WIKIPEDIA

Elliptic Curve Digital Signature Algorithm

In cryptography, the **Elliptic Curve Digital Signature Algorithm** (**ECDSA**) offers a variant of the Digital Signature Algorithm (DSA) which uses elliptic curve cryptography.

Contents

Key and signature-size comparison to DSA

As with elliptic-curve cryptography in general, the bit size of the public key believed to be needed for ECDSA is about twice the size of the security level, in bits. For example, at a security level of 80 bits (meaning an attacker requires a maximum of about 2^{80} operations to find the private key) the size of an ECDSA public key would be 160 bits, whereas the size of a DSA public key is at least 1024 bits. On the other hand, the signature size is the same for both DSA and ECDSA: approximately $4t$ bits, where t is the security level measured in bits, that is, about 320 bits for a security level of 80 bits.

Signature generation algorithm

Suppose Alice wants to send a signed message to Bob. Initially, they must agree on the curve parameters $(\mathbf{CURVE}, G, n)$. In addition to the field and equation of the curve, we need G, a base point of prime order on the curve; n is the multiplicative order of the point G.

Parameter	
CURVE	the elliptic curve field and equation used
G	elliptic curve base point, such as a pt (x_0, y_0) on $y^2 = x^3 + 7$, a generator of the elliptic curve with large prime order *n*
n	integer order of *G*, means that $n \times G = O$, where O is the identity element.

The order n of the base point G **must be prime**. Indeed, we assume that every nonzero element of the ring $\mathbb{Z}/n\mathbb{Z}$ are invertible, so that $\mathbb{Z}/n\mathbb{Z}$ must be a field. It implies that n must be prime (cf. Bézout's identity).

Alice creates a key pair, consisting of a private key integer d_A, randomly selected in the interval $[1, n-1]$; and a public key curve point $Q_A = d_A \times G$. We use $\times$ to denote elliptic curve point multiplication by a scalar.

For Alice to sign a message m, she follows these steps:

1. Calculate $e = \mathbf{HASH}(m)$. (Here HASH is a cryptographic hash function, such as SHA-2, with the output converted to an integer.)
2. Let z be the L_n leftmost bits of e, where L_n is the bit length of the group order n. (Note that z can be *greater* than n but not *longer*.[1])
3. Select a **cryptographically secure random** integer k from $[1, n-1]$.
4. Calculate the curve point $(x_1, y_1) = k \times G$.
5. Calculate $r = x_1 \mod n$. If $r = 0$, go back to step 3.
6. Calculate $s = k^{-1}(z + rd_A) \mod n$. If $s = 0$, go back to step 3.
7. The signature is the pair (r, s). (And $(r, -s \mod n)$ is also a valid signature.)

As the standard notes, it is not only required for k to be secret, but it is also crucial to select different k for different signatures, otherwise the equation in step 6 can be solved for d_A, the private key: Given two signatures (r, s) and (r, s'), employing the same unknown k for different known messages m and m', an attacker can calculate z and z', and since $s - s' = k^{-1}(z - z')$ (all operations in this paragraph are done modulo n) the attacker can find $k = \frac{z - z'}{s - s'}$. Since $s = k^{-1}(z + rd_A)$, the attacker can now calculate the private key $d_A = \frac{sk - z}{r}$. This implementation failure was used, for example, to extract the signing key used for the PlayStation 3 gaming-console.[2] 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.[3] To ensure that k is unique for each message one may bypass random number generation completely and generate deterministic signatures by deriving k from both the message and the private key.[4]

Signature verification algorithm

For Bob to authenticate Alice's signature, he must have a copy of her public-key curve point Q_A. Bob can verify Q_A is a valid curve point as follows:

1. Check that Q_A is not equal to the identity element O, and its coordinates are otherwise valid
2. Check that Q_A lies on the curve
3. Check that $n \times Q_A = O$

After that, Bob follows these steps:

1. Verify that r and s are integers in $[1, n-1]$. If not, the signature is invalid.
2. Calculate $e = \mathrm{HASH}(m)$, where HASH is the same function used in the signature generation.
3. Let z be the L_n leftmost bits of e.
4. Calculate $u_1 = zs^{-1} \mod n$ and $u_2 = rs^{-1} \mod n$.
5. Calculate the curve point $(x_1, y_1) = u_1 \times G + u_2 \times Q_A$. If $(x_1, y_1) = O$ then the signature is invalid.
6. The signature is valid if $r \equiv x_1 \pmod n$, invalid otherwise.

Note that an efficient implementation would compute inverse $s^{-1} \mod n$ only once. Also, using Shamir's trick, a sum of two scalar multiplications $u_1 \times G + u_2 \times Q_A$ can be calculated faster than two scalar multiplications done independently.[5]

Correctness of the algorithm

It is not immediately obvious why verification even functions correctly. To see why, denote as C the curve point computed in step 6 of verification,

$$C = u_1 \times G + u_2 \times Q_A$$

From the definition of the public key as $Q_A = d_A \times G$,

$$C = u_1 \times G + u_2 d_A \times G$$

Because elliptic curve scalar multiplication distributes over addition,

$$C = (u_1 + u_2 d_A) \times G$$

Expanding the definition of u_1 and u_2 from verification step 5,

$$C = (zs^{-1} + rd_A s^{-1}) \times G$$

Collecting the common term s^{-1},

$$C = (z + rd_A)s^{-1} \times G$$

Expanding the definition of s from signature step 6,

$$C = (z + rd_A)(z + rd_A)^{-1}(k^{-1})^{-1} \times G$$

Since the inverse of an inverse is the original element, and the product of an element's inverse and the element is the identity, we are left with

$$C = k \times G$$

From the definition of r, this is verification step 7.

This shows only that a correctly signed message will verify correctly; many other properties are required for a secure signature algorithm.

Security

In December 2010, a group calling itself *failoverflow* announced recovery of the ECDSA private key used by Sony to sign software for the PlayStation 3 game console. However, this attack only worked because Sony did not properly implement the algorithm, because k was static instead of random. As pointed out in the Signature generation algorithm section above, this makes d_A solvable and the entire algorithm useless.[6]

On March 29, 2011, two researchers published an IACR paper[7] demonstrating that it is possible to retrieve a TLS private key of a server using OpenSSL that authenticates with Elliptic Curves DSA over a binary field via a timing attack.[8] The vulnerability was fixed in OpenSSL 1.0.0e.[9]

In August 2013, it was revealed that bugs in some implementations of the Java class SecureRandom (https://docs.oracle.com/javase/10/docs/api/java/security/SecureRandom.html) sometimes generated collisions in the k value. This allowed hackers to recover private keys giving them the same control over bitcoin transactions as legitimate keys' owners had, using the same exploit that was used to reveal the PS3 signing key on some Android app implementations, which use Java and rely on ECDSA to authenticate transactions.[10]

This issue can be prevented by deterministic generation of k, as described by RFC 6979.

Concerns

There exist two sorts of concerns with ECDSA:

1. *Political concerns*: the trustworthiness of NIST-produced curves being questioned after revelations that the NSA willingly inserts backdoors into software, hardware components and published standards were made; well-known cryptographers[11] have expressed[12][13] doubts about how the NIST curves were designed, and voluntary tainting has already been proved in the past.[14][15]
2. *Technical concerns*: the difficulty of properly implementing the standard,[16] its slowness, and design flaws which reduce security in insufficiently defensive implementations of the Dual EC DRBG random number generator.[17]

Both of those concerns are summarized in *libssh curve25519 introduction*.[18]