Existential

Category: Crypto
Difficulty: Medium
Author: rugo
Writeup by: Nils

Description

> Some forgeries are existential. Luckily, this can't happen here.

Summary

The author provided a python file with the callenge code. It generates a random ECDSA key and lets us sign up to 69 chosen messages that do not contain the string `flag`. The flag is printed if one finds a valid signature and corresponding message that contains the string `flag`.

Solution

There is a suspicious comment in the challenge file:

```python
def efficient_k(msg):
    # Make semi-deterministic to not exhaust the entropy pool too fast
    return int.from_bytes(
        shake_128(msg).digest(16) + os.urandom(16),
        "big"
    )
```

Knowing the `msg`, the 128 most significant bits of this `k` can be computed. The remaining least significant 128 are unknown.

ECDSA

The challenge uses libraries implementing the Elliptic Curve Digital Signature Algorithm. Following the chain of definitions on Wikipedia, this is a public-key cryptosystem to certify the authenticity of a message.

The algorithm is presented on Wikipedia roughly as this:

- The signing party picks an elliptic curve and a point on that curve. There are predefined curves and points to choose from. Another property of such a predefined curve and point is a public modulus q.
- The signing party picks a private key or exponent randomly from $\mathbb{Z}_q$. This key is used for all signatures.

So far, the binary does exactly that by using a library to generate a private key for a curve called `BRAINPOOLP256r1`.

- For each signature, the signing party selects a **cryptographically secure random** integer nonce k from $[1, q-1]$.
- It then calculates a value $r \mod q$ based on the curve and k. The details are not important for this challenge. r is the first part of the signature.
- The second part is calculated as $s \equiv k^{-1} \cdot (h + r \cdot d) \mod q$, where h is derived from a digest of the message to sign.
- The signature is (r, s).

Later the article also mentions k again:

> Another way ECDSA signature may leak private keys is when k is generated by a faulty random number generator. Such a failure in random number generation caused users of Android Bitcoin Wallet to lose their funds in August 2013.

So this is a scheme to non-deterministically generate signatures and a “semi-deterministic” choice of k could render the scheme vulnerable.

Ducking[1] for similar problems[2], I found some ideas that solve the related problem for some known significant bits and unknown most significant. I could have looked further and would have probably found a script solving the challenge but decided to use it as learning opportunity and derive a solution for the problem at hand from the one for that other problem.

The first step is to collect some signature samples signed with the secret key. It does not really matter if the messages for these samples differ because the value for `k` is partly random, generating distinct signatures for the same message.

Let m be the message, n be the number of samples, q be the modulus, d be the unknown private key and for the i-th sample let (R_i, S_i) be the known signature and K_i be the unknown nonse `k` used to generate this signature.
Let $a =$ shake_128(m) be the known part of each K_i and B_i be the unknown part with bit length

[1]https://duckduckgo.com
[2]https://crypto.stackexchange.com/a/48379

$l = 128$.
K_i can now be expressed as $K_i = a \cdot 2^l + b$, where $0 \leq a, B_i < 2^l$.

The algorithm used for the message digest is `sha1` and can be found as the default parameter in `SigningKey.generate`. Finally, let $h = \text{sha1}(m)$ be the the message digest for the signature.

Wikipedia tells us that $S_i \equiv K_i^{-1} \cdot (h + R_i \cdot d) \mod q$, which is

$$S_i \equiv (a \cdot 2^l + B_i)^{-1} \cdot (h + R_i \cdot d) \mod q$$

Everything except d and B_i is known. Hoping to find a solution similar to the one of the related problem, I rearranged the equation to isolate B_i.

$$\begin{aligned}
S_i &\equiv (a \cdot 2^l + B_i)^{-1} \cdot (h + R_i \cdot d) && \mod q \mid \cdot(a \cdot 2^l + B_i) \\
(a \cdot 2^l + B_i) \cdot S_i &\equiv h + R_i \cdot d && \mod q \mid \cdot S_i^{-1} \\
a \cdot 2^l + B_i &\equiv S_i^{-1} \cdot (h + R_i \cdot d) && \mod q \mid -a \cdot 2^l \\
B_i &\equiv S_i^{-1} \cdot (h + R_i \cdot d) - a \cdot 2^l && \mod q \\
B_i &\equiv \underbrace{S_i^{-1} \cdot R_i}_{X_i} \cdot d + \underbrace{S_i^{-1} \cdot h - a \cdot 2^l}_{Y_i} && \mod q
\end{aligned}$$

Thus I know $X_i \equiv S_i^{-1} \cdot R_i$ and $Y_i \equiv S_i^{-1} \cdot h - a \cdot 2^l$ must satisfy

$$B_i \equiv X_i \cdot d + Y_i \mod q$$

B_i is small, between 0 and 2^{128} compared to the integer ring modulo q which spans 2^{256} elements. Therefore, a vector of B is very short compared to a random vector in this field.

Finding B is an instance of the shortest vector problem.

Lucky for me, the details are not important, just that given an integer basis, a non-zero vector with integer coefficients in that basis can be found that minimizes its euclidian norm (length) compared to other vectors with integer coefficients in the basis.

In this challenge, I want some B that is as close as possible to the single vector from $\{2^{l-1}\}^n$, the center of $\mathbb{Z}_{2^l}^n$, which is the space of possible B. If it is within this space, it is likely the correct B.

The basis for the shortest vector problem has as many dimensions as the number of samples I collected and is composed of X and Y such that one of the coefficients is d and the other 1.

The shortest vector problem does not consider modulus. We can make it, though, by adding all vectors with one coordinate q and all others 0 to the basis, allowing the vectors to “wrap around” in the space $\mathbb{Z}_q^n$.

To seach for the closest vector to $\{2^{l-1}\}^n$ instead of the zero vector, the equation above is shifted by -2^{l-1}.

The equation to minimize looks like this, where $?$ are unknown integer coefficients to implement modulus and 1 and d happen to be the values minimizing it:

$$\begin{bmatrix} q & 0 & \dots & 0 & X_1 & Y_1 - 2^{l-1} \\ 0 & q & \dots & 0 & X_2 & Y_2 - 2^{l-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ & & & q & X_n & Y_n - 2^{l-1} \end{bmatrix} \times \begin{bmatrix} ? \\ \vdots \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} B_1 - 2^{l-1} \\ B_2 - 2^{l-1} \\ \vdots \\ B_n - 2^{l-1} \end{bmatrix}$$

Using sagemath, a candidate for B can be computed. From that, the secret K and thus the private key d can be derived:

$$\begin{aligned} S_i &\equiv K_i^{-1} \cdot (h + R_i\dot{d}) && \mod q \mid \cdot K_i \\ S_i \cdot K_i &\equiv h + R_i\dot{d} && \mod q \mid -h \\ S_i \cdot K_i - h &\equiv R_i\dot{d} && \mod q \mid \cdot R_i^{-1} \\ (S_i \cdot (a \cdot 2^l + B_i) - h) \cdot R_i^{-1} &\equiv d && \mod q \end{aligned}$$

All values on the left hand side of the equation are known. Thus the private key can be computed and used to sign the message `flag`, then I can exchange the signature for the flag.

I went for 5 samples because they are enough and with more, the library I used to solve the shortest vector problem occasionally crashed.

Exploit Code

```python
import ecdsa
import ecdsa.curves
import ecdsa.util
from hashlib import shake_128, sha1
from sage.all import matrix, vector
from sage.modules.free_module_integer import IntegerLattice
from pwn import process

sample_message = b''
n = 5

curve = ecdsa.curves.BRAINPOOLP256r1
h = int.from_bytes(sha1(sample_message).digest(), 'big')
a = int.from_bytes(shake_128(sample_message).digest(16), 'big')
l = 8 * 16
q = int(curve.order)
```

```python
offset = 2 ** (l-1)
offset_vector = vector([offset] * n)

Samples = []

# p = process(['python', './main.py']) # local testing
p = process('ncat --ssl id-existential.challenge.master.cscg.live 31337'.
    split(' '))
p.recvuntil(b'Choice >')

for _ in range(n):
    p.sendline(b'1')
    p.sendlineafter(b'Message (hex):', sample_message.hex().encode())

    p.recvuntil(b'Signature (hex): ')
    signature = p.recvline().decode()
    sig = bytes.fromhex(signature)
    Samples.append(sig)

    p.recvuntil(b'>')

R, S = zip(*(ecdsa.util.sigdecode_string(sig, q) for sig in Samples))

S_inv = [pow(s, -1, q) for s in S]

X = vector([(s_inv * r) % q for (s_inv, r) in zip(S_inv, R)])
Y = vector([(s_inv * h - (a << l)) % q for s_inv in S_inv])

M = matrix((matrix.identity(n) * q).rows() + [X] + [Y - offset_vector])
L = IntegerLattice(M)

B = L.shortest_vector() + offset_vector
K = [(a << l) + b for b in B]
D = [(s * k - h) * pow(r, -1, q) % q for s, r, k in zip(S, R, K)]

print(f'Secret exponent: {D[0]}')
privkey = ecdsa.SigningKey.from_secret_exponent(D[0], curve)
message = b'flag'
signature = privkey.sign(message)

print(f'Message: {message.hex()}')
print(f'Signature: {signature.hex()}')

p.sendline(b'2')
p.recvuntil(b'Message (hex):')
p.sendline(message.hex().encode())
p.recvuntil(b'Signature (hex):')
p.sendline(signature.hex().encode())
p.recvuntil(b'You won a bounty!')
lines = p.recvlines(2)
```

```
flag = lines[1].decode()
print(f'FLAG: {flag}')
```

Mitigation

- Do not try to save resources without evaluating the effects on cryptographic schemes. In this case, adhere to the definition and pick a k cryptographically securely random in $\mathbb{Z}_q$.
- Don't do your own crypto. The `sign` method does not require the parameter `k` to be set and can generate a sufficiently random one on its own.

Flag

`CSCG{OwNowHowDidYouDoThat}`