ECDSA: Elliptic Curve Signatures

The **ECDSA** (Elliptic Curve Digital Signature Algorithm) is a cryptographically secure **digital signature scheme**, based on the elliptic-curve cryptography (**ECC**). **ECDSA** relies on the math of the **cyclic groups of elliptic curves over finite fields** and on the difficulty of the **ECDLP problem** (elliptic-curve discrete logarithm problem). The **ECDSA sign / verify** algorithm relies on EC point multiplication and works as described below. ECDSA keys and signatures are shorter than in RSA for the same security level. A 256-bit ECDSA signature has the same security strength like 3072-bit RSA signature.

ECDSA uses cryptographic **elliptic curves** (EC) over finite fields in the classical Weierstrass form. These curves are described by their **EC domain parameters**, specified by various cryptographic standards such as **SECG: SEC 2** and **Brainpool (RFC 5639)**. Elliptic curves, used in cryptography, define:

- **Generator point G**, used for scalar multiplication on the curve (multiply integer by EC point)
- **Order *n*** of the subgroup of EC points, generated by **G**, which defines the length of the private keys (e.g. 256 bits)

For example, the 256-bit elliptic curve `secp256k1` has:

- Order ***n*** = 115792089237316195423570985008687907852837564279074904382605163141518161494337 (prime number)
- Generator point **G** {***x*** = 55066263022277343669578718895168534326250603453777594175500187360389116729240, ***y*** = 32670510020758816978083085130507043184471273380659243275938904335757337482424}

Key Generation

The **ECDSA key-pair** consists of:

- **private key** (integer): ***privKey***
- **public key** (EC point): ***pubKey*** = ***privKey*** * **G**

The **private key** is generated as a **random integer** in the range [0...***n***-1]. The public key ***pubKey*** is a point on the elliptic curve, calculated by the EC point multiplication: ***pubKey*** = ***privKey*** * **G** (the private key, multiplied by the generator point **G**).

The public key EC point {***x***, ***y***} can be **compressed** to just one of the coordinates + 1 bit (parity). For the `secp256k1` curve, the private key is 256-bit integer (32 bytes) and the compressed public key is 257-bit integer (~ 33 bytes).

ECDSA Sign

The ECDSA signing algorithm (**RFC 6979**) takes as input a message ***msg*** + a private key ***privKey*** and produces as output a **signature**, which consists of pair of integers {***r***, ***s***}. The **ECDSA signing** algorithm is based on the **ElGamal signature scheme** and works as follows (with minor simplifications):

1. Calculate the message **hash**, using a cryptographic hash function like SHA-256: ***h*** = hash(***msg***)
2. Generate securely a **random** number ***k*** in the range [1..***n***-1]
   - In case of **deterministic-ECDSA**, the value ***k*** is HMAC-derived from ***h*** + ***privKey*** (see RFC 6979)
3. Calculate the random point ***R*** = ***k*** * **G** and take its x-coordinate: ***r*** = ***R***.**x**
4. Calculate the signature proof: ***s*** = $k^{-1} * (h + r * privKey) \pmod n$
   - The modular inverse $k^{-1} \pmod n$ is an integer, such that $k * k^{-1} \equiv 1 \pmod n$
5. Return the **signature** {***r***, ***s***}.

The calculated **signature** {***r***, ***s***} is a pair of integers, each in the range [1...***n***-1]. It encodes the random point ***R*** = ***k*** * **G**, along with a proof ***s***, confirming that the signer knows the message ***h*** and the private key ***privKey***. The proof ***s*** is by idea verifiable using the corresponding ***pubKey***.

ECDSA signatures are **2 times longer** than the signer's **private key** for the curve used during the signing process. For example, for 256-bit elliptic curves (like `secp256k1`) the ECDSA signature is

512 bits (64 bytes) and for 521-bit curves (like `secp521r1`) the signature is 1042 bits.

ECDSA Verify Signature

The algorithm to **verify a ECDSA signature** takes as input the signed message ***msg*** + the signature {***r***, ***s***} produced from the signing algorithm + the public key ***pubKey***, corresponding to the signer's private key. The output is boolean value: ***valid*** or ***invalid*** signature. The **ECDSA signature verify** algorithm works as follows (with minor simplifications):

1. Calculate the message **hash**, with the same cryptographic hash function used during the signing: ***h*** = hash(***msg***)
2. Calculate the modular inverse of the signature proof: ***s1*** = $s^{-1} \pmod n$
3. Recover the random point used during the signing: ***R'*** = (***h*** * **s1**) * **G** + (***r*** * ***s1***) * ***pubKey***
4. Take from ***R'*** its x-coordinate: ***r'*** = ***R'*.x**
5. Calculate the signature validation **result** by comparing whether ***r'*** == ***r***

The general idea of the signature verification is to **recover the point *R'*** using the public key and check whether it is same point ***R***, generated randomly during the signing process.

How Does it Work?

The **ECDSA signature** {***r***, ***s***} has the following simple explanation:

- The signing **signing** encodes a random point ***R*** (represented by its x-coordinate only) through elliptic-curve transformations using the private key ***privKey*** and the message hash ***h*** into a number ***s***, which is the **proof** that the message signer knows the private key ***privKey***. The signature {***r***, ***s***} cannot reveal the private key due to the difficulty of the **ECDLP problem**.
- The **signature verification** decodes the proof number ***s*** from the signature back to its original point ***R***, using the public key ***pubKey*** and the message hash ***h*** and compares the x-coordinate of the recovered ***R*** with the ***r*** value from the signature.

The Math behind the ECDSA Sign / Verify

Read this section **only if you like math**. Most developer may skip it.

How does the above sign / verify scheme work? It is not obvious, but let's play a bit with the equations.

The equation behind the recovering of the point ***R'***, calculated during the **signature verification**, can be transformed by replacing the ***pubKey*** with ***privKey*** * **G** as follows:

R' = (***h*** * ***s1***) * **G** + (***r*** * ***s1***) * ***pubKey*** =
= (***h*** * ***s1***) * **G** + (***r*** * ***s1***) * ***privKey*** * **G** =
= (***h*** + ***r*** * ***privKey***) * ***s1*** * **G**

If we take the number ***s*** = $k^{-1} * (h + r * privKey) \pmod n$, calculated during the signing process, we can calculate ***s1*** = $s^{-1} \pmod n$ like this:

s1 = $s^{-1} \pmod n$ =
= $(k^{-1} * (h + r * privKey))^{-1} \pmod n$ =
= $k * (h + r * privKey)^{-1} \pmod n$

Now, replace ***s1*** in the point ***R'***.

R' = (***h*** + ***r*** * ***privKey***) * ***s1*** * **G** =
= $(h + r * privKey) * k * (h + r * privKey)^{-1} \pmod n$ * **G** =
= **k** * **G**

The final step is to **compare** the **point** ***R'*** (decoded by the ***pubKey***) with the **point** ***R*** (encoded by the ***privKey***). The algorithm in fact compares only the x-coordinates of ***R'*** and ***R***: the integers ***r'*** and ***r***.

It is expected that ***r'*** == ***r*** if the signature is **valid** and ***r'*** ≠ ***r*** if the signature or the message or the public key is incorrect.

ECDSA: Public Key Recovery from Signature

It is important to know that the **ECDSA signature scheme** allows the **public key to be recovered** from the signed **message** together with the **signature**. The recovery process is based on some **mathematical computations** (described in the **SECG: SEC 1** standard) and returns 0, 1 or 2 possible EC points that are valid **public keys**, corresponding to the signature. To avoid this ambiguity, some ECDSA implementations add one additional bit ***v*** to the signature during the signing process and it takes the form {***r***, ***s***, ***v***}. From this extended ECDSA signature {***r***, ***s***, ***v***} + the signed **message**, the signer's public key can be restored with confidence.

The **public key recovery from the ECDSA signature** is very useful in bandwidth constrained or storage constrained environments (such as blockchain systems), when transmission or storage of the public keys cannot be afforded. For example, the Ethereum blockchain uses extended signatures {***r***, ***s***, ***v***} for the signed transactions on the chain to save storage and bandwidth.

Public key recovery is possible for signatures, based on the **ElGamal signature scheme** (such as DSA and ECDSA).