Project 4: Implement and Attack ECDSA with Repeated Nonce

Chaoyun Li

November 2022

The goal is to implement ECDSA and then mount a key recovery attack by using your implementation to generate two message/signature pairs.

1 Attacks on ECDSA with Repeated Nonce

Assume a lazy signer always uses the same nonce for ECDSA signatures. Then you can readily recover the private key by collecting two message/signature pairs.

Let E be an elliptic curve over a prime field $\mathbb{F}_p$, G a point in E of order n, where n is a prime number. Let $H(\cdot)$ be a hash function. The ECDSA signature algorithm is given below.

Algorithm 1: ECDSA signature

Input : the message m, private key d
Output: the signature (r, s)

1 $e \leftarrow H(m)$
2 $k \xleftarrow{\$} \{1, \cdots, n-1\}$; `/* Nonce` k `*/`
3 $R = (x_R, y_R) \leftarrow [k]G$
4 $r \leftarrow x_R \bmod n$
5 $s \leftarrow k^{-1}(e + rd) \bmod n$
6 **if** $r = 0$ *or* $s = 0$ **then**
7 | Go to step 2
8 **end**
9 Return $(r,\ s)$

Assume that one gets two triples (m_0, r_0, s_0) and (m_1, r_1, s_1) from the same unknown nonce k. Then one has the equations :

$$\begin{cases} s_0 = & k^{-1}(e_0 + r_0 d) \bmod n \\ s_1 = & k^{-1}(e_1 + r_1 d) \bmod n \end{cases}$$

Note that $e_i = H(m_i)$ are known. It is easy to solve the above equations as the following

$$\begin{cases} k = & (s_0 - s_1)^{-1}(e_0 - e_1) \bmod n \\ d = & (s_0 k - e_0) r_0^{-1} \bmod n \end{cases}$$

Therefore, the private key d can be determined. This very attack was used by hackers to extract the master private key from the Sony PlayStation 3 (PS3) in 2010.

2 Task 1

Implement ECDSA signature and verification. You are supposed to implement the following functions:

```
def ecdsa_sign() # given message m, return signature (r,s)
def ecdsa_verify() # given message m and signature (r,s), return Ture or False #
```

The parameters include a 256-bit private key byte-string d, finite field $\mathbb{F}_p$ with prime $p \approx 2^{256}$, curve order $n \approx 2^{256}$ (also prime), generator G. Some parameters are from `https://www.secg.org/sec2-v2.pdf` under *Section 2.4.2 Recommended Parameters secp256r1*. See also the attachment in Moodle. The hash function should be `sha256()`, which is available in `hashlib`.

Note that you can reuse the function `modinv()` for Project 4 to compute the modular inverse.

3 Task 2

First generate message/signature pairs by the `ecdsa_sign()` function you have implemented but with a fixed nonce k. The main task is to implement the following function:

```
def attack()  #  given two lists [m,r,s], return private key d
```

4 Requirements

To implement the elliptic curves, you can install an open-source software `Sage`[1], which is based on Python. Please refer to the doc[2] for more details on constructing elliptic curves with `Sage`. The FAQ is also available[3].

To ease the grading, please submit the source codes (.py, .sage, or .ipynb files) and write detailed comments of your codes! Please also submit a report including

- your choices of d and k and whether your attack recovers the key correctly
- how long does it take to do one signature and verification with your implementation

[1] https://www.sagemath.org/
[2] https://doc.sagemath.org/html/en/reference/arithmetic_curves/sage/schemes/elliptic_curves/ell_finite_field.html
[3] https://doc.sagemath.org/html/en/faq/faq-usage.html