HAUTE ÉCOLE BRUXELLES-BRABANT
FACULTY OF SCIENCES
MASTER OF CYBERSECURITY
2025-2026

Secure Software Development & Web Security
Computer Project

Medical Records System

Candidates

Peetroons Simon	**(519237)**
Andrianirina Mino	**(604350)**
Botton David	**(615056)**
Gerday Léandre	**(616822)**
Varga Ferenc	**(617441)**

Advisor

Professor Absil

Rue Royale 67, 1000 Bruxelles, Belgium

Contents

1 Introduction

The goal of this project is to implement a secure client/server system handling medical records, with an untrusted server and emphasis on security.

We designed our system as a Dockerized application using Django for the backend, React for the frontend, Nginx as a reverse proxy, and Step-CA for PKI management. Authentication relies on WebAuthn with PRF extensions for key derivation, supporting multi-device access via a primary-secondary hierarchy. All sensitive data, including medical records, is encrypted client-side using AES-GCM, with keys managed via ECDH for sharing between patients and appointed doctors. The server stores only ciphertexts and metadata, ensuring no access to plaintext even if compromised. This approach prioritizes efficiency and security: WebAuthn over passwords to resist phishing, client-side E2EE to maintain confidentiality.

2 System Characteristics and Architecture

The architecture is client/server, with clients driven by users (patients or doctors) and an untrusted server.

We containerized the system using Docker Compose, including services for the frontend (React with Tanstack, PNPM, Tailwind, Vite), backend (Django with custom WebAuthn), Nginx reverse proxy, and Step-CA for PKI. This ensures reproducibility and isolation, with Nginx handling TLS termination for client-server communication.

2.1 The Server

The server is not trusted. In particular, we do not know its set of public keys. Consequently, we provide a mechanism to "securely" transfer it and check its ownership via a local PKI chain: Root CA signs Intermediate CA, which issues leaf certificates for services (e.g., server.healthsecure.local). Fingerprints are computed and verified client-side.

No sensitive data is stored in plaintext; all medical records are ciphertexts. Metadata (e.g., request time, size, privileges, tree depth) is sent separately for anomaly detection, without depending on content.

2.2 Users and Medical Records

Users are patients or doctors. Doctors are trusted, with x509 certificates signed by the PKI for integrity and non-repudiation of attributes (e.g., organization).

Patients have first/last name, DOB, medical record (encrypted directory of dated files), and appointed doctors.

Doctors have first/last name and medical organization.

Medical records are client-encrypted directories; content/names/dates are sensitive and never exposed server-side.

3 Features and Security Implementation

In each protocol, data exchange and storage are secured with emphasis on confidentiality, integrity, and additional measures like anti-injection.

We implemented more than basic protections, reCAPTCHA, and logging with separation of duties, favoring efficiency (e.g., WebAuthn over MFA for simplicity and security).

3.1 User Registration, Authentication, and Revocation

Registration generates WebAuthn credentials client-side, with PRF extension for KEK derivation. Doctors include x509 certs signed by PKI for attribute verification.

Authentication uses WebAuthn assertions, validating sign counts to detect clones. Sessions are short-lived (20 min) and encrypted.

Multi-device: Primary device (first registered) approves secondaries via one-time codes (<10 min expiry). Only primary can change credentials or revoke devices.

Revocation flushes sessions and deletes credentials, logged with metadata.

Avoided: Traditional passwords (vulnerable to phishing/dictionary attacks). *Preferred:* WebAuthn for hardware-bound, phishing-resistant auth.

Addresses Q2 (hardened auth: zero-knowledge WebAuthn), Q5 (non-repudiation: signatures), Q14 (auth not broken: per OWASP).

3.2 Adding/Deleting a Doctor

Patients add doctors via ECDH-shared secrets for DEK sharing; requires patient approval if doctor-initiated. Deletion revokes access by removing shared keys. Integrity ensured via Ed25519 signatures over lists. *Avoided:* Server-mediated sharing (compromises confidentiality). *Preferred:* Client-side ECDH for E2EE sharing. Addresses Q4 (sequence integrity: signatures), Q9 (request forgery: CSRF tokens).

3.3 Viewing a Medical Record

Patients/doctors decrypt records client-side using KEK/DEK. Server provides ciphertexts only to authorized users (via session checks).

No one else accesses; RBAC enforced backend.

Addresses Q1 (confidentiality: E2EE), Q3 (integrity: Ed25519 signatures).

3.4 Uploading, Editing, and Deleting Files

Files encrypted client-side with random DEK, signed with Ed25519. DEK encrypted with KEK (patient) and ECDH (doctors).

Doctor actions require patient approval.

Deletion overwrites with random data to mitigate remanence.

Addresses Q8 (remanence: overwrite), Q7 (injections: Django validators).

3.5 Monitoring and Additional Security

Logs record activity

4 Process Flows Protocols

This section details the cryptographic protocols for the main actions using sequence diagrams. These diagrams illustrate the execution flow of critical operations within the application.

4.1 Doctor Login & Key Recovery

The doctor login process, detailed in the sequence diagram (Figure 1), enforces zero-trust principles by deriving encryption keys strictly on the client side without ever exposing them to the server.

1. **PRF Salt Normalization:** Upon receiving the authentication challenge and the PRF salt from the server, the client first normalizes the salt (converting from Base64URL to ArrayBuffer) to ensure compatibility with the WebCrypto API.
2. **KEK Derivation:** The authenticator (e.g., YubiKey) processes the normalized salt to produce the raw entropy, which is then used to derive the **Key Encryption Key (KEK)**.
3. **Transient Storage:** This KEK is stored exclusively in volatile memory (`window.__KEK__`) and is never written to disk, mitigating data remanence risks.
4. **E2EE Key Recovery:** The client retrieves the doctor's encrypted X25519 private key (used for E2EE) and decrypts it using the resident KEK.
5. **Anti-Mismatch Check:** Before finalizing the session, the client derives the public key from the decrypted private key and compares it against the public key stored on the server. This "Anti-Mismatch" verification ensures that the local identity matches the server-side identity, preventing potential key synchronization attacks.

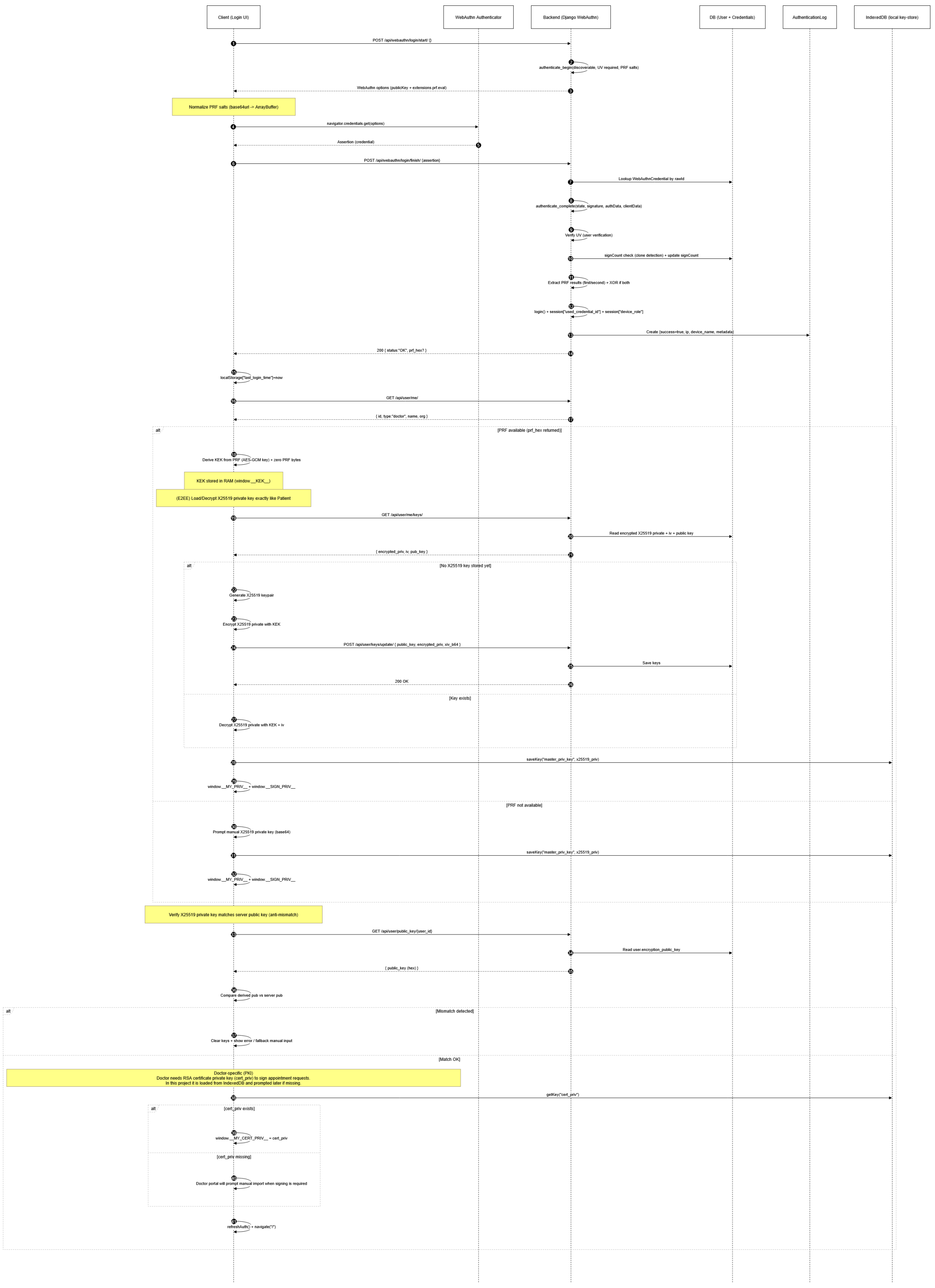
Client (Login UI)
WebAuthn Authenticator
Backend (Django WebAuthn)
DB (User + Credentials)
AuthenticationLog
IndexedDB (local key-store)
1 POST /api/webauthn/login/start/ {}
2 authenticate_begin(discoverable, UV required, PRF salts)
3 WebAuthn options (publicKey + extensions.prf.eval)
Normalize PRF salts (base64url -> ArrayBuffer)
4 navigator.credentials.get(options)
5 Assertion (credential)
6 POST /api/webauthn/login/finish/ {assertion}
7 Lookup WebAuthnCredential by rawId
8 authenticate_complete(state, signature, authData, clientData)
9 Verify UV (user verification)
10 signCount check (clone detection) + update signCount
11 Extract PRF results (first/second) + XOR if both
12 login() + session["used_credential_id"] + session["device_role"]
13 Create {success=true, ip, device_name, metadata}
14 200 { status:"OK", prf_hex? }
15 localStorage["last_login_time"]=now
16 GET /api/user/me/
17 { id, type:"doctor", name, org }
alt
[PRF available (prf_hex returned)]
18 Derive KEK from PRF (AES-GCM key) + zero PRF bytes
KEK stored in RAM (window.__KEK__)
(E2EE) Load/Decrypt X25519 private key exactly like Patient
19 GET /api/user/me/keys/
20 Read encrypted X25519 private + iv + public key
21 { encrypted_priv, iv, pub_key }
alt
[No X25519 key stored yet]
22 Generate X25519 keypair
23 Encrypt X25519 private with KEK
24 POST /api/user/keys/update/ { public_key, encrypted_priv, xiv_b64 }
25 Save keys
26 200 OK
[Key exists]
27 Decrypt X25519 private with KEK + iv
28 saveKey("master_priv_key", x25519_priv)
29 window.__MY_PRIV__ + window.__SIGN_PRIV__
[PRF not available]
30 Prompt manual X25519 private key (base64)
31 saveKey("master_priv_key", x25519_priv)
32 window.__MY_PRIV__ + window.__SIGN_PRIV__
Verify X25519 private key matches server public key (anti-mismatch)
33 GET /api/user/public_key/{user_id}
34 Read user.encryption_public_key
35 { public_key (hex) }
36 Compare derived pub vs server pub
alt
[Mismatch detected]
37 Clear keys + show error / fallback manual input
[Match OK]
Doctor-specific (PKI)
Doctor needs RSA certificate private key (cert_priv) to sign appointment requests.
In this project it is loaded from IndexedDB and prompted later if missing.
38 getKey("cert_priv")
alt
[cert_priv exists]
39 window.__MY_CERT_PRIV__ = cert_priv
[cert_priv missing]
40 Doctor portal will prompt manual import when signing is required
41 refreshAuth() + navigate("/")

Figure 1: Sequence Diagram: Doctor Login...

4.2 Patient Login & Key Recovery

The patient login flow (Figure 2) is designed to securely restore the user's cryptographic context on a new session without entrusting private keys to the server.

1. **Salt Normalization:** Upon authentication success, the client receives the PRF salt from the server and normalizes it (Base64URL to ArrayBuffer) to prepare for key derivation.
2. **KEK Derivation & Storage:** The authenticator computes the deterministic PRF output, which is converted into the **Key Encryption Key (KEK)**. This key is held strictly in volatile memory (`window.__KEK__`).
3. **Material Retrieval:** The server returns the patient's encrypted E2EE material (the X25519 private key and its initialization vector) along with the public key.
4. **Decryption:** The client uses the resident KEK to decrypt the X25519 private key, enabling the patient to decrypt their own medical records locally.

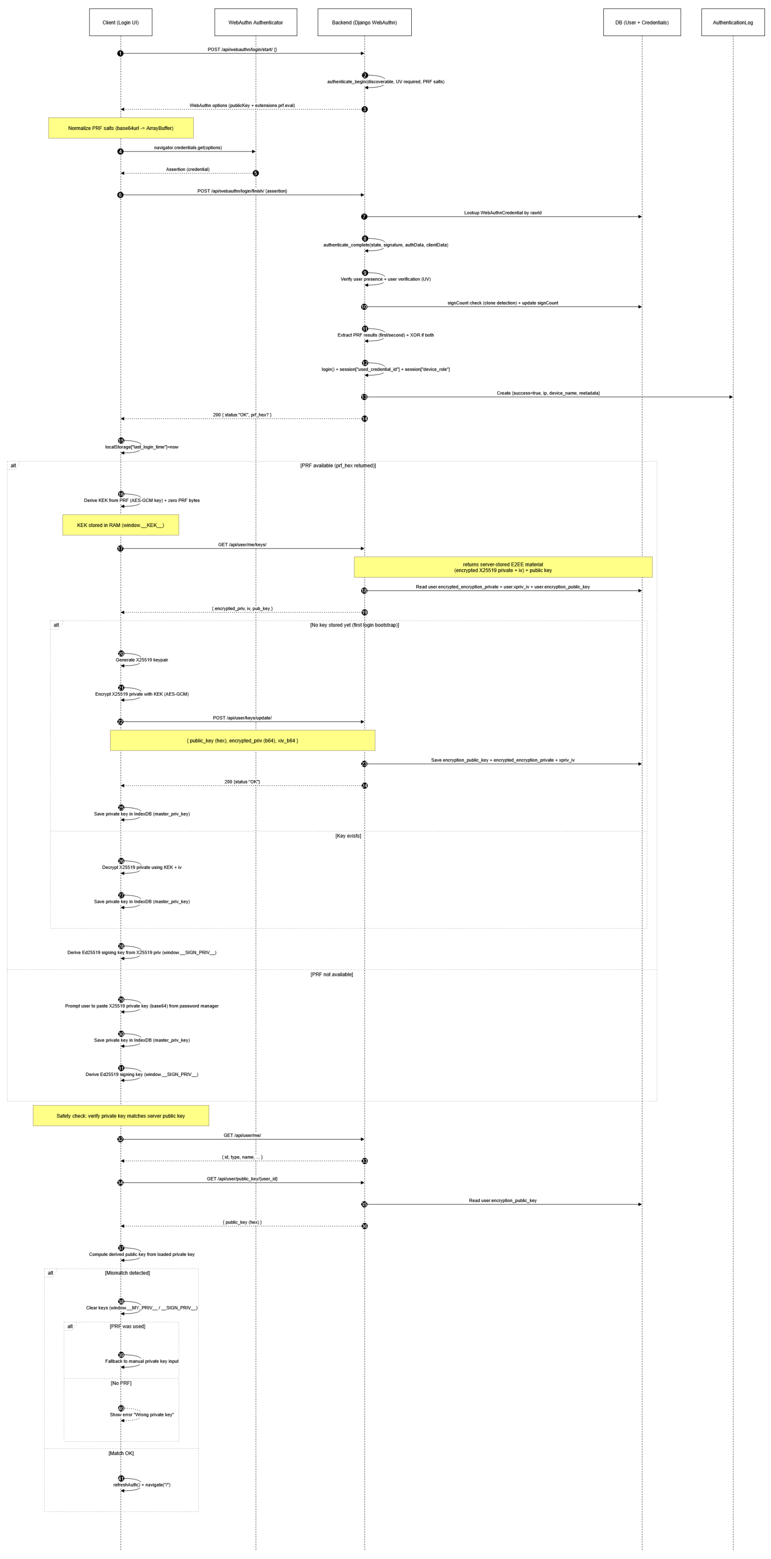
Client (Login UI)
WebAuthn Authenticator
Backend (Django WebAuthn)
DB (User + Credentials)
AuthenticationLog
1 POST /api/webauthn/login/start/ {}
2 authenticate_begin(discoverable, UV required, PRF salts)
3 WebAuthn options (publicKey + extensions.prf.eval)
Normalize PRF salts (base64url -> ArrayBuffer)
4 navigator.credentials.get(options)
5 Assertion (credential)
6 POST /api/webauthn/login/finish/ {assertion}
7 Lookup WebAuthnCredential by rawId
8 authenticate_complete(state, signature, authData, clientData)
9 Verify user presence + user verification (UV)
10 signCount check (clone detection) + update signCount
11 Extract PRF results (first/second) + XOR if both
12 login() + session["used_credential_id"] + session["device_role"]
13 Create {success=true, ip, device_name, metadata}
14 200 { status:"OK", prf_hex? }
15 localStorage["last_login_time"]=now
alt
[PRF available (prf_hex returned)]
16 Derive KEK from PRF (AES-GCM key) + zero PRF bytes
KEK stored in RAM (window.__KEK__)
17 GET /api/user/me/keys/
returns server-stored E2EE material
(encrypted X25519 private + iv) + public key
18 Read user.encrypted_encryption_private + user.xpriv_iv + user.encryption_public_key
19 { encrypted_priv, iv, pub_key }
alt
[No key stored yet (first login bootstrap)]
20 Generate X25519 keypair
21 Encrypt X25519 private with KEK (AES-GCM)
22 POST /api/user/keys/update/
{ public_key (hex), encrypted_priv (b64), xiv_b64 }
23 Save encryption_public_key + encrypted_encryption_private + xpriv_iv
24 200 {status:"OK"}
25 Save private key in IndexDB (master_priv_key)
[Key exists]
26 Decrypt X25519 private using KEK + iv
27 Save private key in IndexDB (master_priv_key)
28 Derive Ed25519 signing key from X25519 priv (window.__SIGN_PRIV__)
[PRF not available]
29 Prompt user to paste X25519 private key (base64) from password manager
30 Save private key in IndexDB (master_priv_key)
31 Derive Ed25519 signing key (window.__SIGN_PRIV__)
Safety check: verify private key matches server public key
32 GET /api/user/me/
33 { id, type, name, ... }
34 GET /api/user/public_key/{user_id}
35 Read user.encryption_public_key
36 { public_key (hex) }
37 Compute derived public key from loaded private key
alt
[Mismatch detected]
38 Clear keys (window.__MY_PRIV__ / __SIGN_PRIV__)
alt
[PRF was used]
39 Fallback to manual private key input
[No PRF]
40 Show error "Wrong private key"
[Match OK]
41 refreshAuth() + navigate("/")

Figure 2: Sequence Diagram: Patient Login and E2EE Material Recovery

4.3 Key Rotation & Revocation

The key rotation protocol (Figure 3) allows a user to revoke their cryptographic identity (e.g., in case of device theft) and migrate to a new keypair without losing data access.

1. **New Keys Generation:** The client generates a fresh set of X25519 (encryption) and Ed25519 (signing) keypairs locally.
2. **Record Re-encryption:** The client iterates through the user's medical records. It decrypts the Data Encryption Keys (DEKs) using the *old* private key and immediately re-encrypts them with the *new* public key. Note that the actual medical data (large blobs) is not decrypted, only the small keys protecting them.
3. **Access Re-wrapping:** If the user had shared access with doctors, the relevant access entries are updated ("re-wrapped") to ensure they point to the new cryptographic identity where necessary.
4. **Atomic Server Update:** The new Public Key and the batch of re-encrypted DEKs are sent to the server in a single, atomic transaction. The server replaces the old public key and updates the records simultaneously. If the transaction fails, the old keys remain valid, preventing data corruption.

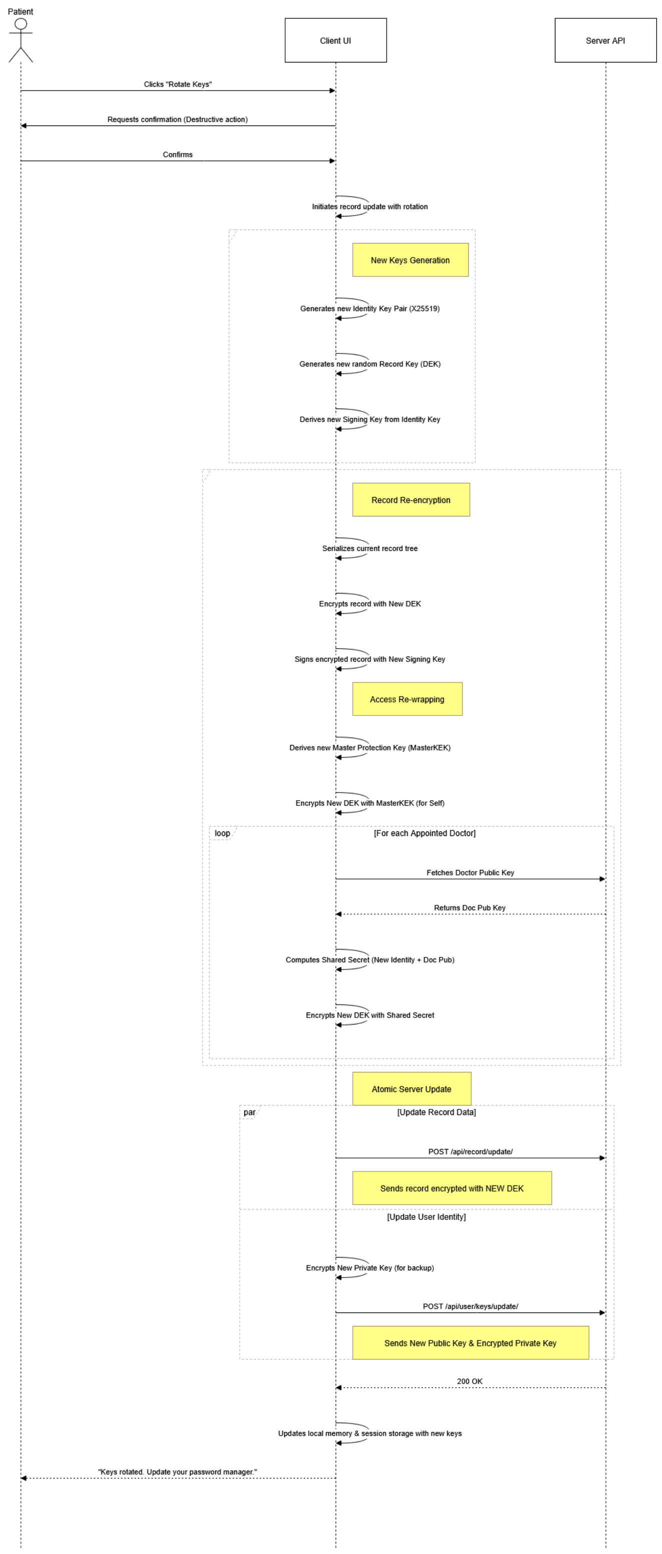
Patient
Client UI
Server API
Clicks "Rotate Keys"
Requests confirmation (Destructive action)
Confirms
Initiates record update with rotation
New Keys Generation
Generates new Identity Key Pair (X25519)
Generates new random Record Key (DEK)
Derives new Signing Key from Identity Key
Record Re-encryption
Serializes current record tree
Encrypts record with New DEK
Signs encrypted record with New Signing Key
Access Re-wrapping
Derives new Master Protection Key (MasterKEK)
Encrypts New DEK with MasterKEK (for Self)
loop
[For each Appointed Doctor]
Fetches Doctor Public Key
Returns Doc Pub Key
Computes Shared Secret (New Identity + Doc Pub)
Encrypts New DEK with Shared Secret
Atomic Server Update
par
[Update Record Data]
POST /api/record/update/
Sends record encrypted with NEW DEK
[Update User Identity]
Encrypts New Private Key (for backup)
POST /api/user/keys/update/
Sends New Public Key & Encrypted Private Key
200 OK
Updates local memory & session storage with new keys
"Keys rotated. Update your password manager."

Figure 3: Sequence Diagram: Atomic Key Rotation and Data Re-encryption

4.4 Doctor-Initiated Appointment & Approval

This protocol (Figure 4) addresses the scenario where a doctor proposes a medical link. To respect the patient's data sovereignty, this process is split into two asynchronous phases: request creation and patient approval.

Phase 1: Request Creation (Doctor)

1. **Request Generation:** The doctor initiates a "Link Request" targeting a specific patient. This request includes the doctor's identity and is signed with their Ed25519 private key to prevent spoofing.

2. **Request Encryption:** The request payload is encrypted with the patient's public key (retrieved from the server) before upload. This ensures that the nature of the request and the medical relationship remain confidential; the server sees only an opaque blob pending for the patient.

Phase 2: Approval (Patient)

1. **Fetch & Decrypt:** Upon logging in, the patient retrieves pending requests and decrypts them using their private key (recovered via WebAuthn PRF).

2. **Verification:** The client application verifies the doctor's signature against their PKI certificate to ensure the request is legitimate.

3. **Consent & Wrapping:** If the patient accepts, the client performs the ECDH key exchange (computing the shared secret) and encrypts the Record DEK for the doctor.

4. **Finalization:** The wrapped DEK is uploaded, effectively converting the "pending request" into an active "authorized link."

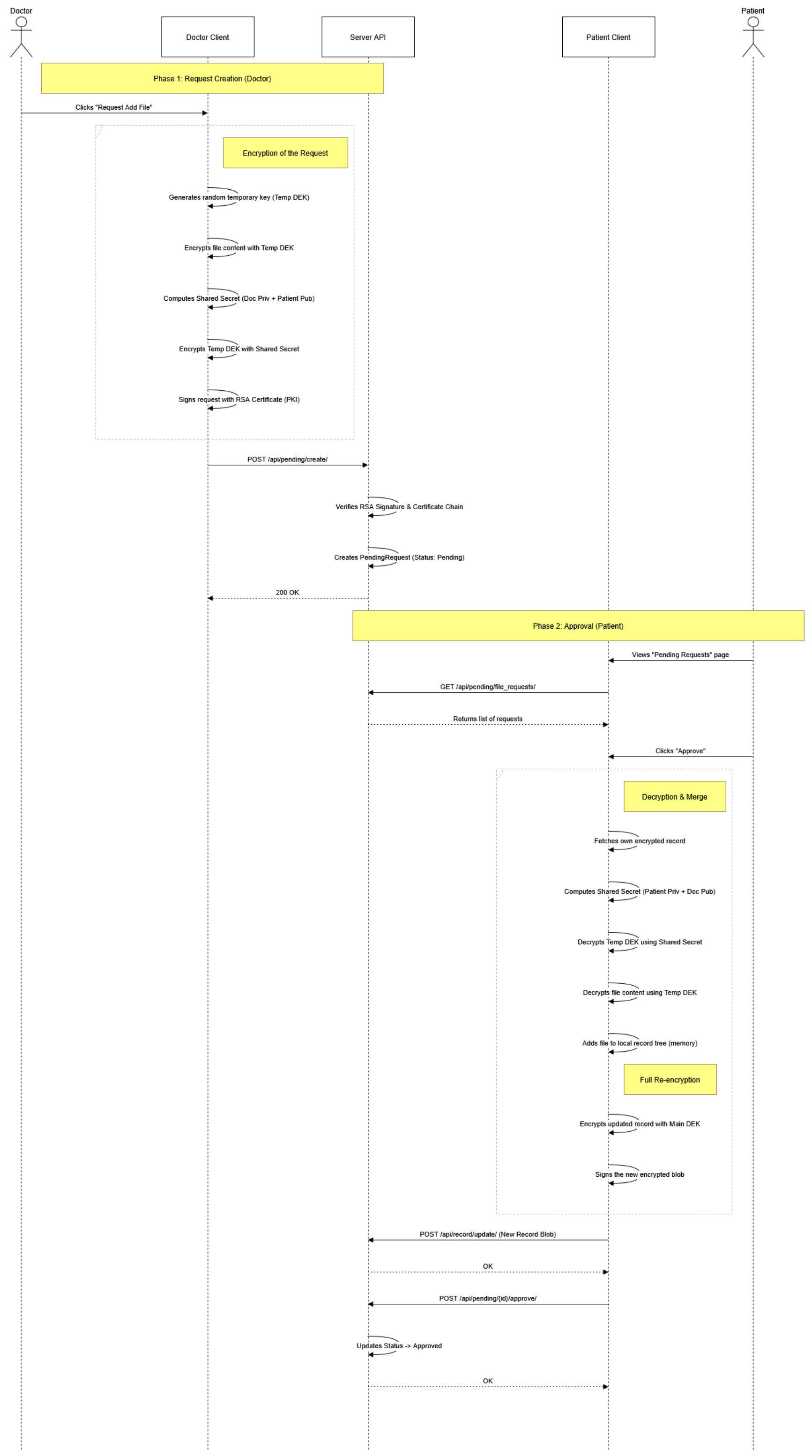
Doctor
Doctor Client
Server API
Patient Client
Patient
Phase 1: Request Creation (Doctor)
Clicks "Request Add File"
Encryption of the Request
Generates random temporary key (Temp DEK)
Encrypts file content with Temp DEK
Computes Shared Secret (Doc Priv + Patient Pub)
Encrypts Temp DEK with Shared Secret
Signs request with RSA Certificate (PKI)
POST /api/pending/create/
Verifies RSA Signature & Certificate Chain
Creates PendingRequest (Status: Pending)
200 OK
Phase 2: Approval (Patient)
Views "Pending Requests" page
GET /api/pending/file_requests/
Returns list of requests
Clicks "Approve"
Decryption & Merge
Fetches own encrypted record
Computes Shared Secret (Patient Priv + Doc Pub)
Decrypts Temp DEK using Shared Secret
Decrypts file content using Temp DEK
Adds file to local record tree (memory)
Full Re-encryption
Encrypts updated record with Main DEK
Signs the new encrypted blob
POST /api/record/update/ (New Record Blob)
OK
POST /api/pending/{id}/approve/
Updates Status -> Approved
OK

Figure 4: Sequence Diagram: Doctor-Initiated Request and Patient Approval

4.5 Patient Record Creation & Upload

This sequence (Figure 5) details how a patient securely adds a new file (video or document) to their medical record. The process ensures that the server acts purely as a storage provider for encrypted blobs.

1. **DEK Generation:** The client generates a fresh, random **Data Encryption Key (DEK)** (AES-256) specifically for this file.
2. **Encryption (E2EE):** The file content is encrypted locally using this DEK (AES-GCM), ensuring confidentiality.
3. **Signing:** To guarantee integrity and authenticity, the client signs the resulting ciphertext using the patient's **Ed25519 private key**.
4. **Key Wrapping:** The DEK itself is encrypted with the patient's **X25519 public key**. This allows the patient to recover the key later (during login/decryption) without storing the plaintext key on the server.
5. **Upload:** The client sends a payload containing the *Encrypted Content*, the *Digital Signature*, and the *Wrapped DEK* to the server. The server stores these artifacts and confirms persistence (200 OK).

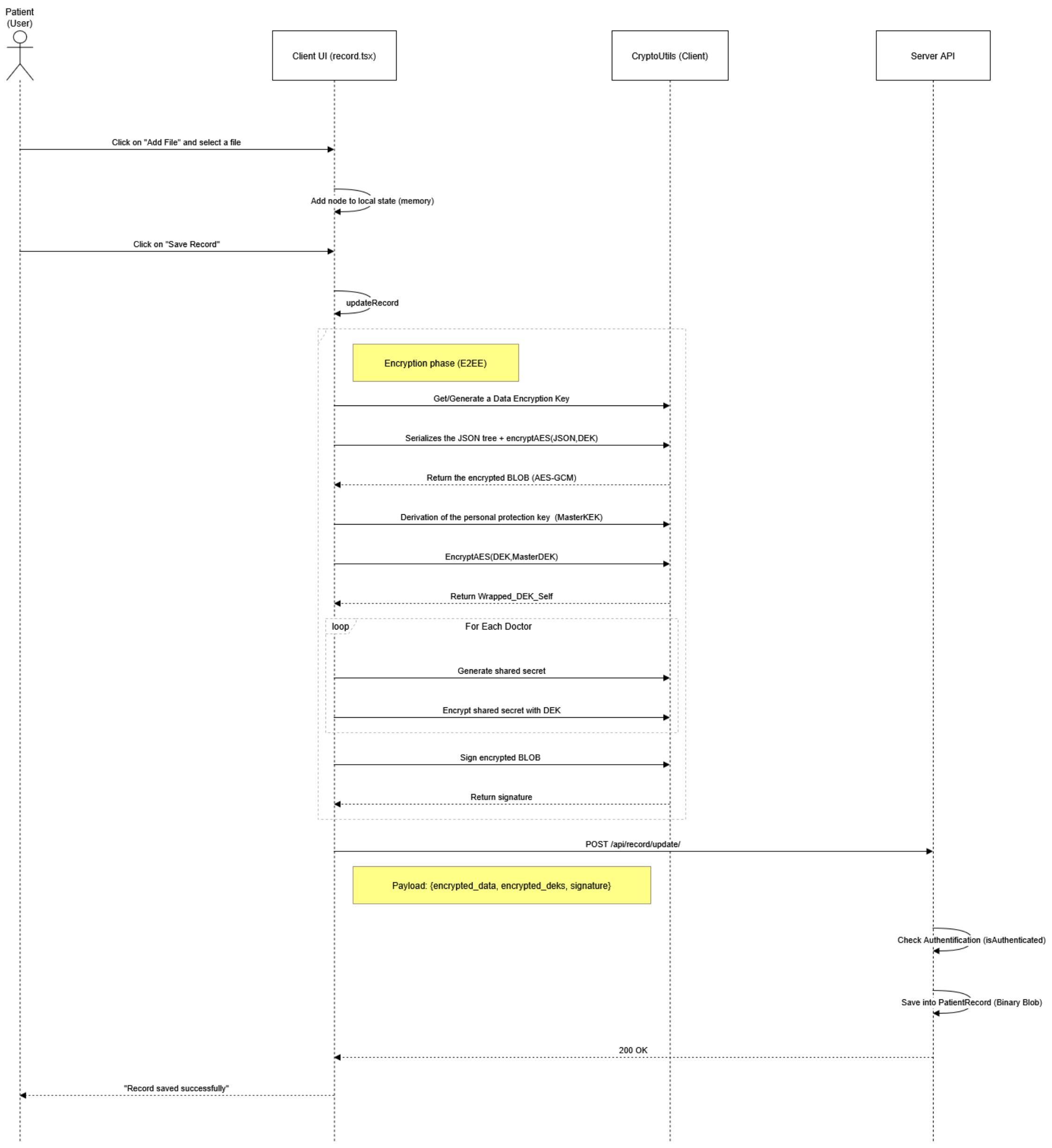
Patient (User)
Client UI (record.tsx)
CryptoUtils (Client)
Server API
Click on "Add File" and select a file
Add node to local state (memory)
Click on "Save Record"
updateRecord
Encryption phase (E2EE)
Get/Generate a Data Encryption Key
Serializes the JSON tree + encryptAES(JSON,DEK)
Return the encrypted BLOB (AES-GCM)
Derivation of the personal protection key (MasterKEK)
EncryptAES(DEK,MasterDEK)
Return Wrapped_DEK_Self
loop
For Each Doctor
Generate shared secret
Encrypt shared secret with DEK
Sign encrypted BLOB
Return signature
POST /api/record/update/
Payload: {encrypted_data, encrypted_deks, signature}
Check Authentification (isAuthenticated)
Save into PatientRecord (Binary Blob)
200 OK
"Record saved successfully"

Figure 5: Sequence Diagram: Secure Record Creation and Upload by Patient

5 Threat Modeling (STRIDE)

We applied the STRIDE methodology to guide our security choices:

Threat	Mitigation Strategy
Spoofing	WebAuthn (FIDO2) prevents phishing and impersonation. PKI/X.509 certificates authenticate the Server and Doctors.
Tampering	All records and lists are signed using Ed25519. Any modification by the server is detected by the client upon retrieval.
Repudiation	Doctor actions are signed (digital signatures), ensuring they cannot deny authorizing a record creation or modification.
Information Disclosure	End-to-End Encryption (AES-GCM). The server sees only ciphertext.
Denial of Service	Rate limiting via Nginx.
Elevation of Privilege	Role-Based Access Control (RBAC) enforced by backend logic + Cryptographic Access Control (you cannot decrypt if you don't have the key).

Table 1: STRIDE Analysis

6 Security Checklist Analysis

The following analysis addresses the security questions posed in the *Security questions check-list*.

6.1 Confidentiality and Data Protection

1. Do I properly ensure confidentiality?

Status: Yes.

Confidentiality is enforced via End-to-End Encryption (E2EE) and strict Transport Layer Security.

- **Transmission:** All data between client and server is encrypted using TLS v1.2/1.3. The Nginx configuration enforces HSTS (HTTP Strict Transport Security) to prevent downgrade attacks.

```
        ...
        ssl_protocols TLSv1.2 TLSv1.3;
        ...
        add_header Strict-Transport-Security "max-age
           =31536000; includeSubDomains" always;
        ...
    }
```

Listing 1: Nginx configuration (nginx.conf)

- **Storage:** The server operates on a "Zero-Knowledge" principle. Medical records are encrypted on the client side using **AES-256-GCM** before upload. The database only stores the encrypted blobs (`encrypted_data`) and encrypted keys.

- **Access:** Administrators cannot view sensitive data because they do not possess the user's private key required to unwrap the Data Encryption Keys (DEKs).

```
export function encryptAES(data: Uint8Array, key: Uint8Array) {
  const iv = randomBytes(12);
  const aes = new AES(key);
  const cipher = new GCM(aes); // Authenticated Encryption
  const sealed = cipher.seal(iv, data);
  // Returns ciphertext, iv, and tag
}
```

Listing 2: Client-side Encryption (CryptoUtils.ts)

2. Did I harden my authentication scheme?

Status: Yes.

The project uses a passwordless authentication scheme based on FIDO2/WebAuthn.

- **MFA by Design:** WebAuthn provides multi-factor authentication (Possession of device + Biometrics/PIN) by default.
- **Zero-Knowledge:** The server never receives a password or a private key; it only validates a cryptographic signature.
- **Anti-Bot:** Google reCAPTCHA v3 is implemented on the registration endpoint to prevent automated spam.

```
try:
    recaptcha_token = clean_str(data.get("recaptcha_token", ""), "
       recaptcha_token", max_len=4096)
except ValueError as e:
    return bad(str(e), status=400)

verify_url = "https://www.google.com/recaptcha/api/siteverify"
verify_data = {
    "secret": settings.RECAPTCHA_SECRET_KEY,
    "response": recaptcha_token,
    "remoteip": request.META.get("REMOTE_ADDR"),
}

try:
    verify_resp = requests.post(verify_url, data=verify_data,
       timeout=6)
    verify_json = verify_resp.json()
except Exception:
    return bad("reCAPTCHA verification error", status=400)

if (not verify_json.get("success")) or (verify_json.get("score",
   0) < settings.RECAPTCHA_SCORE_THRESHOLD):
    return bad("reCAPTCHA verification failed - are you for real?"
       , status=400)
```

Listing 3: reCAPTCHA use (server/webauthn/views.py)

```
# User verification required ensures local auth (biometrics/pin)
options, state = server.register_begin(
    user={ ... },
    credentials=[],
    user_verification="required",
)
```

Listing 4: WebAuthn Registration (server/webauthn/views.py)

6.2 Integrity and Non-Repudiation

3. Do I properly ensure integrity of stored data?

Status: Yes.

We utilize **Ed25519** digital signatures. Before uploading a record, the client signs the encrypted blob using a signing key derived from their identity. When fetching data, the signature is verified. If the data on the server is tampered with (e.g., by a malicious admin), the signature verification will fail on the client side.

```
    // Signing the encrypted blob (concatenated) before upload
    const sig = signEd25519(concatenated, edPriv)
    ...
    const updatePayload = {
      encrypted_data: bytesToBase64(concatenated),
      encrypted_deks: deks,
      signature: bytesToBase64(sig),
      metadata,
    };
```

Listing 5: Payload upload with signature and encrypted blob (client/src/routes/record.tsx)

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

Status: Yes.

The medical record is structured as a tree but serialized into a single JSON object before encryption. This means the entire structure (file sequence, folder hierarchy) is encrypted and signed as one monolithic block. It is impossible to insert, delete, or reorder an item in the sequence without invalidating the signature of the whole record.

```
    const raw = new TextEncoder().encode(JSON.stringify(record))
    const encrypted = await encryptAES(raw, dek)
    const concatenated = new Uint8Array([...encrypted.iv, ...
       encrypted.ciphertext, ...encrypted.tag])
    const edPriv = deriveEd25519FromX25519(priv).privateKey
    const sig = signEd25519(concatenated, edPriv)
```

Listing 6: Encrypted serialize JSON as a block (client/src/routes/record.tsx)

5. Do I properly ensure non-repudiation?

Status: Yes.

- **Patients:** All record updates are signed with their private Ed25519 key.
- **Doctors:** Doctors are issued X.509 Certificates signed by an internal Certificate Authority (Step-CA). Critical actions (like requesting file changes) must be signed with the private key associated with this certificate, preventing doctors from denying their actions later.

```
const certPrivKey = await crypto.subtle.importKey(
  "pkcs8",
  window.__MY_CERT_PRIV__,
  { name: "RSASSA-PKCS1-v1_5", hash: "SHA-256" },
  false,
  ["sign"]
);

const signature = await crypto.subtle.sign(
  "RSASSA-PKCS1-v1_5",
  certPrivKey,
  requestMsg
);
```

Listing 7: RSA signature of a request for non-repudiation (client/src/routes/record.tsx)

6.3 Vulnerability Management

6. Do my security features rely on secrecy?

Status: No.

The system follows Kerckhoffs's principle. We use standard, open-source cryptographic libraries (`@stablelib`, `cryptography` in Python) and standard algorithms (AES-GCM, SHA-256, Curve25519). Security relies solely on the secrecy of the keys, not the code.

7. Am I vulnerable to injection?

Status: No.

- **SQL Injection:** The backend uses Django's ORM, which parameterizes queries to prevent SQL injection.
- **XSS:** The frontend uses React (which escapes output by default). Nginx is configured with a strict `Content-Security-Policy` (CSP).
- **Validation:** A strict input validation layer (`inputValidation.ts`) sanitizes all incoming data.

8. Am I vulnerable to data remanence attacks?

Status: Partially Mitigated.

The system employs a hybrid storage strategy balancing security and usability:

- **Identity Keys (Patients & Doctors):** The master identity key (X25519) used for encryption is kept in **SessionStorage** and volatile memory (RAM). It is never written to persistent storage like LocalStorage or Cookies. Closing the tab wipes this key.
- **Signing Keys (Doctors only):** To facilitate daily workflows, the doctor's RSA private key used for signing requests is stored in **IndexedDB** (persistent client-side storage). While this improves UX, it introduces a data remanence risk on shared devices, requiring explicit logout to clear.

```
useEffect(() => {
  // 1. Identity Key: Volatile (SessionStorage)
  const stored = sessionStorage.getItem('x25519_priv_b64');
  if (stored) {
    window.__MY_PRIV__ = base64ToBytes(stored);
  }

  // 2. Doctor Signing Key: Persistent (IndexedDB)
  async function loadCertKey() {
    if (user?.type === "doctor") {
      const storedCert = await getKey('cert_priv');
      if (storedCert) {
        window.__MY_CERT_PRIV__ = storedCert as Uint8Array;
      }
    }
  }
  loadCertKey();
}, [user]);
```

Listing 8: Hybrid Key Storage Strategy (client/src/routes/record.tsx)

9. Am I vulnerable to fraudulent request forgery?

Status: Mitigated.

The application enforces CSRF protection. The frontend must extract the `csrftoken` cookie and send it in the `X-CSRFToken` header for every state-changing request.

```
  const r = await fetch('/api/record/update/', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      'X-CSRFToken': getCookie('csrftoken') || '',
      "X-Metadata": updateMetadataHeader // Attach as header
    },
    credentials: 'include',
    body: JSON.stringify(updatePayload),
  })
```

Listing 9: Manual inclusion of the CSRF token in fetch headers (client/src/routes/record.tsx)

6.4 Monitoring and Configuration

10. Am I monitoring enough user activity?

Status: Yes.
A dedicated **Logger Service** records sensitive actions (login, key access).

11. Am I using components with known vulnerabilities?

Status: Managed.
Dependencies are managed via `npm` and `pip`, allowing for regular audits. Docker containers are used to isolate the environment and allow for easy system updates.

12. Is my system updated?

Status: Yes.
The architecture relies on Docker. Updating the system (OS patches, library updates) is done by rebuilding the container images, ensuring a clean and up-to-date state.

13. Is my access control broken?

Status: No.
Access control is enforced at two levels:

1. **Logical:** Django views check user permissions (e.g., `@login_required`).
2. **Cryptographic:** Even if logical checks fail, a user cannot read data without the correct decryption key (DEK), which is physically encrypted for specific users only.

14. Is my authentication broken?

Status: No.
By eliminating passwords, we eliminate the most common authentication vulnerabilities (Weak passwords, Credential stuffing, Phishing).

15. Are my general security features misconfigured?

Status: No.
Server configuration is hardened:

- **Django:** Debug mode is controlled via env vars, secrets are not hardcoded.
- **Nginx:** Security headers are active (`X-Frame-Options`, `X-Content-Type-Options`, `CSP`).