subnoto.

White Paper

September 2025

General architecture **3**
Principles 3
Signature's platform 3
Hardware security for isolation and in use memory encryption 4
Attestations and binaries reproducibility 6
Application's session security 7
Encryption keys management 7
Authentification et Intégrité – HTTP Signature RFC 9421 ..... **9**
General 9
Signature process 9
Mandatory signed elements 10
Implementation of the verification 10
Timing attacks security 10
Passwordless authentication **11**
Supported methods 11
Email Authentication 11
Workflow of email authentication 11
Generation and storage of OTP 12
OTP validation 12
Passkey authentication (WebAuthn) **13**
Implementation 13
Workflow of passkey registration 13
Workflow of passkey authentication 14
Management of email tokens **15**
Usage 15
Secure links generation 15
Tokens link validation 15
Sessions security **17**
Architecture 17
Sessions management 17
Encryption and protection of data **18**
AES-GCM encryption with context 18
Contextual encryption: details 18
Architecture of encryption 18
Compliance and standards **19**
Standards used 19
Security good practices 19

General architecture

Principles

Subnoto's platform is built around the following key assumptions:

- **Security must be transparent and provable**—pinky promises and security through obscurity aren't enough. That's why we use remotely attestable components.
- Encryption at rest and in transit—the current industry standards—aren't sufficient for today's modern threats. **All sensitive data is encrypted at rest, in transit, and even while in use.**
- **Customers own their data** (meaning only they can decide who can access it), and we design everything to ensure nobody—not even Subnoto administrators or cloud providers—can access it.
- Classic passwords or any stealable tokens (like cookies or standard API keys) are unreliable and should be avoided. Instead, we use **Passkeys**, signed API requests, and NOISE tunneling over HTTPS.

Signature's platform

The signature's platform consists of four main components:

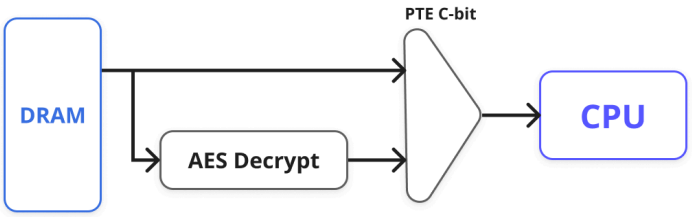
1. **Web Application**:
    - A single-page application (SPA) served from a trusted, signed container.
    - Integrity is verifiable by rebuilding from public source code.
2. **Main Enclave**:
    - Runs in isolation with memory encryption.
    - Handles critical operations like encryption/decryption, session management, and signatures.
3. **Key Provisioners**:
    - Generate and protect encryption keys, ensuring only authorized enclaves can access them.
4. **Utility Main API**:
    - Manages non-critical operations such as external payments and invoicing.

Hardware security for isolation and in use memory encryption

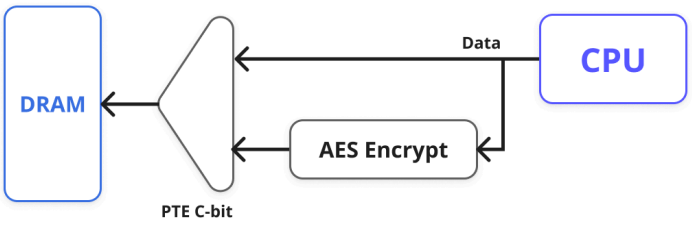
Subnoto leverages state-of-the-art hardware isolation and memory encryption using AMD SEV/SNP, the Linux Kernel, QEMU, and Google's open-source Oak middleware[1]. Our enclaves are "Confidential VMs" (CVMs) built on these key technologies.

The AMD SEV/SNP[2] secure microcontroller is responsible for:

- **Memory encryption**: This is hardware-based, and no software can access the encryption keys. The keys are managed entirely by the Secure Processor, a 32-bit microcontroller (ARM) that functions as a dedicated security subsystem integrated within the SoC (system on chip). Each key is generated using the onboard NIST SP 800-90-compliant hardware random number generator and stored in dedicated hardware registers, where it is never exposed outside the SoC in clear text.
    - Memory Read

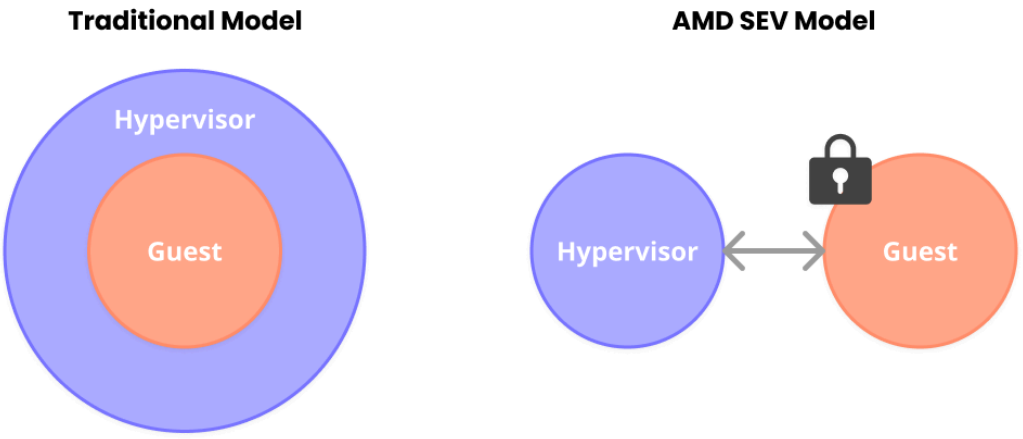


    - Memory Write



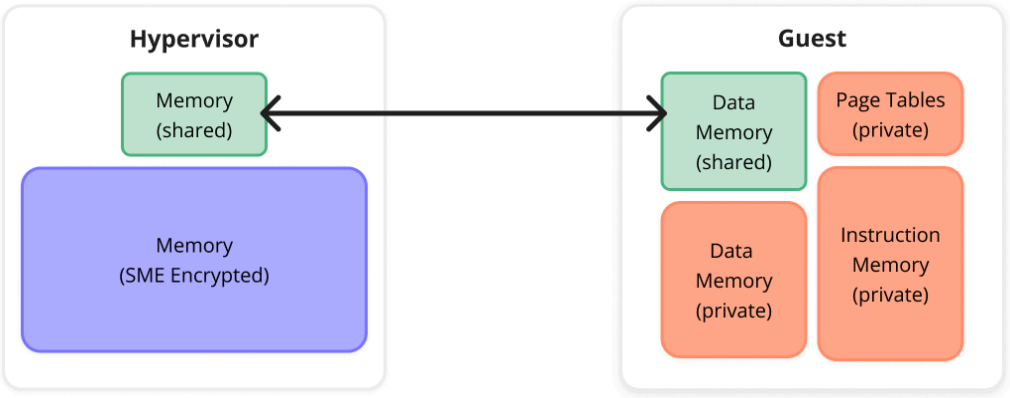
[1] Project oak: https://github.com/project-oak/oak
[2] AMD SEV/SNP https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf

- and **Software isolation:** Unlike the traditional model where the hypervisor must be trusted, CVMs eliminate this requirement.



The Linux Kernel and QEMU are untrusted components that call various CPU APIs to:

- Launch and trigger the hardware encryption to encrypt the Confidential VM.
- Trigger the signed measurement of the initial state from the CPU
- Allow selected data to flow between the insecure environment and the secure one. Most communication occurs through a memory space allocated and shared between the Guest and the Hypervisor.



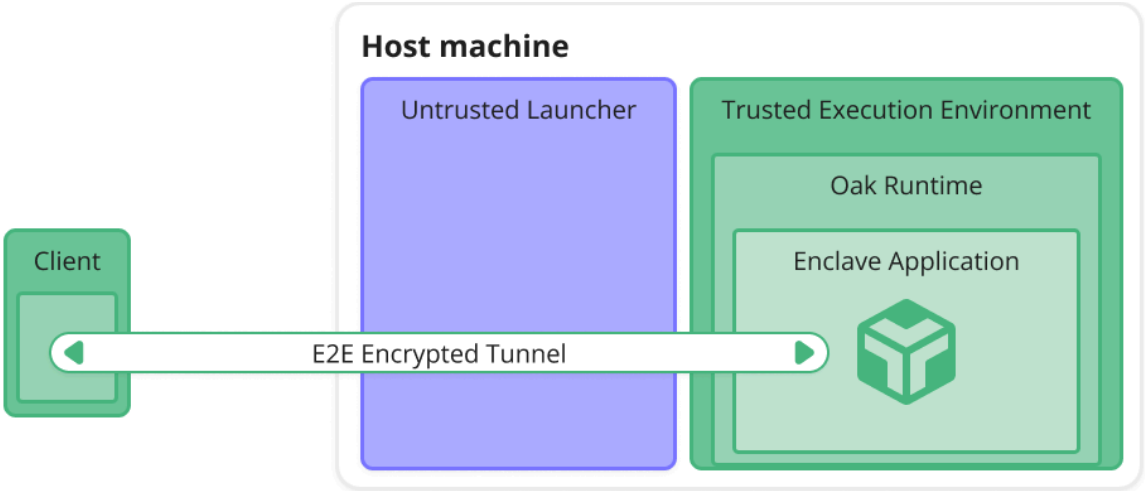
Attestations and binaries reproducibility

To maximize the benefits of this architecture, we need trusted and verifiable code running inside the enclave. We draw heavy inspiration from the Oak project and use hermetic Bazel build[3] to achieve reproducibility.

- The supply chain is minimized to the essentials and pinned to specific versions.
- A key property of Bazel builds is determinism: each set of rules ensures the same output every time the build runs. That's what allows the customers or a third party to verify that the published code is indeed generating the attested components used in production.

This process produces a set of components (from the VM firmware and BIOS to the container running the application code, as well as the system image and kernel) that are hashed and signed independently using both a private key and the Cosign keyless system.

These steps generate a set of attestations that are presented to our client apps, which can then decide whether to trust them.



The Cosign keyless system allows us to link each attestation to a commit of our git repository, it also prevents the usage of a long lived signature key (that could stolen or misused) by leveraging the Sigstore's Certificate Authority and an OIDC authentication from the CI job itself.[4]

[3] Bazel hermetic builds: https://bazel.build/basics/hermeticity
[4] Sigstore's Certificate Authority: https://docs.sigstore.dev/certificate_authority/certificate-issuing-overview/

Application's session security

When a client application connects to an Enclave Application, it first requests the attestation evidence and associated endorsements. The endorsements include a certificate chain from AMD to prove that the attestation report was signed by a legitimate AMD CPU, as well as signed statements from the developers of the various components running inside the enclave.

It then verifies the evidence using these endorsements and the expected reference to ensure it matches expectations. The client application should also confirm that the enclave application is running in an up-to-date and correctly configured Trusted Execution Environment (TEE)—for example, that the CPU is using the latest versions of microcode and SNP firmware, and that debugging is disabled—and that the identities of the various components inside the enclave match expectations.

Once the client app is satisfied that the enclave application meets all its requirements, it uses a public key bound to the evidence to establish an encrypted channel with the enclave application.

This session occurs on top of the standard HTTPS session and is based on the NOISE[5] protocol.

All the sensitive API methods of our signature app are using this type of session.

Encryption keys management

A challenging aspect of the architecture is providing persistent encryption keys for persistent data that we are unable to access. AMD SEV/SNP can generate such keys for each hardware device, but the usable key is derived from the initial measurement of the enclave extracting it. This means the keys differ for each different CPU and when the initial measurement changes (e.g., during code updates).

To address this, we've built a dedicated service that encrypts a persistent key to external storage using the AMD SEV-derived key. Enclave apps can query this key provider, which verifies their attestations through a bidirectional Oak session[6] to confirm authorization. New versions of the key provider can use the same mechanism to retrieve the encryption key and persist it using their own TEE-derived key.

[5] The Noise protocol framework : https://noiseprotocol.org/
[6] Oak Session : https://github.com/project-oak/oak/tree/main/oak_session

This key provider is using only two small components from Oak on top of the application itself (no system, no linux kernel or anything else) :

- **stage0**[7], that is the firmware used to boot the CVM, attest the other components and launch the kernel.
- the **oak_restricted_kernel**[8] (60k lines of rust) that is doing the minimum required to launch our application.
- The application itself (a few lines of rust).

It allows us to have a very small trusted computing base for this critical component and to be very simple to audit and verify that no one can access the encryption keys.

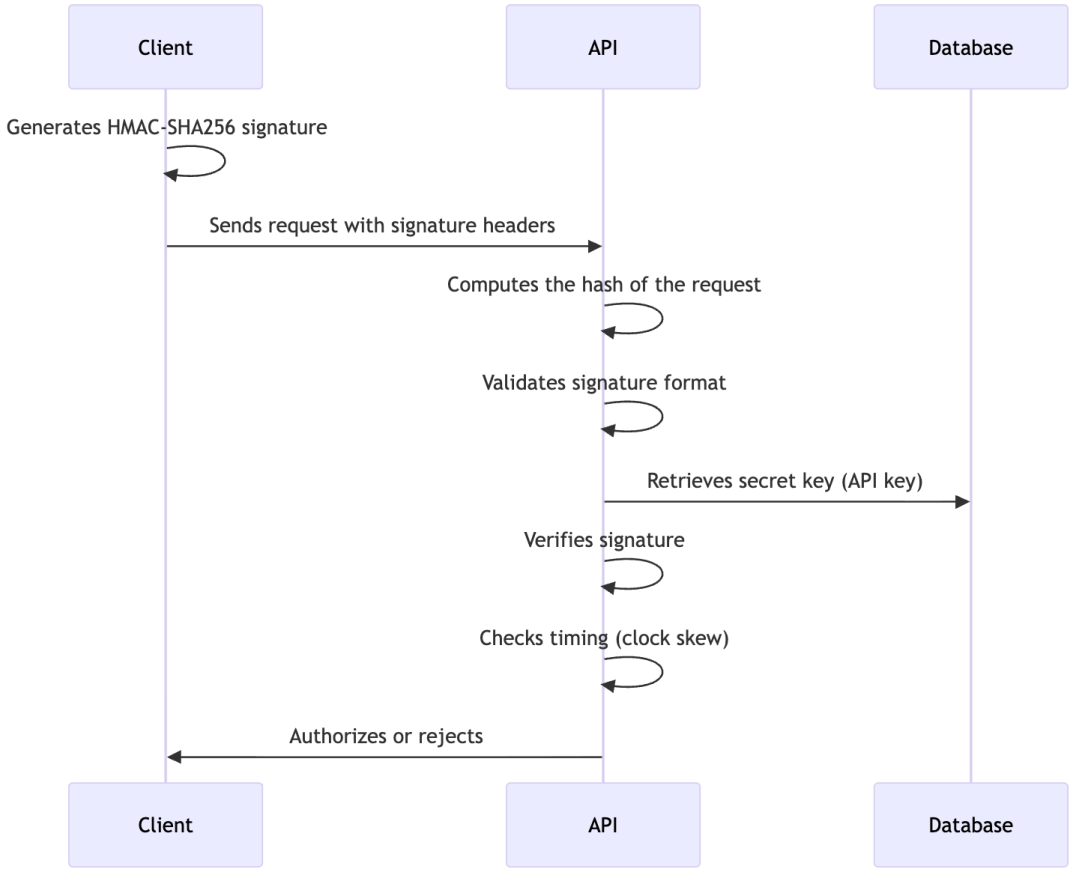
[7] Oak stage0 : https://github.com/project-oak/oak/tree/main/stage0
[8] Oak restricted kernel : https://github.com/project-oak/oak/tree/main/oak_restricted_kernel

Integrity and authentication

General

All requests to the enclaves are encapsulated within a NOISE tunnel (ensuring integrity and confidentiality).
Furthermore, each API request uses the RFC 9421 standard to guarantee authenticity and integrity.
The HTTP signature protects both critical headers and the request body against any alteration or interception.

Signature process



Mandatory signed elements

Each request must sign four critical components to ensure the complete integrity of the communication. The system is required to verify the request's timestamp to prevent replay attacks, the HTTP authority to ensure the request is intended for the correct server, the content type to validate the format of the data being sent, and finally the hash of the request body to guarantee that the content has not been tampered with during transit.

Implementation of the verification

The HTTP signature verification process follows a rigorous sequence of five main steps. First, the system extracts and validates all the required signature headers from the incoming request. Next, it parses the signature according to the strict specifications of RFC 9421, breaking down the structural elements of the signature. The third step consists of identifying the type of authentication used, whether it is an application key or a session key. The system then retrieves the corresponding secret key from the database. Finally, it performs the actual cryptographic verification using the HMAC-SHA256 algorithm, comparing the provided signature with the one calculated locally. If this verification fails, an authentication error is immediately raised to reject the request.

Timing attacks security

The API implements a sophisticated protection against replay attacks by rigorously validating the time offset of requests. The system checks that the signature creation timestamp falls within an acceptable time window. This approach protects against attempts to maliciously reuse previously captured signatures, while taking into account normal clock variations between different systems.

Passwordless authentication

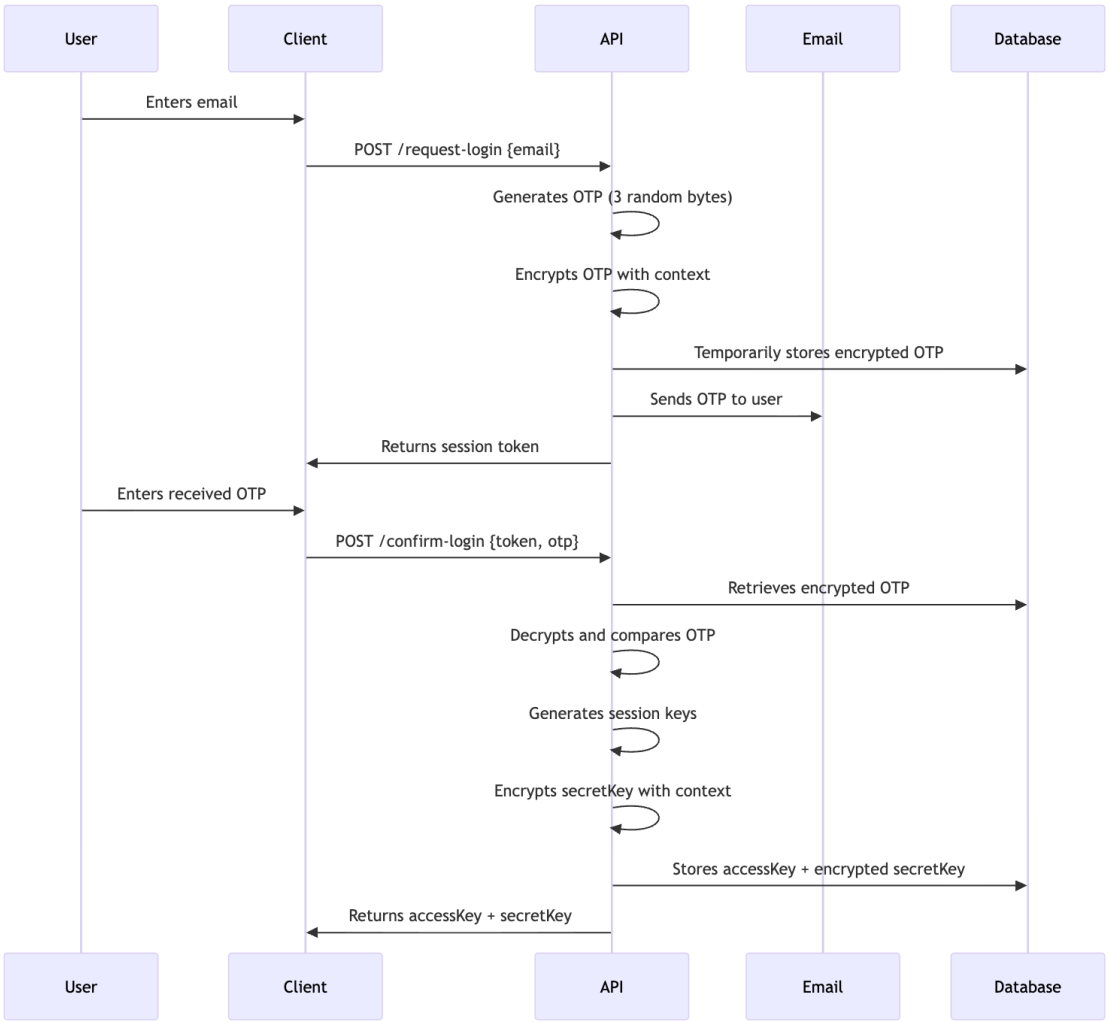
The Subnoto system completely eliminates the use of traditional passwords, opting for more secure authentication methods.

Supported methods

- **OTP by email** : one time code, secure transaction
- **WebAuthn/Passkeys** : biometric authentication or security hardware key
- **SSO/OIDC** : delegate to third party authentication service

Email Authentication

Workflow of email authentication



Generation and storage of OTP

The OTP (One-Time Password) generation process follows strict cryptographic standards to ensure their unpredictability. The system first generates a random three-byte code using the system's secure cryptographic generator, thus producing a six-character hexadecimal code. Simultaneously, a unique UUID token is created to identify the login session.

The OTP is then encrypted using the AES-GCM algorithm with a specific context that includes the token and the user's email address. This contextual encryption approach prevents malicious alterations and exchanges in the database. Therefore, if during decryption the context is not exactly the same, the decrypted value is never obtained and the authentication process fails.

The encrypted code is temporarily stored in memory with a lifetime limited to five minutes, along with the encryption key version and the expiration date. The OTP is sent via the Brevo service (a French transactional email provider), where the code is displayed in uppercase to make it easier for the user to enter.

OTP validation

The OTP validation process follows a secure protocol that protects against several types of attacks. When a user submits their code, the system first retrieves the temporarily stored encrypted OTP using the session token.

Decryption is performed using the same key and context as during the initial encryption, ensuring the integrity of the process. The comparison between the submitted OTP and the decrypted OTP uses a time-constant comparison function, which takes the same amount of time regardless of the similarity of the values, thus protecting against timing attacks.

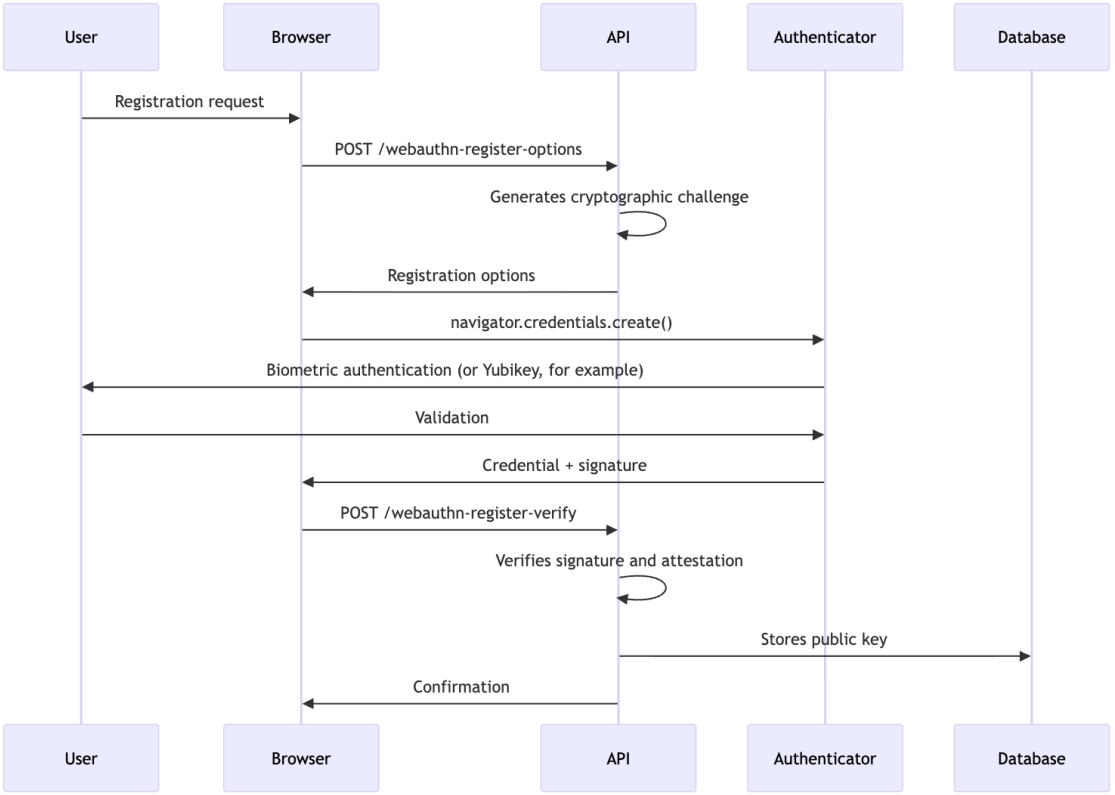
The system also checks that the OTP has not expired by comparing the current timestamp with the stored expiration date. If validation fails for any reason—incorrect or expired code—a specific exception is raised. The OTP is immediately deleted from temporary storage after use, whether successful or not, preventing any reuse and limiting the exposure window.

Passkey authentication (WebAuthn)

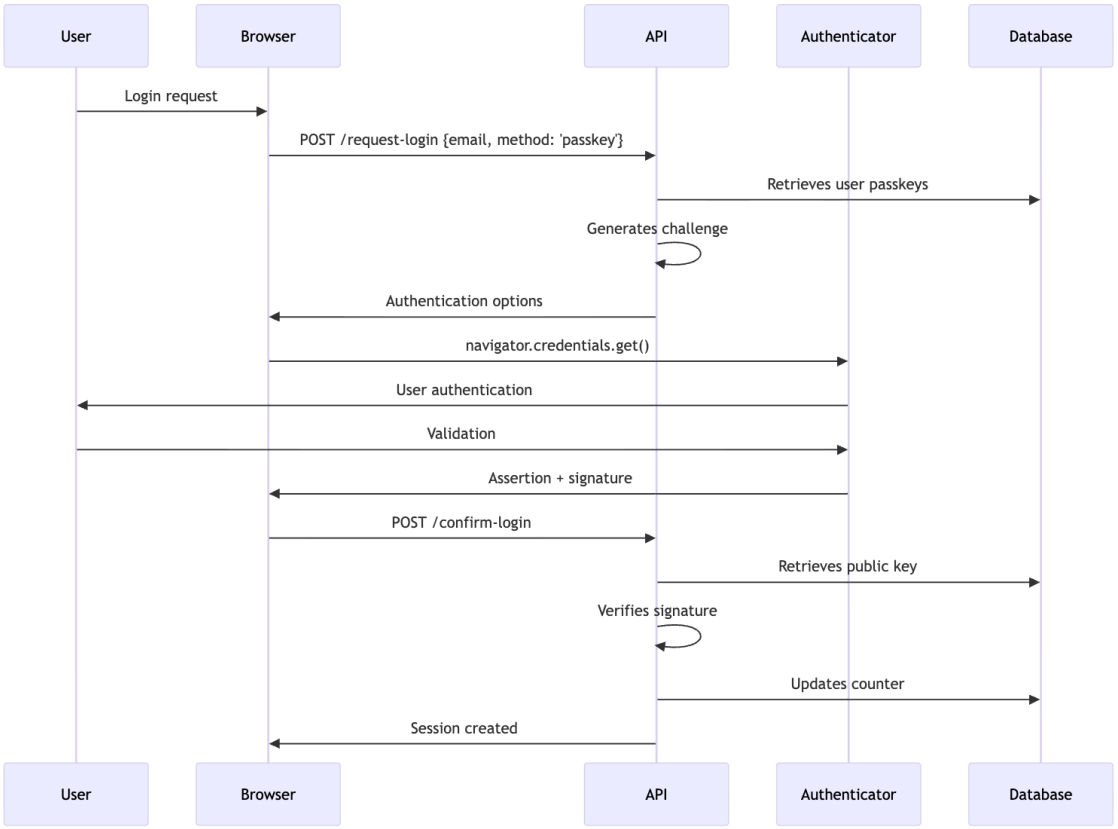
Implementation

- WebAuthn level 2 : Full support for passkey authentication (biometric key, hardware key, etc.).
- Registration : Creation of a public/private key pair linked to the device, with storage on the client side.
- Authentication : Cryptographic challenge, the private key never leaves the device.

Workflow of passkey registration



Workflow of passkey authentication



Management of email tokens

Secret tokens are a mechanism used whenever an invitation is sent via an external communication channel such as email. To ensure that even a Subnoto employee cannot alter or access the content of these invitations, secret tokens are always encrypted—in transit, during execution, and when stored.

Usage

Subnoto uses unique secret tokens to secure the following links:

- envelope signature links
- invitations to join a team

Secure links generation

The process of creating secure signature links combines several cryptographic elements to ensure maximum security. When sending an envelope, the system first generates a unique UUID token that will serve as the public identifier for the link. At the same time, a 32-byte secret key is generated using a cryptographically secure random number generator (TRNG).

This secret key is then encrypted using the AES-GCM algorithm with a specific context that includes the recipient's email, the UUID of the resource to be accessed (such as an envelope or team), and the generated token. This contextual encryption approach ensures that the key can only be decrypted in the exact context for which it was created.

The encrypted data is stored in the database, along with the version of the encryption key used and the recipient's metadata. The final link is constructed by combining the frontend URL with the UUID of the resource to be accessed, and by including the token and the secret key in the URL fragment (after the # symbol), which prevents their transmission to the server in standard HTTP logs.

Tokens link validation

The validation of link tokens follows a rigorous cryptographic verification process to ensure the authenticity and integrity of each access. When a user accesses a secure link, the system first retrieves the recipient's information associated with the provided token from the database.

The decryption process reconstructs the original secret key using the same encryption key and context as during link creation. This context includes the recipient's email, the resource UUID, and the token, ensuring that decryption can only succeed if all these elements match exactly.

Final validation uses a cryptographically time-constant comparison between the decrypted secret key and the one provided in the URL. This comparison method takes the same amount of time regardless of the similarity of the values, protecting against timing attacks that could gradually reveal the correct key.

Sessions security

Architecture

Subnoto sessions use a cryptographic key system rather than traditional cookies.

Sessions management

The session creation process completely abandons the traditional cookie-based approach in favor of a robust cryptographic key system. Generation begins with the creation of two distinct keys: a public access key and a secret key, both generated using cryptographically secure algorithms.

The secret key is immediately encrypted using the AES-GCM algorithm with a specific context including the access key (converted to hexadecimal), the user's email address, and their numeric identifier. This contextual encryption approach ensures that the key can only be decrypted in the exact context of its intended use.

Database storage includes not only the keys but also complete security metadata: the access key in hexadecimal format, the encrypted secret key encoded in base64, the version of the encryption key used, the user identifier, a description of the session, the originating IP address, and the client's user agent. This information allows for complete auditing and access traceability.

The system finally returns the access key in hexadecimal format and the secret key in base64url format, optimized respectively for storage and secure transmission via URLs.

Encryption and protection of data

AES-GCM encryption with context

- **Contextual AES-GCM** is used for all sensitive data
- **Context:** metadata related to the data, strict structure
- **Decryption impossible** outside the exact context (protection against reuse, key theft, etc.)
- **Base64 encoding** for portability

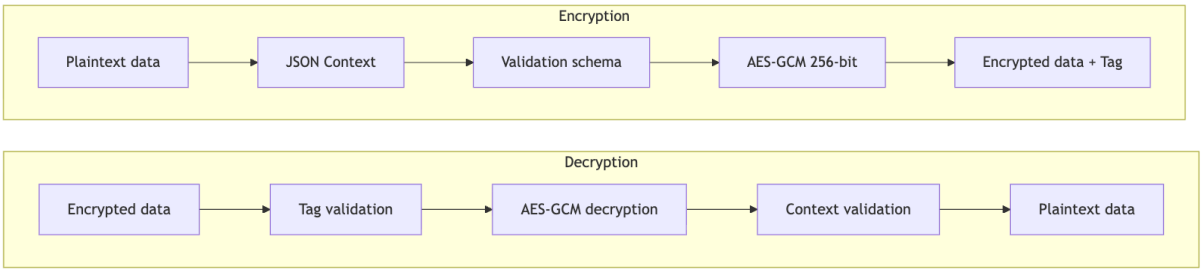
Contextual encryption: details

The contextual encryption system for secrets implements an advanced security approach that goes beyond simple symmetric encryption. This method integrates specific metadata directly into the cryptographic process to create an inseparable link between the encrypted data and its usage context.

For example, in the context of invitation links to sign a document, the encryption context is defined by a strict structure that includes the envelope UUID, the recipient's email address, and the invitation's primary key in base format. This structure is validated against a rigid JSON schema that specifies the expected data types and marks all fields as required, with an explicit prohibition of additional properties.

The AES-GCM algorithm with authenticated context ensures that data can only be decrypted if the exact context is provided. This approach protects against attacks where an attacker might try to reuse encrypted keys in a different context, even if they had access to the main encryption key. The encryption result is encoded in base64 to facilitate storage and transmission.

Architecture of encryption



Compliance and standards

Standards used

1. **RFC 9421:** HTTP Message Signatures
2. **WebAuthn Level 2:** Secure web authentication
3. **FIDO2/CTAP2:** Strong authentication protocols
4. **OpenID Connect:** Identity federation
5. **AES-GCM:** Authenticated encryption

Security good practices

1. **Modern cryptography:** Exclusive use of proven algorithms
2. **Key management:** Automatic rotation and versioning
3. **Zero Trust principle:** Systematic verification
4. **Security logging:** Complete access traceability
5. **Automatic expiration:** Tokens and sessions have limited duration