explanation

Discrete Logarithm Attack: We leverage the fact that the modulus `p` is smaller than expected due to the XOR mistake. This allows us to compute the discrete logarithm and recover r more easily than intended.

Brute-forcing r: As an alternative method, we brute-force r by iterating over possible values and reversing the point multiplication to recover the original bytes.

Finding x-coordinate:

Let P be the original elliptic curve point, and let r be a random scalar used to compute Q:

$$Q = r.P$$

Given Q, our goal is to recover P. Since the challenge guarantees that the order of the elliptic curve is prime, the scalar r is invertible. Thus, we can find r^{-1} (the inverse of r) such that $r.r^{-1} = 1$.

$$Q = r.P \Leftrightarrow r^{-1}.Q = r^{-1}.(r.P) \Leftrightarrow r^{-1}.r.P = P$$

Thus, we recover the original point P:

$$P = r^{-1}.Q$$

Since the point P is now recovered, we can easily access its x-coordinate (denoted x_P) by extracting the first component of P

Note: The line `x_value = [...]` checks whether the x-coordinate of point P, $P[0]$, is greater than 40. If it is, the value is directly interpreted as the ASCII code. If it's less than 40, it means the value was wrapped around due to the modulo p (since $p = 107$ and ASCII characters like "`{`" are larger than 107). In this case, p is added to reverse the wrap and recover the correct value.

Final Check: Once we recover the flag, we assert that the two methods (discrete logarithm and brute-force) yield the same r.

4 Conclusion

This challenge highlights the importance of correctly defining prime fields in cryptographic algorithms and demonstrates how small mistakes, such as the use of bitwise XOR instead of exponentiation, can lead to vulnerabilities like breaking the discrete logarithm problem.