Cry Me A River - Write up

This is a write-up for the crypto challenge *Cry Me A River* from FCSC 2023.
When we connect to the service, we are given:

- A public key on the elliptic curve P-521;
- A message (24 bytes);
- An ECDSA signature of the message.

Then, we are asked to submit another signature! According to the source of the challenge, we cannot chose the message, and it must be signed with the same private key which is unknown.
The goal of the challenge becomes clear: we must retrieve the private key from the public key and a single signature, then use it to sign a specific message.

Observations from the source code

The code is in Python and uses the `fastecdsa` module for elliptic curve computation, but ECDSA has been rewritten.
We can observe the following:

- The elliptic curve P-521 is used;
- The private key is 48 bytes long, so 384 bits (first alert, it's way less than 521);
- The nonce is generated from a LCG (linear congruential generator), which is not suited for cryptographic purpose (second (big) alert).

Nonce generation

The LCG used generates 128-bit outputs, each one completely determined by the previous one: if X_i is an output, then the next one is:

$$X_{i+1} = aX_i + c \bmod 2^{128}$$

The nonce is generated from **four** subsequent outputs:

$$k = X_3 + X_2 2^{128} + X_1 2^{256} + X_0 2^{384}$$

The whole nonce is completely determined from X_0 only, so it has only 128 bits of entropy.
A good lattice should be able to help.

Why a single signature is enough

An ECDSA signature of a hash h of a message under private key d is (r, s) such that we have the relation

$$ks \equiv h + dr \pmod{n}$$

where n is the prime order of the elliptic curve P-521.
As such, we can express d as a function of the nonce and other elements of the signature:

$$d = (ks - h)/r \bmod n$$

Theoritically, if one could run through all possible values for k, we would get at most 2^{128} candidates for the private key d, which is much less by several orders of magnitude compared to n (around 2^{521}).

On another hand, we know from the source code that the private key is less than 2^{384}. Under the assumption that each candidate for the private key are uniformly distributed in the range $[1, n-1]$, then the probability that a wrong candidate is smaller than 2^{384} is near 0. So, we can expect that only value that small is the private key.

Therefore, a single signature is sufficient to uniquely determine the private key.

> Of course, one could say that you could always use a single signature since for each potential nonce k', we can check if $x([k']G)$ matches with r. But for lattices attack, we cannot use calculations on elliptic curves!

Lattice attack

The idea is to construct a basis matrix for a lattice that contains a short vector related to the nonce and private key.

We already have a few relations between the unknowns:

$$\begin{cases} ks - h - dr + \lambda_0 n = 0 \\ X_1 - aX_0 - c + \lambda_1 2^{128} = 0 \\ X_2 - aX_1 - c + \lambda_2 2^{128} = 0 \\ X_3 - aX_2 - c + \lambda_3 2^{128} = 0 \end{cases}$$

The four zeros on the right side could be coordinates of our short vector.

The first four columns of the matrix integrate the four relations, then constraints on the unknown are added so that other coordinates of the short vector are small:

$$M = \begin{bmatrix} n & & & & & & & & & \\ & 2^{128} & & & & & & & & \\ & & 2^{128} & & & & & & & \\ & & & 2^{128} & & & & & & \\ -s2^{384} & -a & & & 1/2^{128} & & & & & \\ -s2^{256} & 1 & -a & & & 1/2^{128} & & & & \\ -s2^{128} & & 1 & -a & & & 1/2^{128} & & & \\ -s & & & 1 & & & & 1/2^{128} & & \\ r & & & & & & & & 1/2^{384} & \\ h & -c & -c & -c & & & & & & 1 \end{bmatrix}$$

Now, consider the vector u below:

$$u = (\lambda_0, \lambda_1, \lambda_2, \lambda_3, X_0, X_1, X_2, X_3, d, 1)$$

If we multiply M by this vector on the left, we get the vector $v = uM$ below (see it as scalar product with each column):

$$v = \left(0, 0, 0, 0, \frac{X_0}{2^{128}}, \frac{X_1}{2^{128}}, \frac{X_2}{2^{128}}, \frac{X_3}{2^{128}}, \frac{d}{2^{384}}, 1\right)$$

Each coordinate of v are in $[0, 1]$, which is indeed quite small!

If we apply the LLL algorithm on M, we might find this vector on one of its rows

Code

The whole code is given below in Python3, with `fpylll`, `fastecdsa` as dependencies (and `pwn` to connect to the service).

```
#!/usr/bin/env python3

from fpylll import IntegerMatrix, LLL
from hashlib import sha512
from fastecdsa.curve import P521
from fastecdsa.point import Point
from pwn import *
```

```python
import sys

lcg_a = 200565257846616591441313188858237974233
lcg_c = 1
n = P521.q # curve prime order

def construct_matrix(m, r, s):
    h = int.from_bytes(sha512(m).digest(), 'little')
    M = IntegerMatrix(10, 10)

    # First column:
    # Signature equation is h + d*r - k*s + L0*n = 0
    M[0, 0] = n
    M[4, 0] = -2**384*s
    M[5, 0] = -2**256*s
    M[6, 0] = -2**128*s
    M[7, 0] = -s
    M[8, 0] = r
    M[9, 0] = h

    # Second column:
    # Equation X1 - a*X0 - c + L1*2^128
    M[1, 1] = 2**128
    M[4, 1] = -lcg_a
    M[5, 1] = 1
    M[9, 1] = -lcg_c

    # Third column:
    # Equation X2 - a*X1 - c + L2*2^128
    M[2, 2] = 2**128
    M[5, 2] = -lcg_a
    M[6, 2] = 1
    M[9, 2] = -lcg_c

    # Fourth column:
    # Equation X3 - c1*X0 - c0 + L3*2^128
    M[3, 3] = 2**128
    M[6, 3] = -lcg_a
    M[7, 3] = 1
    M[9, 3] = -lcg_c

    # Multiply by 2**384 to avoid factors in following columns
    for i in range(10):
        for j in range(4):
            M[i, j] *= 2**384

    M[4, 4] = 2**256 # 2^384 * 1/2^128
    M[5, 5] = 2**256 # 2^384 * 1/2^128
    M[6, 6] = 2**256 # 2^384 * 1/2^128
    M[7, 7] = 2**256 # 2^384 * 1/2^128
    M[8, 8] = 1      # 2^384 * 1/2^384
    M[9, 9] = 2**384 # 2^384 * 1

    return M

def find_privkey(pubkey, m, r, s):
    M = construct_matrix(m, r, s)
    LLL.reduction(M)
    for i in range(10):
        guess = int(abs(M[i, 8]))
        if guess*P521.G == pubkey:
            return guess
    return None

def ecdsa_sign(privkey, m):
    # k = 1
    h = int.from_bytes(sha512(m).digest(), 'little')
    r = P521.G.x
    s = (h + privkey*r) % n
    return r, s
```

```python
def solve_challenge():
    ss = remote('challenges.france-cybersecurity-challenge.fr', 2151)

    # Parse public key
    tmp = ss.recvline().decode()[17:-2].split(',␣')
    xQ = int(tmp[0])
    yQ = int(tmp[1])
    pubkey = Point(xQ, yQ, curve=P521)

    # Parse message and signatures
    data = ss.recvline() # "Here is a valid signature"
    m = bytes.fromhex(ss.recvline().decode().split('␣=␣')[1][2:])
    tmp = ss.recvline().decode()[7:-2].split(',␣')
    r = int(tmp[0])
    s = int(tmp[1])

    # Use LLL to find the private key
    privkey = find_privkey(pubkey, m, r, s)
    if privkey is None:
        ss.close()
        sys.exit()

    # Forge signature
    m = b'All␣right,␣everybody␣be␣cool,␣this␣is␣a␣robbery!␣Give␣me␣the␣flag!'
    forged_sig = ecdsa_sign(privkey, m)

    # Send forged signature and get the flag
    data = ss.recvline() # "Your turn! Give me another one!"
    data = ss.recv(4)    # "r = "
    ss.send(str(forged_sig[0]).encode() + b'\n')
    data = ss.recv(4)    # "s = "
    ss.send(str(forged_sig[1]).encode() + b'\n')
    data = ss.recvline()
    print(f'Flag:␣{data.decode()}')

    ss.close()

if __name__ == "__main__":
    solve_challenge()
```