

---


# Computer Project: Report

---

000615056 BOTTON David

Pr. Absil R.

August 2025





# Contents

1	Security Overview . . . . .	2
1.1	Architecture . . . . .	2
1.2	Traffic . . . . .	2
1.3	Data . . . . .	2
1.4	Streams . . . . .	2
2	Quickstart Guide . . . . .	3
2.1	Boot the Stack . . . . .	3
2.2	API Documentation (Swagger) . . . . .	3
2.3	Register a Normal User . . . . .	3
2.4	Register a Trusted User . . . . .	3
2.5	Change Crypto Bundle Password . . . . .	3
2.6	Email Verification . . . . .	4
2.7	Login (2FA + Key Possession) . . . . .	4
2.8	Record Upload Stream . . . . .	4
2.9	Share a Stream (Normal → Trusted) . . . . .	4
2.10	View a Shared Stream (Trusted User) . . . . .	4
2.11	Revoke Access . . . . .	4
3	Pipelines . . . . .	5
3.1	Registration . . . . .	5
3.2	Certificate Issuance . . . . .	6
3.3	Login . . . . .	7
3.4	Stream Upload . . . . .	9
3.5	Stream Share . . . . .	9
3.6	Key Revocation . . . . .	11
4	Mission Checklist . . . . .	12
<hr/>		
A	Annex: Pipeline – Registration . . . . .	17
B	Annex: Pipeline – Certificate . . . . .	18
C	Annex: Pipeline – Login . . . . .	19
D	Annex: Pipeline – Stream . . . . .	20
E	Annex: Pipeline – Share . . . . .	21
F	Annex: Pipeline – Revocation . . . . .	22

## ABSTRACT

*This report details a security-focused infrastructure along its implementation in an end-to-end encrypted, PKI-based dashcam system. To be presented in this document; a quick-start guide with screenshots showing how users navigate the application, a systematic review of all components with an emphasis on the security aspect, the limitations and impediments encountered, a Q/A list resuming the system's security posture. Public `git` repository and initial setup available at the `GitLab README`.*

# 1 Security Overview

## 1.1 Architecture

SecCam runs each service in a dedicated Docker container and anchors trust in a local PKI with step-ca. The PKI bootstrap script generates a root and intermediate CA, then issues 2048-bit RSA leaf certificates for the Nginx gateway, the application server and other internal helpers; it also prepares an optional PKCS#12 bundle for full mutual-TLS. Inside the cluster, every socket is negotiated with service certificates that chain to this intermediate CA, so impersonation requires the CA's signing key, not just an access to a single host key. For external clients the trust anchor is pinned: the React bundle embeds the CA-root SHA-256 fingerprint at build time, rejecting any server that does not present the expected public key.

## 1.2 Traffic

All traffic enters through an Nginx reverse proxy that terminates TLS 1.3 exclusively and disables session tickets to foil passive resumption correlation. HSTS, CSP, Referrer-Policy and X-Frame-Options headers are set globally. Upstream calls from the proxy to the application server reuse TLS with client authentication: Nginx presents its own certificate and validates the server's certificate and SAN before forwarding, eliminating blind-trust hop vulnerabilities, container-level impersonation, or transparent MITM. Also, rate-limiting and header sanitation add basic DoS and injection resistance. After TLS is established, the session identifier is carried in a signed JWT that the server sets as an encrypted, httpOnly, sameSite cookie, so its exfiltration is greatly mitigated.

## 1.3 Data

The browser creates its own cryptographic material before any upload: a RSA key pair for wrapping, a fresh AES-GCM content key, and a HMAC-SHA-256 key for explicit integrity separation. User profile fields and each video segment are encrypted with AES-GCM and tagged with the HMAC, only ciphertext and MAC leave the client. Both symmetric keys are wrapped with the user's RSA public key (RSA-OAEP) and stored alongside the ciphertext. The private key is saved only as an encrypted JWK inside the user-controlled .sec.json bundle and is decrypted into memory on demand; it never appears in cleartext on disk or in server logs. Because the content key is user-specific and the server never sees it unwrapped, a breach of the database yields nothing usable without the client's private key.

## 1.4 Streams

Trusted users undergo a dual-CSR workflow: one certificate asserts their organizational identity, the other authorizes platform access. After the server enacts the signature of both CSRs through the intermediate CA, these certificates allow normal users to share a stream key simply by wrapping it with the trusted user's certificate public key. Server stores only the wrapped blob and when the trusted party requests the video, the server hands over that blob and the encrypted chunks. Decryption happens entirely in the client, leaving the server permanently blind. Revocation reduces to deleting the stored wrapped key, after which the former recipient can no longer unwrap future AES keys. All sharing, verification and revocation therefore occur at the cryptographic edge.

## 2 Quickstart Guide

### 2.1 Boot the Stack

1. From the project root, bootstrap the local PKI and generate fresh service certificates:  
`./cmd/1-pki,setup.sh.Thisproducesstep-root.pem;importitintoyourOS/browsertruststoreb`
2. Start (or rebuild) the full stack with TLS and health checks: `./cmd/2-total,un.sh.GitHub`
3. Open the web UI at `https://localhost:3443/` All traffic goes through the TLS reverse proxy. GitHub

### 2.2 API Documentation (Swagger)

1. Browse the live API at `https://localhost:3443/api-docs` Use this for the exact request/response shapes, auth requirements, and example payloads for every endpoint. GitHub

### 2.3 Register a Normal User

1. In the UI, go to *Register* at `https://localhost:3443/`
2. Enter a unique username and a reachable email.
3. Confirm: key material is generated client-side; private keys never leave the browser. GitHub
4. When prompted, download your encrypted Crypto Passport (`*.sec.json`) and store it safely (offline backup recommended).

### 2.4 Register a Trusted User

1. In *Register*, switch to the *Trusted* tab.
2. Provide identity fields (first name, last name, organization, country).
3. Complete registration; your trusted-user credentials are bound to the declared identity via the PKI flow. GitHub
4. Download and securely store your `*.sec.json` Crypto Passport.

### 2.5 Change Crypto Bundle Password

1. Open *Change Bundle Password* in the UI.
2. Select your current encrypted bundle (`.sec.json`) and enter its current password.
3. Choose a strong new password (recommend  $\geq 12$  characters) and confirm.
4. The app decrypts the bundle *locally* and re-encrypts it with the new password.
5. Download the new `.sec.json` and replace your old copy; update any backups.

**Note** If you lose the bundle password, it cannot be recovered. The server never sees or stores it; treat the bundle like a hardware token. GitHub

## 2.6 Email Verification

1. Check your inbox for the verification email sent during registration.
2. Click the verification link to activate your account before first login. (The stack exercises a dev mail service during boot; ensure you completed the TLS setup in the boot scripts.)  
GitHub

## 2.7 Login (2FA + Key Possession)

1. Open *Login* at `https://localhost:3443/`  
.- 2. Select and decrypt your `*.sec.json` in the browser (bundle password required).
- 3. Enter your TOTP code from your authenticator app.
- 4. On success, you are authenticated for API access; decrypted keys remain only in browser memory.

## 2.8 Record Upload Stream

1. Navigate to *Stream / Record*.
2. Allow camera/microphone permissions.
3. Start recording: chunks are encrypted client-side and uploaded over the TLS reverse proxy at `https://localhost:3443/`  
.- 4. GitHub

## 2.9 Share a Stream (Normal → Trusted)

1. Open *Share / Trust*.
2. Select a trusted user; the client wraps your stream keys to their public key (key wrapping).
3. Confirm: the server stores only the wrapped keys; plaintext keys are never exposed.  
GitHub

## 2.10 View a Shared Stream (Trusted User)

1. Log in as a trusted user and load your `*.sec.json`.
2. Open *Shared Streams*.
3. Select a stream: wrapped keys are fetched and unwrapped locally; chunks are decrypted in the browser.

## 2.11 Revoke Access

1. As the normal user, open *Share / Trust*.
2. Click *Untrust* for the target trusted user.

3. The wrapped-keys entry is deleted; future access is blocked at the server and decryption fails client-side.

## 3 Pipelines

### 3.1 Registration

**Goal** Establish identity and keys without trusting the server with plaintext or private keys; deliver an encrypted, client-held bundle required for all future sessions.

#### Flow

1. **Client key generation.** The browser generates a user RSA key pair (identity), a symmetric AES-GCM key (content), and an HMAC key (integrity). It builds a PKCS#10 CSR with the public key and identifiers.
2. **Client-side protection.** PII is encrypted with AES-GCM; an HMAC-SHA-256 is computed over sensitive fields/records. The AES and HMAC keys are wrapped to the user's RSA public key via RSA-OAEP (the server stores only wrapped blobs).
3. **Submission over pinned TLS.** The client sends { CSR(s), encrypted PII + HMACs, wrapped keys } to `/api/user/register` over TLS 1.3. The stack is provisioned with a local CA and strict TLS; mTLS support exists but is off by default for development.<sup>1</sup>
4. **CA signing (server → CA).** The server forwards the CSR(s) to the internal *Step-CA* and receives signed X.509 leaf certificate(s) (user; and org for trusted users). Private keys never leave the client.
5. **Persist minimal state.** The database stores encrypted PII + HMACs, the user's public key, and the RSA-OAEP-wrapped AES/HMAC keys. No private keys are stored server-side.
6. **2FA bootstrap.** The server creates a TOTP secret, returns a QR URI, and stores the secret encrypted; login will require a valid TOTP code.
7. **Client export (required).** The browser assembles one encrypted crypto bundle (`*.sec.json`) holding the user's private key(s), public cert(s), and session keys as needed, protected with a password-derived key. The bundle is downloaded and only loaded transiently in app state at login; it is not uploaded back to the server.

**Registration.** For the complete flow, see Annex: Pipeline – Registration (p. 17).

#### Technology choices & justification

**TLS 1.3 end-to-end; local PKI (Step-CA).** The compose stack is provisioned with a self-contained CA; services terminate TLS 1.3 and bootstrap scripts verify chains and cipher strength (AES-256-GCM). This removes external CAs from the trust boundary and allows strict pinning/chain checks. mTLS is supported; it is disabled by default in development and can be enforced at NGINX when needed. *Justification:* modern cipher suites, smallest

---

<sup>1</sup>See repository notes.



handshake surface, and a PKI you control; easy revocation/rotation and no plaintext transit.

**RSA-OAEP (SHA-256) for key wrapping.** Wrapping the per-user AES and HMAC keys to the user’s public key keeps the server permanently blind; only the client’s private key can unwrap. *Justification:* OAEP with SHA-256 is the standard, misuse-resistant KEM for RSA in browsers; interops cleanly with WebCrypto and X.509.

**AES-GCM for PII and video chunks.** GCM gives confidentiality and integrity (auth tag) per record/chunk, so any DB tampering or transit alteration fails open on the client. *Justification:* single-pass AEAD, hardware-accelerated in browsers; perfect for chunked streaming and small PII blobs.

**HMAC-SHA-256 on stored records.** In addition to GCM tags, the system tracks HMACs for critical fields (email/identity). *Justification:* the client can detect silent server/DB edits even when fields are re-encrypted; cheap and widely audited.

**PKCS#10 CSR + Step-CA issuance.** Users (and orgs for “trusted” accounts) submit CSRs from the browser; the CA signs and returns X.509 certs bound to those public keys. *Justification:* proves possession of key material at enrollment; standard path to mTLS, signing, and verifiable identity without exposing private keys.

**Encrypted .sec.json bundle (client custody).** The only copy of private keys lives in an encrypted JSON, protected with a password-derived key (WebCrypto), downloaded to the user and loaded only in memory at login. *Justification:* keeps the server out of key custody, turns login into MFA + key possession, and simplifies multi-device usage by letting users carry/import their bundle.<sup>2</sup>

**TOTP 2FA (QR seed, encrypted at rest).** Login requires something-you-have (TOTP device) in addition to the encrypted key bundle. *Justification:* mitigates token theft and enlarges the attack surface an adversary must control.

**NGINX reverse proxy as choke point.** Centralizes TLS policy (TLS 1.3 only, session tickets off, strong ciphers) and can enforce mTLS if flipped on in production. *Justification:* one place to harden protocol knobs and rate-limit; consistent policy across services.

---

## 3.2 Certificate Issuance

**Goal** Bind the client-generated public keys to the platform PKI with a CA-signed X.509, while keeping private keys strictly client-side.

### Flow

1. **Client CSR(s).** The browser that just generated the key pair(s) in §3.1 builds PKCS#10 CSR(s) over the public key(s); trusted users include organization attributes in a separate CSR. The private keys remain in memory on the client only.

---

<sup>2</sup>Client-side PKCS#12 is brittle across browsers: see node-forge’s long-standing ECDSA PKCS#12 bug (git issue #925) and broader ECC support gap (#532). Step-CA defaults to ECDSA and requires a manual process to get a full RSA chain. PKIjs also has open PKCS#12 parsing/import issues. Given these unresolved interop constraints, the encrypted .sec.json bundle as well as optional mTLS are standard in this project.

2. **CSR intake (server).** The server receives the CSR payload over pinned TLS (same channel and checks as registration) and validates basic claims (e.g., presence of identifiers expected for the account type). It never interacts with private key material.
3. **CA signing (Step-CA).** The server invokes the internal Smallstep CA to sign the CSR(s), producing leaf certificate(s). This is performed with the Step CLI (`step ca sign / step certificate sign`) against the platform’s intermediate CA; validity and options are controlled by the CLI flags.
4. **Chain & policy.** Issued leafs chain to the project’s own root/intermediate created by the PKI setup script, which the stack already trusts; the build/run script explicitly verifies chains and ciphering end-to-end as part of the boot checks.
5. **Delivery to client.** The signed certificate(s) (PEM) are returned in the same registration response path and added by the client to its encrypted `.sec.json` bundle (see §3.1); no plaintext keys are transmitted.
6. **Persist public material only.** The backend may store the certificate(s) and public key with the user’s record for lookup/auditing; private keys are never persisted server-side (consistent with registration’s “persist minimal state”).

**Certificate Issuance.** For certificate details and chain handling, see Annex: Pipeline – Certificate (p. 18).

## Technology choices & justification

**Smallstep step-ca (leaf signing).** Using `step ca sign / step certificate sign` keeps issuance simple and auditable, with first-class support for PKCS#10 CSRs, explicit `notBefore/notAfter`, and templating. The CA runs as an intermediate under our own root; the stack’s PKI scripts build and verify that trust chain locally. *Justification:* standard CSR in, leaf out; offline/controlled issuance; minimal attack surface and easy rotation.

**Project-local root/intermediate.** The repository’s PKI setup initializes a self-contained CA, issues service leafs, and bakes the root into the stack; the run script then verifies the full chain and TLS properties during boot. *Justification:* removes external CAs from the trust boundary and enables strict pinning/chain checks across services.

**Client-side CSRs.** CSRs are crafted and signed where the keys live (the browser), so possession is proven to the CA without exposing private keys. *Justification:* aligns with X.509 enrollment best practice and the platform’s “keys never leave the client” design.

**Pinned TLS on issuance path.** CSR submission and cert delivery reuse the same pinned TLS channel as registration; any chain or fingerprint mismatch aborts the flow. *Justification:* prevents MITM during identity binding and cert pickup.

---

## 3.3 Login

**Goal** Establish an authenticated session without exposing private keys: OTP is verified server-side; keys are unwrapped client-side from the encrypted bundle created in §3.1, and only held transiently in memory.

## Flow

1. **Inputs (client).** The user provides *username* and current 6-digit TOTP, then loads the encrypted crypto bundle (*\*.sec.json*) and enters its password. The browser decrypts the bundle locally (PBKDF2  $\rightarrow$  AES-GCM) to recover the user’s private key(s) and certificates in memory.
2. **OTP verification (server).** The client calls `/api/user/login` with `{username, TOTP}`. The server rejects unverified emails, decrypts the stored TOTP seed, and verifies the code (time windowed). On failure it returns 4xx; on success it authenticates the user.
3. **Session issuance.** The server creates a short-lived JWT (*username + role*), signs it, encrypts the token payload, and sets it as a cookie with `HttpOnly, Secure, SameSite=strict`. No token is returned in the body.
4. **Fetch encrypted account material.** The client calls `/api/user/current`; the server validates the cookie and returns only public/opaque fields: the RSA-OAEP-wrapped symmetric key and HMAC key (from §3.1), the encrypted PII blob(s), and stored HMAC(s).
5. **Local unwrapping and checks (client).** Using the private RSA key from the bundle, the client RSA-OAEP-decrypts the wrapped symmetric key and HMAC key, then holds them in volatile state. It decrypts PII with AES-GCM and validates HMACs; any mismatch aborts the session. (Trusted users additionally validate their X.509 chain against the platform root from §3.2.)
6. **Established session.** Subsequent API calls use the cookie; all sensitive operations on data use the in-memory symmetric/HMAC keys. Logout clears the cookie and zeroizes client keys; token expiry enforces re-authentication.

**Login.** For the device login and key-unwrap process, see Annex: Pipeline – Login (p. 19).

## Technology choices & justification

**MFA by design.** Login combines TOTP (server-verified) with *key possession*: the encrypted *.sec.json* bundle must be locally decrypted to unwrap session keys. This binds access to both a time factor and the user’s client-held keys.

**Client-side unwrapping (RSA-OAEP).** Only the browser uses the private key to decrypt the wrapped AES/HMAC; the server never sees plaintext keys. This preserves the “server-blind” model introduced at registration.

**Hardened session cookie.** The JWT is short-lived and stored in an `HttpOnly, Secure, SameSite=strict` cookie; the server also encrypts the cookie payload before setting it. *Justification:* resists XSS exfiltration and CSRF; the token is opaque to client JS and never appears in URLs.

**Pinned TLS continuity.** The login path reuses the pinned TLS channel from §3.1, so OTP and account material travel under an authenticated transport.

**End-to-end integrity.** AES-GCM tags protect decrypted PII; record-level HMACs detect any DB tampering before keys are accepted into the session.

### 3.4 Stream Upload

**Goal** Ingest live camera data without exposing content to the server: video is chunked and AES-GCM-encrypted in the browser with user-session keys from §3.3; the server stores ciphertext + metadata only.

#### Flow

1. **Capture (client).** The browser records from the webcam/microphone (MediaRecorder). Chunks are produced at short, fixed intervals.
2. **Per-chunk encryption (client).** For each Blob, the client derives a fresh 96-bit IV and encrypts with the user's AES-GCM session key (from §3.3). The result is `{iv, ciphertext_with_tag}`. No plaintext leaves the browser.
3. **Authenticated upload.** The client POSTs `{encryptedChunk, metadata(videoName, chunkIndex, ts, size)}` to `/api/video/streaming` over pinned TLS, with the `HttpOnly` session cookie set (§3.3).
4. **Store (server).** The API authenticates via the cookie, does not decrypt, and persists: (i) a *Video* entry (upsert by owner+name) and (ii) a *VideoChunk* document containing the encrypted payload + metadata. Each chunk is indexed for ordered retrieval.
5. **Ack + repeat.** The server returns 200; the client continues the encrypt→upload loop. If any chunk is altered at rest, client decryption fails (GCM tag) on playback.

**Stream.** For video encryption and streaming setup, see Annex: Pipeline – Stream (p. 20).

#### Technology choices & justification

**AES-GCM per chunk.** Authenticated encryption amortized per segment: confidentiality + integrity via tag; minimizes blast radius on loss/corruption; ideal for streaming and resumable fetch.

**Fresh IVs.** New 96-bit IV per chunk avoids nonce reuse under a fixed key; suits Web Crypto and keeps records independent.

**Pinned TLS + cookie auth.** Upload rides the same pinned TLS channel and `HttpOnly/Same-Site` cookie from §3.3; prevents token exfiltration and CSRF on the hot path.

**Opaque server.** The server never receives keys or plaintext; it stores ciphertext + sequence metadata. Integrity is enforced client-side by GCM, not trusted storage.

**Small, frequent chunks.** Short intervals reduce data at risk per request, improve UX under flaky links, and ease parallel retry/replay protection at the application layer.

---

### 3.5 Stream Share

**Goal** Grant a *trusted user* access to an existing encrypted stream without disclosing plaintext or keys to the server: the owner wraps their symmetric keys to the trusted user's X.509 public key; the trusted user unwraps locally and decrypts chunks fetched from the server.

## Flow

1. **Select target (owner, client).** The normal user (stream owner) selects a target trusted user from the list exposed by the server (trusted users carry CA-signed X.509 certs from §3.2).
2. **Recover own session keys (owner, client).** The owner requests their stored `encrypted_symmetric_key` and `encrypted_hmac_key` (cookie-authenticated). Using the private RSA key from the local `*.sec.json`, the browser RSA-OAEP-decrypts both and holds the raw AES/HMAC in memory (as in §3.3).
3. **Wrap to trusted user (owner, client).** The owner extracts the trusted user’s RSA public key from the certificate and RSA-OAEP-encrypts the AES and HMAC keys, yielding `wrappedSymmetricKey` and `wrappedHmacKey`. No plaintext keys or chunks are exposed.
4. **Create/Update trust (server).** The owner POSTs `{trustedUserId, wrappedSymmetricKey, wrappedHmacKey}` to `/api/keywrap` over pinned TLS with the session cookie. The server verifies the caller is a *normal* user, the target is a registered *trusted* user with a certificate, and stores/updates the mapping in a key-wrapping record (owner  $\rightarrow$  trusted).
5. **List shares (trusted, client).** A trusted user queries `/api/video/trustedList`; the server returns owners/videos that have an active wrapping toward this trusted user.
6. **Obtain wrapped keys (trusted, client).** When choosing an owner, the trusted user calls `/api/keywrap/{owner}/keys`. The server checks that a wrapping exists (owner  $\rightarrow$  this trusted user) and returns the stored `wrappedSymmetricKey` and `wrappedHmacKey`.
7. **Unwrap locally (trusted, client).** Using their private RSA key from the local `*.sec.json`, the trusted user RSA-OAEP-decrypts the wrapped keys to recover the raw AES/HMAC in memory.
8. **Fetch encrypted chunks (trusted, client).** The trusted user requests chunks via `/api/video/{videoName}`. The server authenticates (cookie), authorizes by checking the wrapping, and returns the stored ciphertext chunks + metadata (from §3.4).
9. **Client-side decryption & playback (trusted, client).** The browser AES-GCM-decrypts each chunk with the recovered AES key (GCM tag enforces integrity), reassembles, and plays. The server remains blind to both keys and content.

**Share.** For encrypted sharing to trusted users, see Annex: Pipeline – Share (p. 21).

## Technology choices & justification

**Public-key wrapping (RSA-OAEP).** Owners encrypt the stream’s AES/HMAC to the trusted user’s certificate public key; only the corresponding private key can recover them. This preserves end-to-end confidentiality while using standard X.509 material issued in §3.2.

**Server as blind intermediary.** The backend stores opaque wrapped keys and ciphertext chunks, enforcing access solely by the wrapping relation; it never holds plaintext keys or video, aligning with the “server-blind” model from §3.1–§3.4.

**Pinned TLS + cookie auth.** All share operations reuse the same pinned TLS transport and `HttpOnly/SameSite` cookie from §3.3, preventing token exfiltration and cross-site forgery on the key path.

**GCM integrity at the edge.** Decryption in the trusted user’s browser validates each chunk’s GCM tag; storage tampering is detected client-side without trusting server integrity.

**Revocation hook** Removing a wrapping entry (see §3.6) immediately blocks further key or chunk retrieval for that pair; the server denies access on subsequent requests, and only clients still holding a plaintext key in memory could decrypt already-fetched data.

---

### 3.6 Key Revocation

**Goal** Immediately rescind a previously granted share without touching plaintext or re-encrypting stored data: remove the owner→trusted wrapping so the server denies further key/chunk retrieval, while private keys and content keys remain client-held.

#### Flow

1. **Untrust (owner, client).** The owner selects a previously trusted user and issues `DELETE /api/keywrap/{trustedUserId}` over pinned TLS with the session cookie (same auth path as §3.3).
2. **Verify & delete (server).** The API authenticates the request, enforces that the caller is a *normal* user, locates the key-wrapping record (owner→trusted), deletes it, and returns 200 “removed”.
3. **Owner UX update (client).** The UI removes the trusted user from the share list; no content keys are uploaded or rotated automatically (the server never had them in plaintext).
4. **Effect on trusted user (server+client).** Subsequent requests from the trusted user fail:
  - (a) `GET /api/keywrap/{owner}/keys` → no wrapping ⇒ 404/403.
  - (b) `GET /api/video/{video}/{owner}/chunks` ⇒ authz check fails (no wrapping) ⇒ 403.

Without wrapped keys from the server, a trusted user cannot (re)obtain the AES/HMAC; any keys left only in volatile client memory expire with the session.

5. **(Optional) key rotation (owner, client).** If the owner wants to invalidate any key material a trusted user might still hold in-memory, they can re-enroll/rotate and re-share; rotation is out-of-band of revocation and not required for the server to block access.

**Revocation.** For device and certificate revocation, see Annex: Pipeline – Revocation (p. 22).

#### Technology choices & justification

**Authorization as gate.** Access to both key material and ciphertext is conditioned on the existence of an owner→trusted wrapping; deleting that single record cuts off all future reads without touching stored blobs.

**Server-blind state.** The server never stores plaintext AES/HMAC, so revocation is a cheap metadata change (delete wrapping) rather than a cryptographic re-write.

**Pinned TLS + cookie session.** Revocation uses the same pinned transport and HttpOnly/Same-Site cookie as prior pipelines, ensuring only the legitimate owner can revoke.

**Fail-closed at the edge.** Even if ciphertext remains in storage, the trusted user cannot decrypt new content without fresh wrapped keys; any already-fetched chunks still require a live in-memory key on the client side.

## 4 Mission Checklist

### 1. Do I properly ensure confidentiality?

All data in transit is protected by strong TLS (a Nginx front allows only TLS 1.3 with AES-256-GCM cipher suite). The system uses its own PKI: each service (client, server, mail) has a unique X.509 certificate, and the client enforces certificate pinning on every API call. Sensitive user data is end-to-end encrypted: for example, user fields are stored only in encrypted form (AES-256-GCM) in the database (except for the username), and each has an HMAC for integrity verification. Video content is never stored in plaintext on the server, video chunks are encrypted on the client side and uploaded already encrypted. Server-side secrets are handled carefully (random CA password and encryption keys are generated at setup and not hardcoded).

## 2. Did I harden my authentication scheme?

Authentication requires both a verified identity and possession of client-held keys. Registration ends with the client generating keys and downloading a single encrypted crypto bundle ("sec.json") that contains the user's certs/keys; that bundle is required at login and is only loaded transiently in React state, never persisted in plaintext. There are no reusable passwords; users must verify email and then present a TOTP code (speakeasy), and the TOTP secret is stored server-side only in encrypted form (AES-GCM). Upon successful login, a short-lived JWT is issued and stored in an HTTP-only, secure, same-site AES-encrypted cookie with attributes *Secure*, *SameSite=strict*, *HttpOnly*. For completeness: the PKI script can also emit a browser .p12; mTLS is supported but disabled by default due to Javascript .p12 generation impediments during development.

### 3. Do I properly ensure integrity of stored data?

The system ensures that stored data cannot be modified undetected. Each user has a unique HMAC key used to protect critical fields: for example, the database stores `hmac_username`, `hmac_email`, etc., alongside the encrypted values. Whenever user data is retrieved, the client verifies these HMACs to detect any tampering before decrypting the fields. Videos are stored in chunks with their original filenames/IDs recorded, and any addition or removal of chunks would be evident to the client by missing sequence numbers or HMAC mismatches (each chunk is encrypted with an authenticated cipher, so modification is detectable). The logging subsystem also guards integrity: !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! every log entry is appended with an HMAC computed using a server-side secret. This means if an attacker altered or forged a log record in the database there will be evidence of integrity breach.

#### 4. Do I properly ensure the integrity of sequences of items?

|||||

|||||

[illegible][illegible][illegible]

Security does not hinge on hidden implementation details or obscurity. The design follows standard cryptographic practices and avoids any secret sauce beyond proper secret keys and credentials. All security features use well-known algorithms (TLS, RSA, AES-GCM, HMAC-SHA256, JWT) rather than proprietary or obscured mechanisms. Critical secrets



like the certificate authority password, JWT signing key, TOTP encryption key, etc., are generated at deployment time or stored in environment variables and not hardcoded in the source. There is no reliance on obscurity such as hidden endpoints or constant values that must remain unknown to attackers, API routes are openly defined and documented in the Swagger docs with access control enforced via proper authentication, not by keeping them secret.

## **7. Am I vulnerable to injection?**

No, the codebase isn't vulnerable to injection attacks (NoSQL injection, command injection or XSS) given the frameworks and mitigations in use. The server uses Mongoose for all database access, passing user-supplied values as parameters to queries (not constructing queries from strings). There are also explicit sanitization measures; for instance, the logging utility escapes or removes dangerous characters from log messages (replacing newlines, %, <, > etc.) to prevent log injection or HTML injection in logs. On the client side, the UI is a Next.js React app which inherently escapes content, and a strict Content Security Policy is set to only allow scripts from self, mitigating XSS risks. There is no use of eval or unsanitized dynamic HTML in the server or client code. Also, all inbound API data is parsed as JSON and not directly concatenated into commands or queries, and the few filesystem operations like writing certificates use sanitized, controlled inputs (organization names are used in file names but are trimmed and cannot break out of the intended directory).

## **8. Am I vulnerable to data remanence attacks?**

The PKI setup script generates a CA password and shreds the temporary file containing it immediately after use, so it cannot be recovered from disk later. The teardown script (0-total\_reset.sh) thoroughly wipes data: it deletes database contents for all collections (users, videos, etc.) on reset, removes Docker volumes, and cleans up all certificate and key files from the project directory (making them writable and then deleting, even using sudo if necessary to remove root-owned leftovers). Uploaded videos are stored only in DB and can be removed via the provided delete function. Encryption keys are held in the user's encrypted bundle and are not transmitted or stored on the server in plaintext. Moreover, secrets like the JWT token are kept only in an encrypted HttpOnly cookie, cleared on logout or expiry.

## **9. Am I vulnerable to fraudulent request forgery?**

CSRF is mitigated by the use of SameSite cookies and strict CORS policies. The authentication cookie is set with SameSite=strict and Secure flags, which means browsers will not include it in cross-origin requests at all. Additionally, the server's CORS configuration only allows requests from the same origin. Because of this, a malicious third-party page could not force a user's browser to perform authenticated actions on the site. The use of an encrypted, HttpOnly cookie for the JWT also means JavaScript can't be tricked into sending the token elsewhere, and there is no reliance on URL parameters or referrer headers for auth. Furthermore, all state-changing endpoints (and sensitive reads) require a valid JWT, so even if a forgery is attempted, without the token it will be rejected. On the server side, no obvious SSRF vectors exist because the server doesn't fetch external URLs based on user input (it only contacts the local CA and uses fixed internal addresses).

## **10. Am I monitoring enough user activity for detection/forensics?**

The system includes monitoring and auditing capabilities to detect malicious activity or facilitate post-attack analysis. All significant actions (registration failures, login attempts, video uploads, deletions, etc.) are logged via a central logger module to both console and a MongoDB logs collection. Each log entry is annotated with a timestamp, severity level, and an HMAC hash of the message content. This design not only provides an audit trail of user and system events, but the HMAC also ensures that an attacker cannot surreptitiously alter log entries without detection. In addition to logging, the Nginx layer has rate limiting configured for various request types. These limits can help detect or throttle malicious behaviors such as brute-force attempts or flooding, and they indicate abnormal activity if triggered (too many API calls in a short time might surface in Nginx logs or metrics). There is no real-time alerting system, but the logs and built-in limitations provide a foundation for detecting malicious intent during a review or forensic analysis.

#### **11. Am I using components with known vulnerabilities?**

No, the project does not include components with known public vulnerabilities as of the latest commit. The dependency stack is modern: it runs on Node.js 20 (Alpine), uses up-to-date libraries like Express 4.x, Mongoose 7.x, Next.js 13, and well-maintained crypto libraries (jsonwebtoken, speakeasy, node-forge etc.). The Docker images chosen are current versions (MongoDB 8.0.11, Smallstep CA 0.28.4, Mailpit latest), none of which are flagged for known security issues at current time. Important server-side libraries (node-forge for certificate parsing and jsonwebtoken for JWTs) are used without usage of deprecated or unsafe functions. The project's npm ci process in Dockerfile installs exact versions from a lockfile. All javascript packages have been audited and report no known vulnerabilities.

#### **12. Is my system updated?**

The development documentation specifies recent platform versions, the Docker build pulls the latest package definitions at build time. The use of Alpine-based images and explicit package installs helps minimize unused libraries and quickly integrate security fixes (20-alpine image will include recent security patches for both Node and Alpine Linux) and the Docker Compose configuration references current images. Critical services such as the certificate authority (Smallstep) and Mailpit are also at recent releases. Moreover, the project's README encourages updating system dependencies and even provides troubleshooting steps for version mismatches.

#### **13. Is my access control broken (OWASP 10)?**

The application enforces authorization at two layers: API policy and cryptography. Every protected route sits behind JWT middleware and explicit role checks; trusted vs. regular users have separate privileges, and listing/reading another user's video requires a matching key-wrapping record. Possession of the user's encrypted ".sec.json" bundle (certs/keys) is required client-side to unwrap the shared AES stream key; without that bundle, ciphertext from any endpoint remains unusable. Endpoints meant for normal users explicitly reject requests if the JWT belongs to a trusted user, and vice-versa. When fetching a video, the server ensures the requester is the owner or a trusted user with a valid wrap, else returns 403. There are no unprotected direct object references; even /api-docs is served over HTTPS and respects the same cookie auth. In practice, decryption is impossible without the client-held keys, even if an API listing were exposed.

## 14. Is my authentication broken (OWASP 10)?

No, authentication requires verified email, a TOTP code, and possession of the encrypted “sec.json” bundle. Registration ends with the client generating keys and downloading that bundle; at login it must be imported and is only loaded transiently in React state (never persisted in plaintext, and the server is never shown its content). The TOTP secret is per-user and stored server-side only in encrypted form; codes are verified with speakeasy. Successful auth issues a short-lived JWT set in a hardened cookie, and the payload is additionally encrypted. This design mitigates session hijacking/fixation and CSRF. No reusable passwords are used or stored at all, removing weak-password and hashing pitfalls. One area to improve is brute-force resistance (e.g., lockout/backoff on repeated bad TOTP), though proxy rate-limits and short code windows are already mitigating that issue.

## 15. Are my security features misconfigured (OWASP 10)?

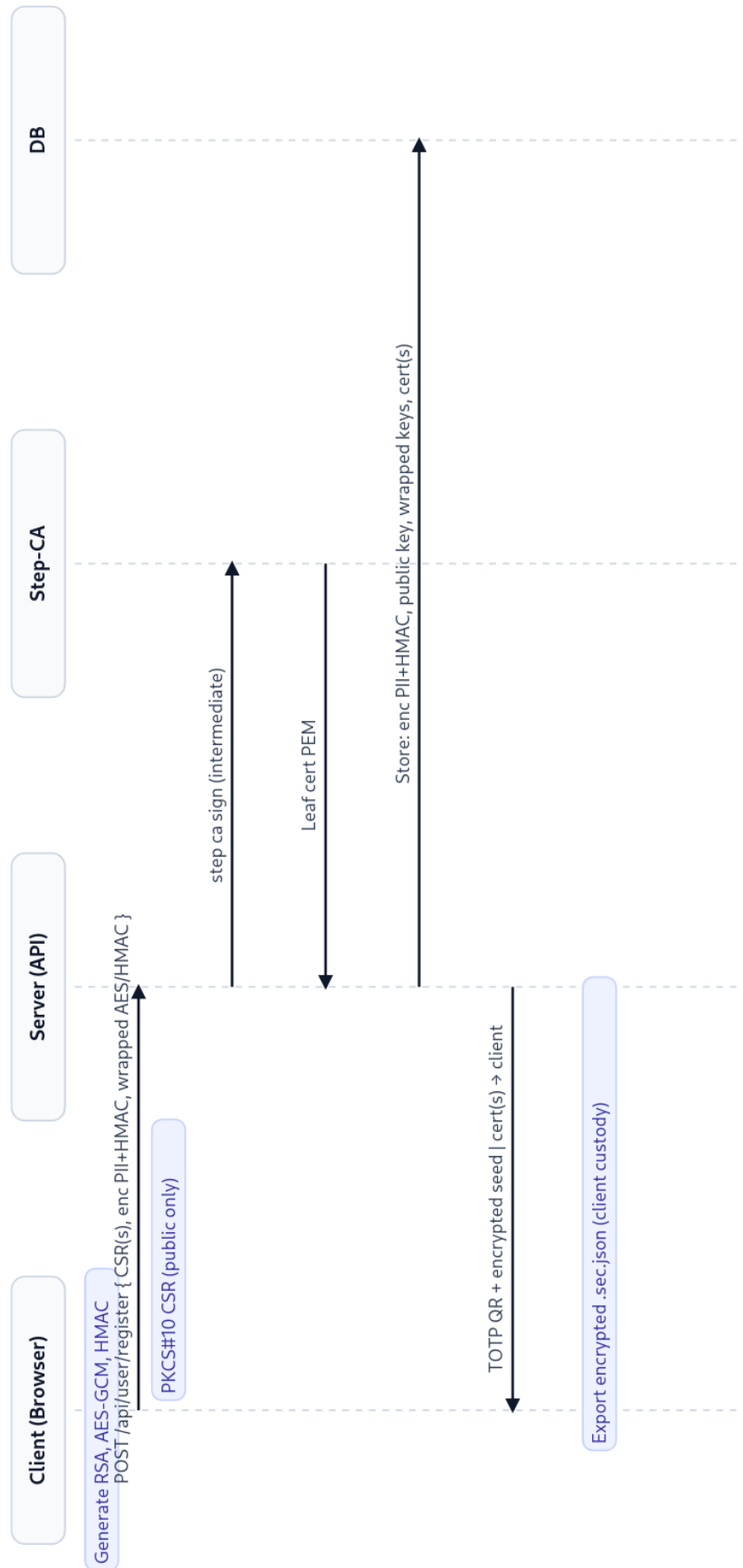
[illegible][illegible]

.....

## A Annex: Pipeline – Registration

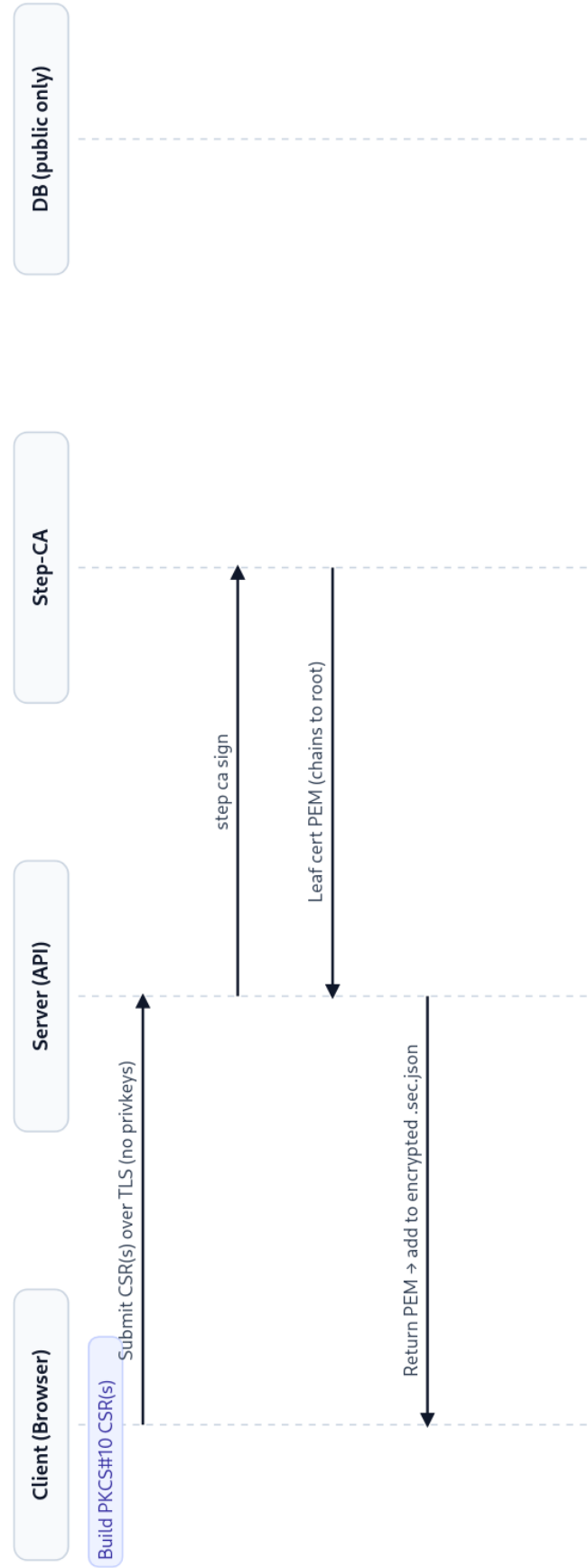
### 3.1 Registration

Client keygen → CSR → Step-CA → minimal persistence → TOTP → encrypted .sec.json



B    Annex: Pipeline – Certificate

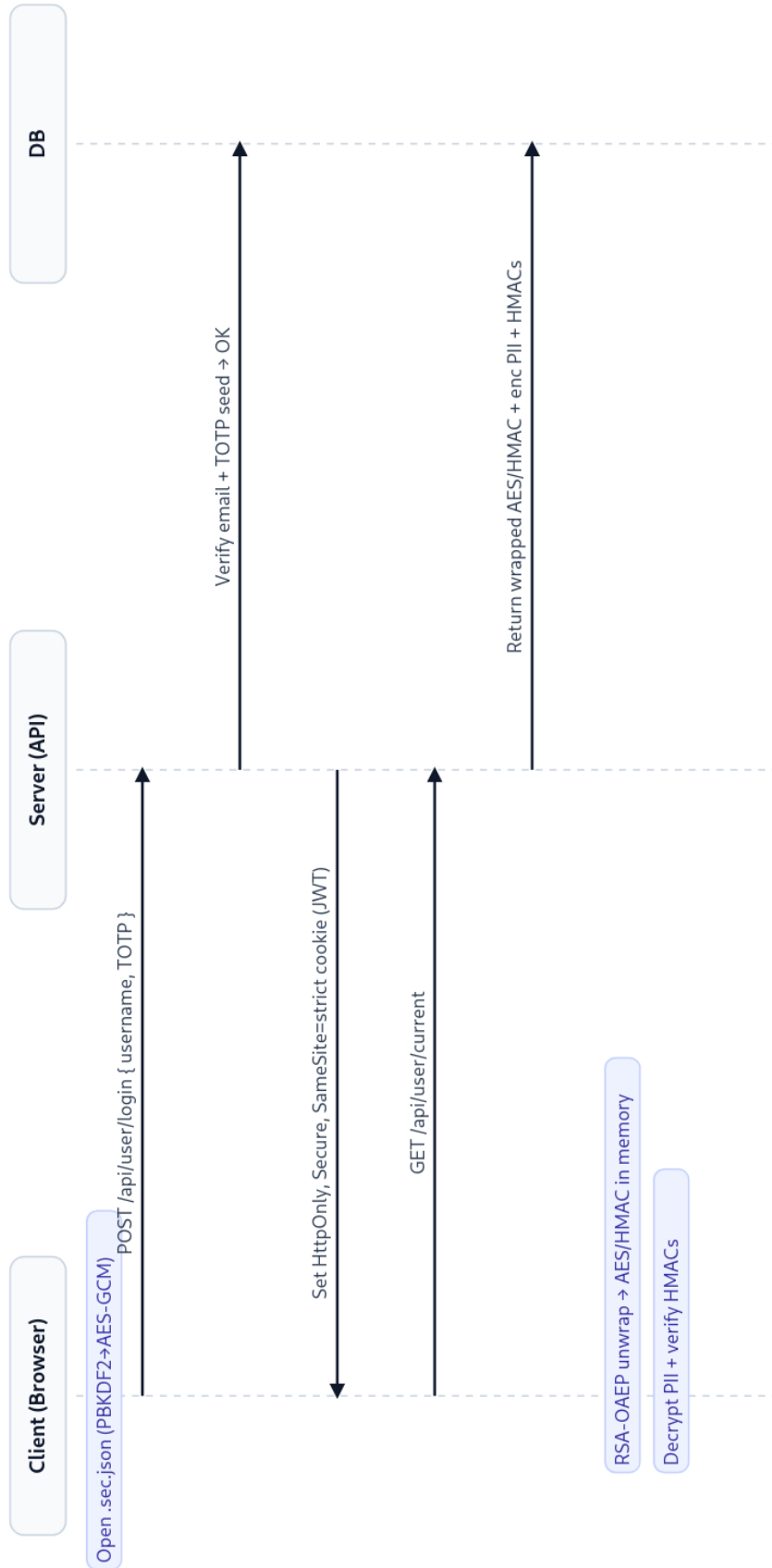
3.2 Certificate Issuance  
Client PKCS#10 → Step-CA leaf → return PEM to client bundle



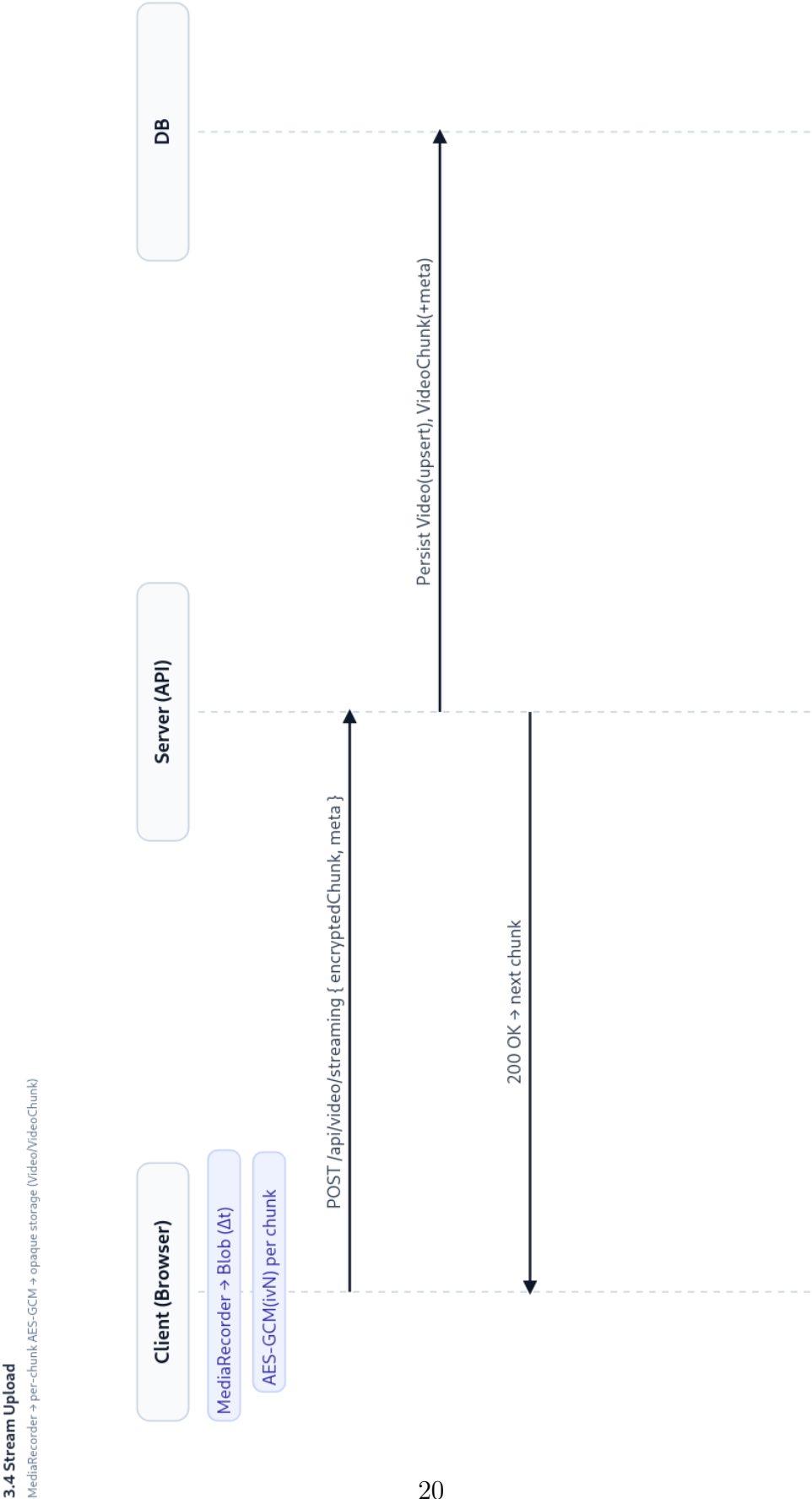
## C Annex: Pipeline – Login

### 3.3 Login

OTP (server) + key possession (client) → signed cookie → unwrap AES/HMAC locally



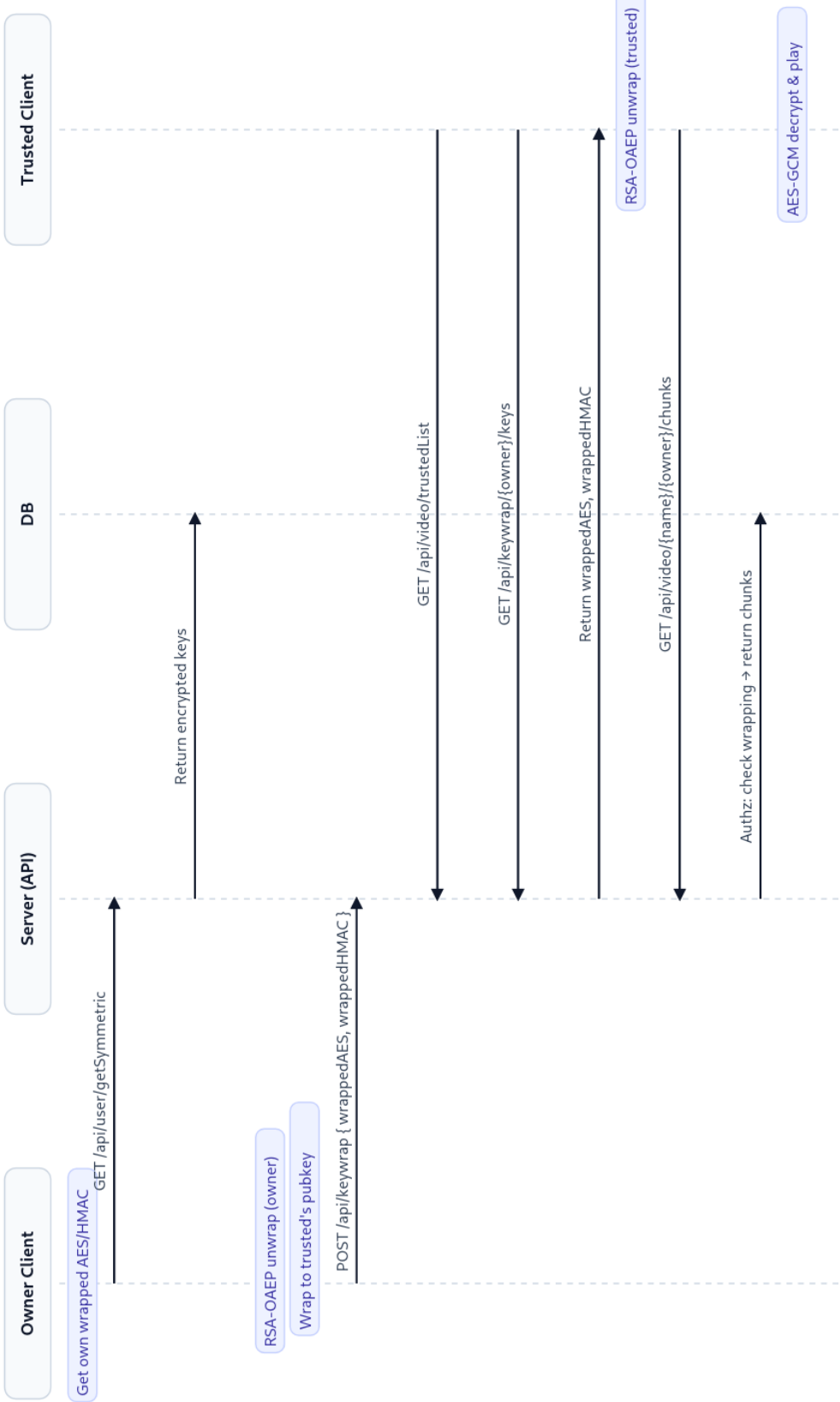
D    Annex: Pipeline – Stream



# E    Annex: Pipeline – Share

## 3.5 Stream Share

Owner wraps AES/HMAC to trusted cert → trusted unwraps → decrypts chunks





# F    Annex: Pipeline – Revocation

## 3.6 Key Revocation

Delete wrapping (owner→trusted) → deny keys/chunks

