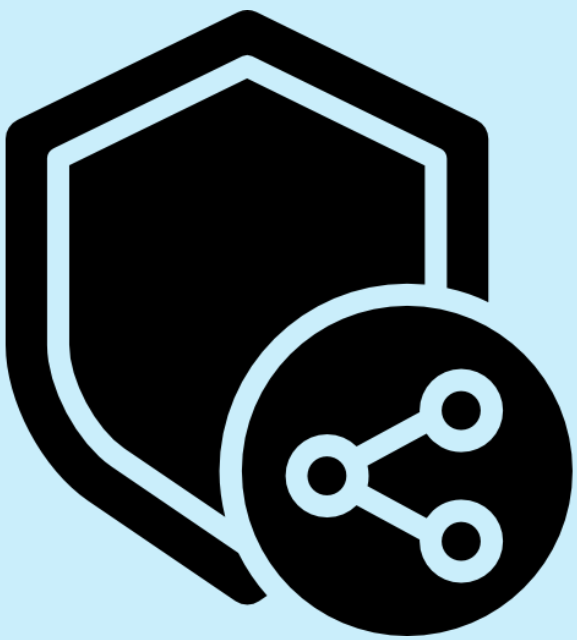
Security Documnent

Explanation and Description



Team : CacheME
Project : Secure File Sharing Platform

Version 1

Table of Contents

Secure File Sharing – Security Model & Cryptography Details (X3DH + AES)

1. Purpose & Scope

This document provides a one-stop explanation of how the website is secured at a high level and, in detail, how encryption works when SENDING a file. It focuses on the X3DH key agreement for establishing shared secrets between users and AES-based authenticated encryption for file confidentiality and integrity. Upload (personal storage) and Send/Share (to a recipient) are covered, with emphasis on Send/Share crypto details.

2. High-Level Security Model (Summary)

- Transport: TLS 1.3 on all client↔server hops; HSTS enforced at the gateway; WAF and rate-limiting in front of the API.
- Authentication (AuthN): Performed at login; the gateway or auth service verifies credentials and issues a short-lived JWT.
- Authorization (AuthZ): Enforced on every sensitive request (upload, send, download, delete).
- Client-side encryption: Files are encrypted in the browser using an AEAD cipher with a unique nonce per file.
- Storage: Only ciphertext + minimal metadata (nonce, hash, algorithm, recipients) are stored. Keys are never stored in plaintext; they are encrypted and stored in a secure vault (Hashicorp).
- Observability: Structured audit logs for security-relevant actions (Activity logs);

Below are high level sequence diagrams that show how personal secure upload in figure 1 and secure sending is done.

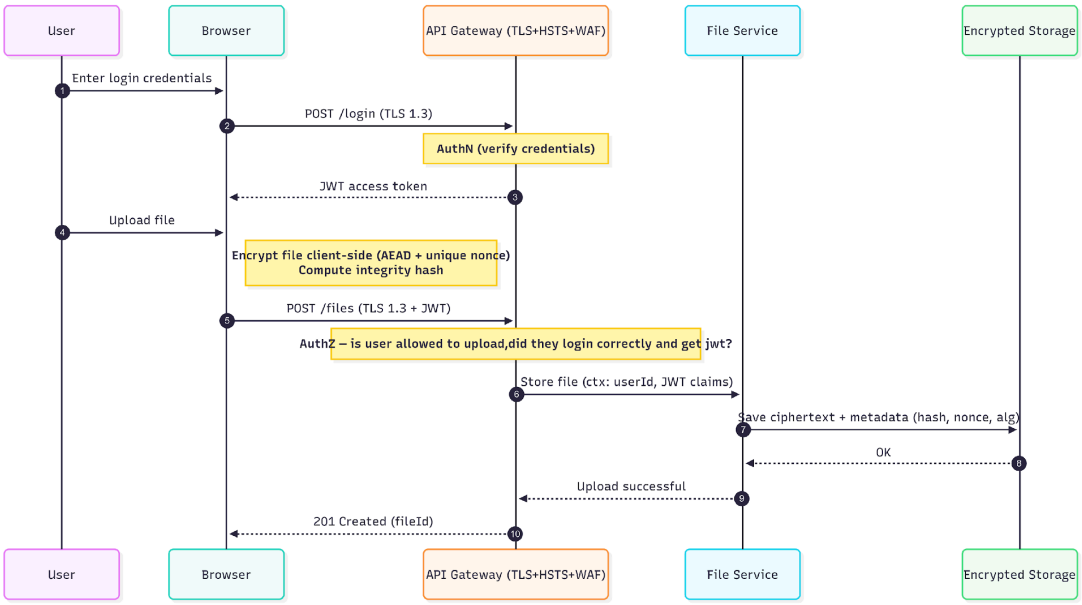


Figure 1: personal upload sequence diagram

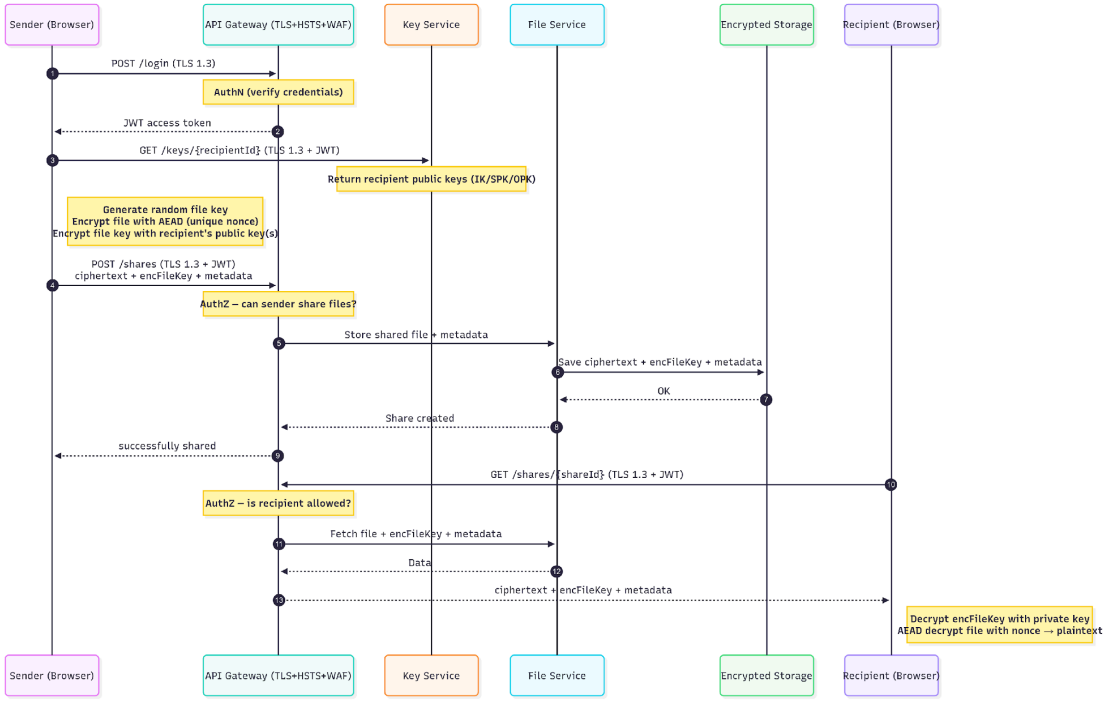


Figure 2: sending sequence diagram

3. Cryptography Overview

Algorithms & Parameters

- Asymmetric key agreement: X25519 (Curve25519 DH) used by X3DH for establishing a shared secret between Sender and Recipient.
- KDF: HKDF-SHA-256 to expand DH output(s) into a strong symmetric root key (RK), then derive sub-keys .
- Symmetric file encryption: AES-256-GCM (nonce: 96-bit). We Use AEAD for confidentiality + integrity.
- File key: 256-bit random key per file (which is never reused again).We do not derive file keys directly from long-term secrets; generate fresh randomness.
- Nonces: Unique per encryption operation. We never reuse a (key, nonce) pair.
- Hash: SHA-256 for file fingerprinting/integrity verification stored as metadata.
- Signatures : Ed25519 signatures for Signed PreKey (SPK) in X3DH.

4. X3DH Roles & Keys

Each user maintains:

- IK (Identity Key): static X25519 key pair (long-lived). Public part is shared; private part is secret.
- SPK (Signed PreKey): medium-term X25519 key pair signed by IK.
- OPK (One-Time PreKeys): a pool of single-use X25519 public keys for initial contacts; 50 are used to prevent high repetition.

The published bundle contains: IK_pub, SPK_pub, SPK_sig, and one OPK_pub.

5. X3DH Handshake – Sender → Recipient (Initial Send)

Goal: derive a shared secret SK that only Sender and Recipient can compute.

We use libsodium and its X3DH functions computes multiple DHs (X25519) and concatenates them before HKDF:

DH1 = DH(Sender_Ephemeral, Recipient_IK)

DH2 = DH(Sender_Ephemeral, Recipient_SPK)

DH3 = DH(Sender_IK, Recipient_SPK)

DH4 = DH(Sender_Ephemeral, Recipient_OPK)

Then:

SK_input = concat(DH1 || DH2 || DH3 || [DH4])

SK = HKDF-Extract-And-Expand(context-specific salt, info="X3DH-FileShare-v1", input=SK_input)

Notes:

- Sender_Ephemeral is a fresh X25519 key pair per session/send.
- Use constant-time operations provided by the crypto library.
- Validate SPK signature using Recipient_IK_pub before use.

6. File Encryption & Key Wrapping (Send/Share)

1) We generate a random 256-bit file key FK using a CSPRNG.

2)We encrypt the file using AES-GCM

- For AES-GCM: A 96-bit nonce is construct via CSPRNG (window.crypto.getRandomValues) and never reused with the same key.

3)We compute SHA-256 hash of the plaintext which is checked when downloading or receiving the file

4) Key wrapping for recipient(s):

- We derive a key-encryption key (KEK) from SK via HKDF: KEK = HKDF(SK, info="KEK-v1").
- Wrap FK: EncFK = AEAD_Encrypt(key=KEK, nonce=UniqueNonce2, plaintext=FK, aad=RecipientId|ShareId).

5) Store/send: {Ciphertext, Nonce, Alg, TAG (implicit in AEAD output), Hash, EncFK, Recipients, CreatedAt, UploaderId, AlgParams}.

- We never store FK or SK in plaintext.

7. Detailed Sending Flow (Crypto)

Sender:

a) Retrieve Recipient bundle {IK_pub, SPK_pub, SPK_sig, OPK_pub}; verify SPK_sig with IK_pub.

b) Generate Sender_Ephemeral (X25519). Compute DH1..DH4.

c) SK = HKDF(DHs, info="X3DH-FileShare-v1").

d) Generate FK (256-bit). Encrypt file with AEAD: (CT, TAG, Nonce1).

e) Derive KEK = HKDF(SK, info="KEK-v1"). Encrypt FK with AEAD: (EncFK, Nonce2).

f) Send to server (TLS 1.3 + JWT): CT, Nonce1, Alg, TAG, Hash, EncFK, Nonce2, AlgParams, RecipientId(s), Sender_Ephemeral_pub, identifiers.

Server:

g) Store ciphertext + metadata; audit the action; never log keys.

Recipient (on download):

h) Obtain Sender_Ephemeral_pub and Recipient's own private keys.

i) Recompute DH1..DH4 → SK; derive KEK; decrypt EncFK → FK.

j) AEAD-Decrypt(CT, Nonce1, FK, aad=MetadataHeader) → plaintext. Verify hash if stored.

Note: AEAD tag length is 16 bytes

8. Metadata Schema (example)

```
{
  "nonce": "<base64 12 or 24 bytes>",
  "tag": "<implicit in AEAD or stored if separate>",
  "hash": "sha256:<hex>",
  "fileSize": 123456,
  "senderEphemeralPub": "<base64 X25519>",
```

```
 "recipientIds": ["..."],
 "encFileKey": "<base64 AEAD ciphertext>",
 "encFileKeyNonce": "<base64>",
 "x3dhInfo": { "usedOPK": true, "bundleVersion": "vN" },
 "createdAt": "ISO-8601",
 "uploaderId": "<uuid>",
 "shareId": "<uuid or slug>"
}
```


9. Security Requirements & Checks

- Unique nonces per encryption. Enforce via library and code review.
- Refuse missing/invalid SPK signature; require fresh bundle version (staleness rules).
- Key separation: derive separate KEK and other subkeys with distinct HKDF info labels.
- Never include secrets in logs; scrub PII in logs where unneeded.
- Rate limit share endpoints; WAF rules for payload anomalies.
- Use Content Security Policy (CSP) to protect front-end crypto code from XSS.
- Backup/restore plans: ensure ciphertext + metadata remain consistent and verifiable after restore.

10. Threats & Mitigations (Send/Share)

- Man-in-the-middle / SSL stripping → TLS 1.3 everywhere + HSTS + preloaded domain (optional).
- Replay attacks on metadata → include ShareId/AAD and server-side idempotency checks; unique nonces.
- Key compromise (server) → E2EE: server never sees FK nor any of the private keys; at-rest encryption still applied for defense-in-depth.
- SPK staleness/impersonation → enforce SPK signature validation and rotation; cache busting and bundle versioning.
- XSS stealing tokens/keys → strict CSP, HttpOnly+SameSite cookies, avoid exposing raw keys to DOM.

- Side-channel risks → rely on vetted crypto libs; avoid custom primitives; constant-time ops.
- Malicious file uploads → validate MIME type and file size before encryption and storage.

11. Implementation Notes (Stack)

- Use libsodium (Web: libsodium.js) for X25519, HKDF, and XChaCha20-Poly1305 if preferred over AES-GCM.
- We use AES-GCM via the Web Crypto API which is 96-bit nonces, random per encryption; store nonce in metadata.
- Strong RNG: window.crypto.getRandomValues (web), sodium.randombytes_buf, or OS RNG on backend.
- JWT: short TTL (e.g., 5–15 min) + refresh rotation; audience/issuer checks; store in HttpOnly cookies or secure storage.
- Headers: Strict-Transport-Security, Content-Security-Policy, X-Content-Type-Options, Referrer-Policy.
- Audits: store actor, action, target, timestamp, IP, UA, requestId.