Security Whitepaper

Secret formats, encrypted delivery and cryptographic technologies

Version 1.5 - 6 September 2026 - secretus.app

1. What Secretus protects

Secretus protects sensitive content by encrypting it on the sending device and decrypting it on the receiving device. In the web application and browser extension, this happens locally in the browser. API integrations perform encryption in the calling application before uploading. In normal operation, the Secretus backend receives neither the secret's plaintext nor its decryption key.

Secret format	How it is handled
Text	Passwords, credentials, recovery codes, tokens and other private text.
Structured templates	SSH keys, database credentials, API keys and Wi-Fi credentials. Template fields are included inside the encrypted payload.
Files	File bytes, filename and MIME type are encrypted together in Standard mode. Maximum Security encrypts file bytes before transfer; file metadata travels inside the DTLS-protected WebRTC channel.
Voice messages	Audio is recorded locally and protected as encrypted content. Files and voice messages are supported in Standard and Maximum Security; Team Split is text-only.

Delivery choices

Mode	Purpose	Protection
Standard	Send a link that can be opened later.	AES-256-GCM; server holds ciphertext.
Maximum Security	Transfer while both browsers are connected.	Hybrid P-256 + ML-KEM-768, ratcheting keys and WebRTC.
Team Split	Require several share holders to reconstruct text.	Shamir k-of-n threshold sharing.
Request	Ask someone to return an encrypted secret.	Requester-generated AES-256-GCM key.

2. Standard: encrypted one-time links

The sender's browser creates a random 256-bit AES key using the Web Crypto API. It encrypts the content with AES-256-GCM, a fresh 96-bit initialization vector (IV) and a 128-bit authentication tag. The tag lets the receiving browser detect changes to the encrypted payload.

Step	Secret and key handling
1. Encrypt locally	Text, template fields and any attachment are encoded into one payload and encrypted before upload.
2. Store ciphertext	The service stores the encrypted payload, an identifier and delivery metadata. A domain-separated SHA-256 key commitment is sent instead of the AES key.
3. Build the link	The client combines the identifier with the key in the URL fragment: /s/<secretId>#async_<key>.
4. Reveal once	The recipient explicitly chooses Reveal. The browser proves the fragment capability using the commitment, retrieves ciphertext and decrypts locally.

What one-time access means

The service coordinates concurrent retrievals and invalidates the link before returning a successful payload response. Expiry and revocation also prevent further access through the service. Ciphertext cleanup is separate and asynchronous; link invalidation does not mean that every stored byte is physically erased at that instant. A recipient can still copy or save content after decryption.

3. Request: receiving an encrypted secret

Request reverses the direction of the workflow. The requester generates an AES-256-GCM key locally and shares a request URL whose `#req_...` fragment carries that key. The submitter's browser uses it to encrypt text or structured template fields before sending the response. This workflow uses the same AES payload format as Standard.

The service checks a key commitment for submission and restricts retrieval to the request owner's authenticated account. The requester then decrypts locally using the retained key. The backend does not obtain that key. Because the submitter also receives the symmetric key, the complete request link must be shared only with the intended submitter.

Payload versus delivery metadata

Encryption protects the content inside the payload. Optional labels, identifiers, expiry and delivery status are separate metadata and can be visible to the service. Put passwords, tokens and other confidential values inside the secret, not in a label.

4. Maximum Security: live encrypted transfer

Maximum Security establishes a live browser-to-browser session. The payload travels through a WebRTC data channel and is encrypted at the application layer before entering that channel. Secretus does not store the P2P secret payload on its servers. Both participants must be connected for the transfer.

Hybrid session establishment

The custom authenticated X3DH-style setup uses P-256 ECDH identity and prekeys, ECDSA signatures and ML-KEM-768 key encapsulation. Classical Diffie-Hellman outputs and the ML-KEM shared secret feed the key derivation process. HKDF-SHA-256 derives session key material. ML-KEM-768 is provided by `@noble/post-quantum`; the classical primitives use Web Crypto.

ML-KEM-768 adds a post-quantum component intended to reduce the risk of recorded traffic being decrypted by a future cryptanalytic attacker. This hybrid setup applies to Maximum Security; Standard, Request and the extension use the AES link workflow. Failure of the required ML-KEM exchange aborts the P2P session rather than silently removing that layer.

Per-message protection

Mechanism	Role in the secret transfer
AES-256-GCM	Encrypts message content and authenticates the protected data.
Directional key chains	Separate sending and receiving chains derive per-message keys through HMAC-SHA-256. Chain state advances after use.
DH ratchet	Fresh P-256 key agreement updates root and chain key material during the session.
Authenticated headers	Message counters and ratchet headers are bound to the ciphertext; replayed messages and excessive gaps are rejected.
WebRTC / DTLS / TURN	WebRTC adds DTLS transport protection. TURN relays encrypted packets when a direct route is unavailable; it does not decrypt the secret content.

Verify the person at the other end

Both browsers display a safety number derived from their identity keys and the ML-KEM public key. The sender must confirm a comparison over an independent, trusted channel before sending. Actually comparing this number is necessary to detect substitution of key bundles by an active signaling attacker.

This is a custom X3DH-style and ratcheting implementation, not a libsignal-compatible protocol. It has not received an independent cryptographic audit and makes no guarantee of recovery after session-state compromise.

5. Team Split: threshold-protected text

Team Split divides a text secret into **n shares** and requires at least **k valid shares** to reconstruct it. For example, a 3-of-5 split can be reconstructed by any three of the five holders. The application uses Shamir's Secret Sharing over GF(256), through the `shamir-secret-sharing` library, in the browser.

Under the Shamir scheme, fewer than k shares do not reveal the secret. Threshold sharing is a different mechanism from encrypting a payload with a single AES key. It is used here for text only, including credentials or keys represented as text.

Share integrity and reconstruction

Current share links include a split identifier, index, threshold and an HMAC-SHA-256 authentication tag. The separate reconstruction link carries the MAC verification key. The browser checks share integrity and consistency before combining them. The reconstruction link alone does not contain enough shares to recover the secret.

Creation and reconstruction happen locally. Share material is carried in URL fragments and is not uploaded by the Team Split workflow. Distribute shares to distinct holders through appropriate channels: anyone who collects enough valid shares can reconstruct. Expiry is a client-side hint, not server-enforced revocation of copied shares.

6. Technologies used to protect secrets

Technology	Used for
Web Crypto API	Browser-native AES-GCM, ECDH P-256, ECDSA, HKDF, HMAC and SHA-256 operations.
Cryptographic randomness	Fresh AES keys, IVs and other random cryptographic values generated using browser cryptographic APIs.
@noble/post-quantum	ML-KEM-768 encapsulation and decapsulation for Maximum Security.
shamir-secret-sharing	Client-side Shamir splitting and reconstruction for Team Split.
WebRTC data channels	Carries application-encrypted live messages, file data and voice content between browsers.
HTTPS / WSS	Protects ciphertext API transfers and session signaling in transit; it complements local payload encryption.
URL fragments + Base64url	Carries keys or shares after # in share links. Base64url is an encoding, not encryption.

7. Browser extension: the same encrypted links

The Secretus Chrome and Firefox extension encrypts text and file attachments locally using Web Crypto AES-256-GCM, with a random 256-bit key, a fresh 96-bit IV and a 128-bit tag. It uses the same encrypted payload format as the web application's Standard mode, so an extension-created link is decrypted by the same web receiver.

The extension uploads ciphertext and a key commitment. It constructs the complete share URL locally, placing the decryption key in the `#async_...` fragment. It does not ask the API to encrypt plaintext. Page text is selected for sharing when the user explicitly invokes the sharing action.

8. API: encrypt first, upload second

The secret-creation API follows the same content-protection model. **Your integration encrypts the secret before the request.** The endpoint `POST /api/v1/secrets` accepts an encrypted payload and a key commitment. It does not generate the secret's encryption key or offer a plaintext encryption service. The API key authorizes creation; it is separate from the AES key needed to decrypt a particular secret.

Client / integration	Secretus API
Generate a random 256-bit AES key and encrypt the payload using AES-GCM with a fresh 12-byte IV and a 128-bit tag.	Receive base64url(iv).base64url(ciphertext + tag), the SHA-256 key commitment and delivery options over HTTPS.
Keep the AES key locally. Send the API credential in the X-API-Key header, separately from the encrypted content.	Validate the request and store ciphertext. Return secretId and expiresAt, without a decryption key or complete share URL.
Build the complete share URL locally from secretId and the Base64url-encoded AES key, then share it with the recipient.	Serve the ciphertext through the same one-time Standard retrieval workflow. Decryption takes place in the recipient's browser.

Why the URL matters

`https://secretus.app/s/<secretId>#async_<base64url-key>`

Browsers omit the fragment after # from ordinary HTTP requests. The service can therefore receive the identifier without receiving the decryption key. The full URL is still sensitive: someone who obtains it can access the secret before consumption. Keep complete links out of logs, analytics and public channels; a key commitment is not a decryption key.

These protections depend on the shipped client or integration encrypting correctly and on trusted sending and receiving devices. They do not prevent a recipient from saving plaintext or protect future secrets from malicious client code. The same boundary applies to the web application, extension and API integration.