Asymmetric Passwords

www.kryptologik.com

Passwords are the most common form of authentication used to control access to information and services. Passwords are widely used because they are simple, inexpensive, convenient and easy to implement. On the other hand, most of the reported security incidents are password related.

Asymmetric Passwords addresses some of the password issues by utilizing asymmetric cryptography. A dedicated library derives a private key from user id/password/domain and shares the associated public key with the verifying agent during registration phase. Client library generates a signed and timestamped token as the evidence of its authenticity where this token is a URL-safe ANSI string. On the server side, passwords are replaced with associated public keys.

Generated tokens look like: *tag.timestamp.payload.signature*

Where:

Tag defines what operation this token is issued for.

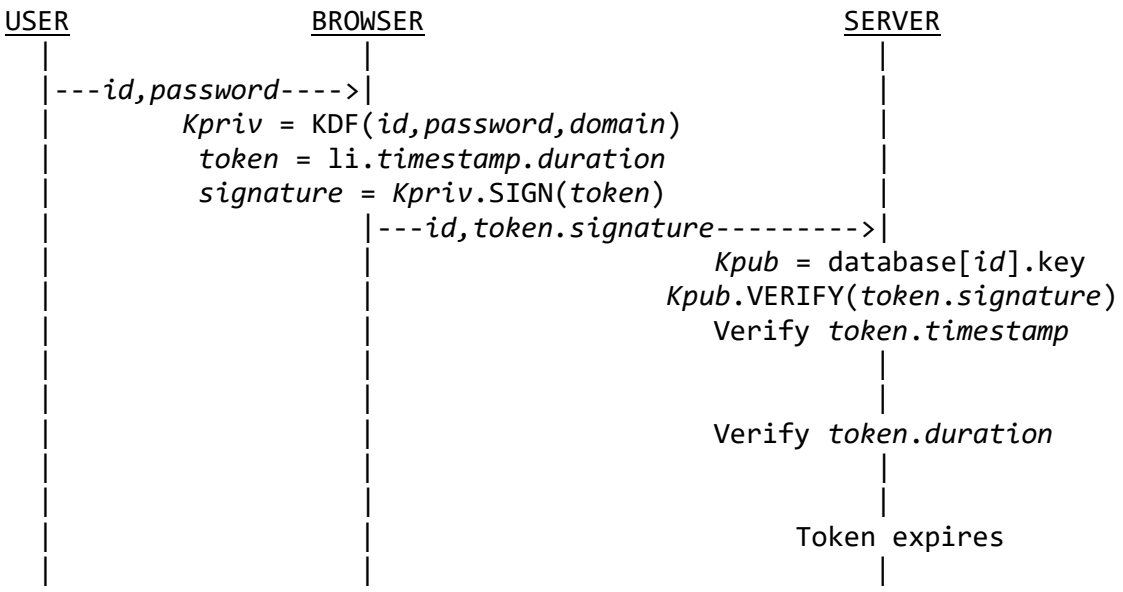
Timestamp is the token creation time (in seconds since Jan 1, 1970 UTC).

Payload contains the parameters for the operation. This part of the token may include a *nonce* as well as other custom information.

Signature field is the last item of a token and is generated by the derived private key. Signature field is presented as a URL-safe base-64 string.

User login looks like:

User login:



Where:

KDF = Key Derivation Function (such as PBKDF2)
ECC = Elliptic Curve Cryptosystem (ECP-256)
timestamp = seconds since 1/1/1970 UTC
duration = how long token is valid (in seconds)

With this approach:

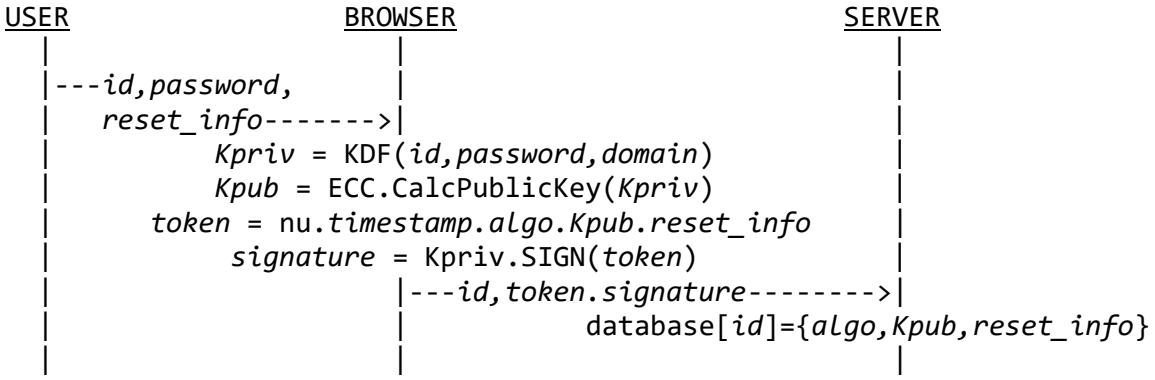
1. Passwords stay private.
    a. Users do not need to share passwords with anyone. Public keys will be given instead.
    b. Passwords do not cross the line.
    c. Protection against eavesdropping on network connections.
2. Promote enforcement of time-based access control.
    a. Protection against replay attacks
    b. Protects against token-reuse (One-time tokens).
3. Server side security breaches do not compromise user accounts.
    a. Leaked user public keys do not compromise account access information.
    b. Protection against insider threats.
    c. Public keys can be stored in clear.
    d. No need for secure connection during login process.
4. Passwords can be shared for multiple accounts.
    a. KDF is client-only operation and may use domain-specific info for differentiation.
    b. Reducing number of passwords to memorize.
5. Support for random challenge/response authentication.

6. Password splitting option:
    a. Part 1: Memorize this part of password.
    b. Part 2: Long and complex password part. This part is going to be stored on specific machines individually.
    c. Part 2 can be encrypted using part 1 as the key.
    d. Ability to enforce logins from pre-determined computers.
    e. A form of multi-factor authentication.
7. Ease of deployment.
    a. Passwords (and password hashes) can easily be converted to public keys.
    b. No need to change user interface.
    c. Push via login portals.
8. Extendable and flexible URL-safe token design.

Developers are free to pick their own KDF and ECC algorithm. PBKDF2 and HMAC are good candidates as KDF and they may include some custom secret keys as extended domain parameter.

To facilitate automatic conversion of 'password hash' databases to public key, use the password hash instead of password as the KDF input parameter.

Register a new user:

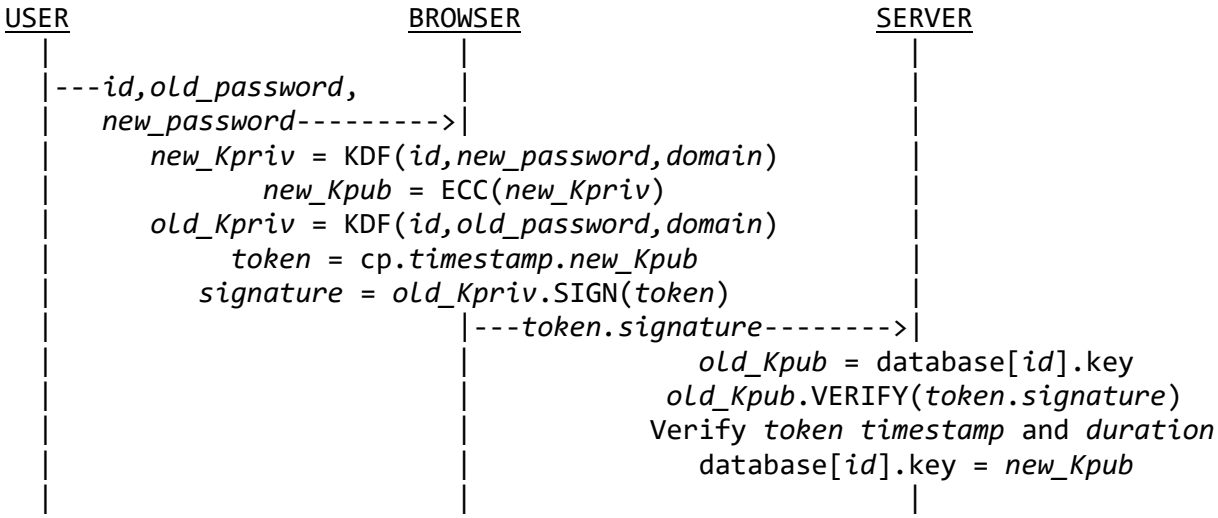


Where:

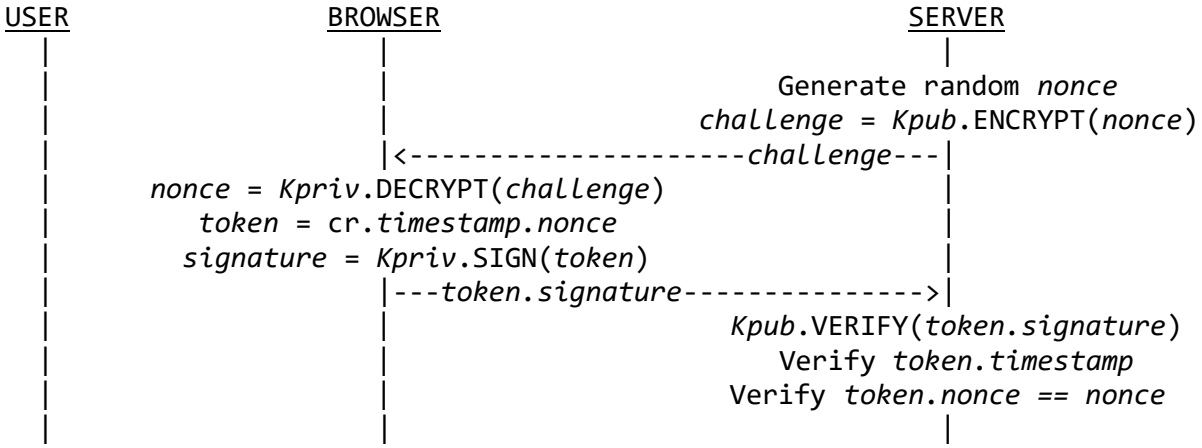
algo = Identifier for the used algorithm (ECC & curve)

reset_info = information used for password reset

Change password:



Challenged authentication:



Challenge/response authentication may be combined with login process. Alternatively, it can be used only in cases that there is too much skew between client and server times.

The challenge exchange part can be simplified by defining the actual nonce as the challenge (with no encryption/decryption).

A C++ project currently available based on using ED25519 as the ECC algorithm and SHA512 as KDF. Javascript version of this project is in development currently.