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 **1**

2 System Characteristics and Architecture **1**
2.1 The Server 1
2.2 Users and Medical Records 1

3 Quickstart Guide **1**
3.1 Booting the Stack 2
3.2 Administration Interface 2
3.3 User Registration 3
3.4 Login 4
3.5 Core Workflows (Demo Path) 5
3.5.1 Initialize a Patient Record (DEK bootstrap) 5
3.5.2 Search and Appoint a Doctor (Patient-initiated) 5
3.5.3 Doctor Views an Appointed Patient Record 6
3.5.4 Doctor-Initiated Appointment Request (Pending Approval) 7
3.5.5 Doctor Requests a File Change and Patient Approves (Pending Requests) 8
3.5.6 Remove a Doctor (Revocation) 9
3.6 Verifying Encrypted Storage 10
3.7 Multi device with chrome dev tools 10
3.8 Logs and Monitoring 11

4 Features and Security Implementation **11**
4.1 User Registration, Authentication, and Revocation 11
4.2 Adding/Deleting a Doctor 12
4.3 Viewing a Medical Record 12
4.4 Uploading, Editing, and Deleting Files 12
4.5 Monitoring and Additional Security 12

5 Process Flows & Protocols **12**
5.1 Doctor Registration & Credential Provisioning 12
5.2 Patient Registration & Initial Cryptographic Setup 13
5.3 Doctor Login & Key Recovery 15
5.4 Doctor Logout 18
5.5 Patient Login & Key Recovery 18
5.6 Patient Logout 21
5.7 Key Rotation & Revocation 21
5.8 Multi-Device Enrollment & Credential Management 24
5.9 Doctor Patient Search (Authenticated Query & Auditing) 26
5.10 Patient Doctor Search (Role-Gated Discovery) 27
5.11 Doctor-Initiated Appointment & Approval 29
5.12 Patient Record Creation & Upload 31

6 Threat Modeling (STRIDE) **33**

7 Security Checklist Analysis **33**
7.1 Confidentiality and Data Protection 33

7.2 Integrity and Non-Repudiation 35
7.3 Vulnerability Management 36
7.4 Monitoring and Configuration 38

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 Quickstart Guide

This section provides a minimal procedure to deploy the system and verify its correct and secure operation. The objective is not usability, but to demonstrate reproducibility, correct cryptographic setup, and confidentiality of stored data.

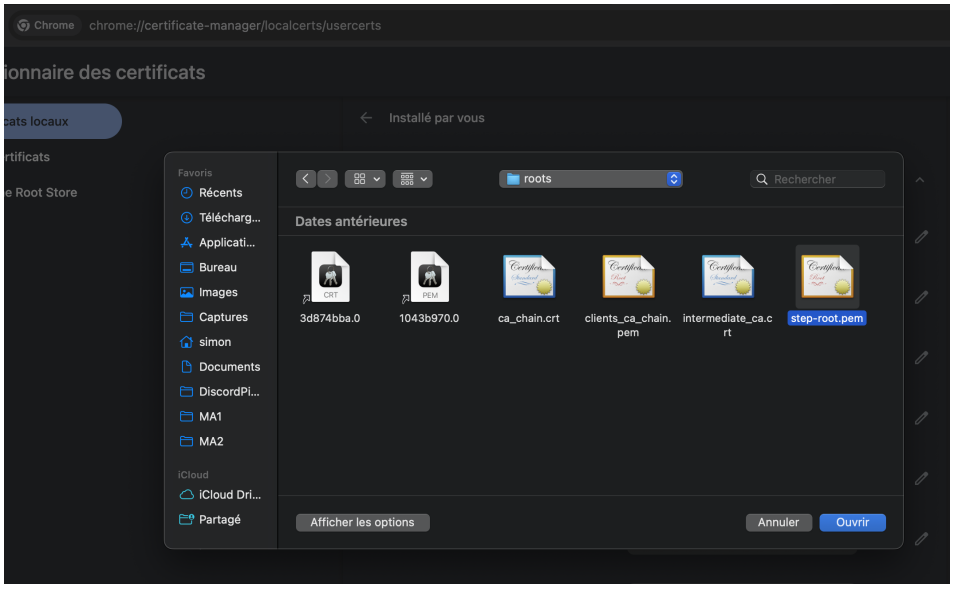
3.1 Booting the Stack

1. Ensure all prerequisites listed in the `README.md` are installed (Docker, Docker Compose, Buildx, Make).

2. From the project root, build and start the full stack:

```
make
```

3. Import the generated Root CA (`step-root.pem`) into the browser trust store.



4. Access the application at:

```
https://healthsecure.local:3443
```

3.2 Administration Interface

For demonstration and inspection purposes, a Django administration interface is available.

1. Create an administrator account:

```
docker compose exec server uv createsuperuser
```

2. Access the admin interface:

```
https://healthsecure.local:3443/admin
```

This interface allows inspection of users, records, pending requests, and certificates, without exposing any plaintext medical data.

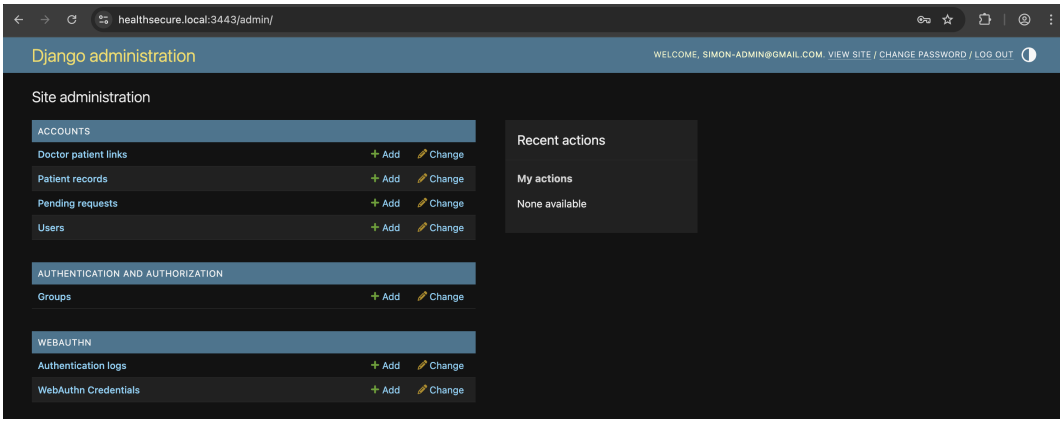


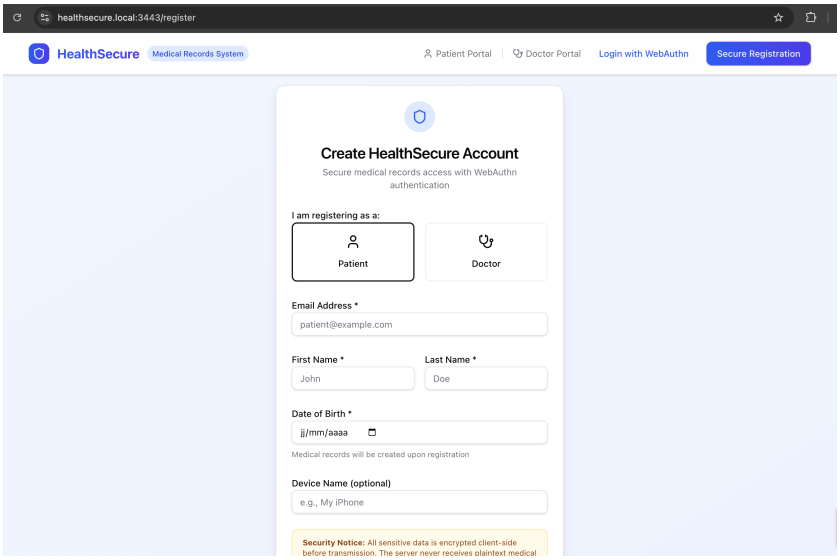
Figure 1: Django administration interface showing system objects without access to plaintext medical data

3.3 User Registration

Patient or Doctor

1. Navigate to:

```
https://healthsecure.local:3443/register
```



2. Fill in the required identity fields.

3. Complete WebAuthn registration using a compatible authenticator.

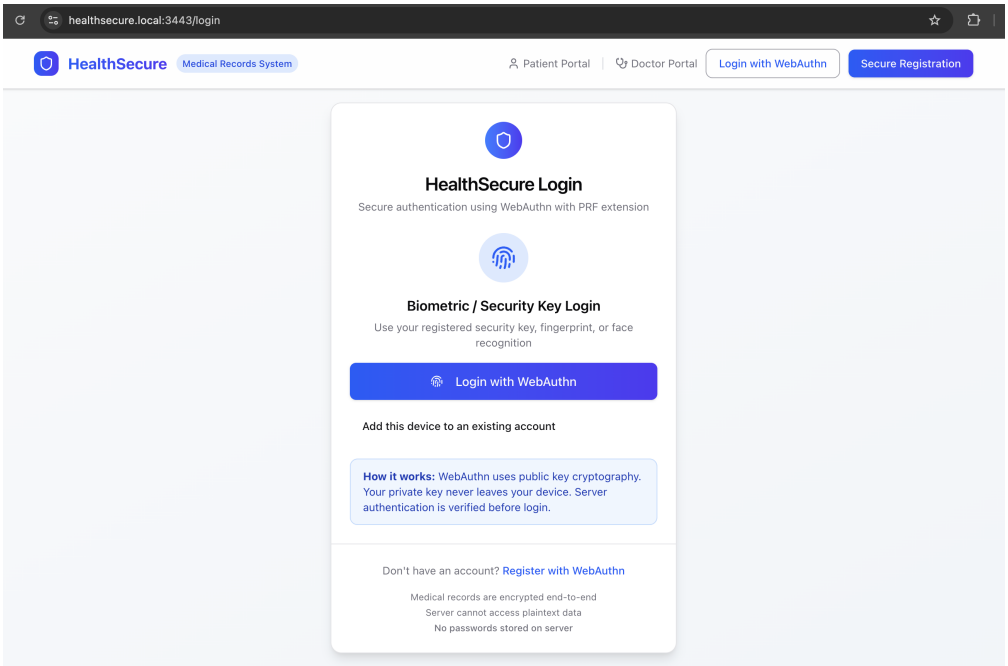
If the authenticator supports the WebAuthn PRF extension (e.g., Windows Hello, Touch ID), it is used for deterministic key derivation. Otherwise, a non-PRF fallback path is used, relying on local key storage. In both cases, private keys never transit through the server.

3.4 Login

1. Navigate to:

```
https://healthsecure.local:3443/login
```

2. Authenticate using WebAuthn.
3. Cryptographic material is restored client-side and held in volatile memory only.



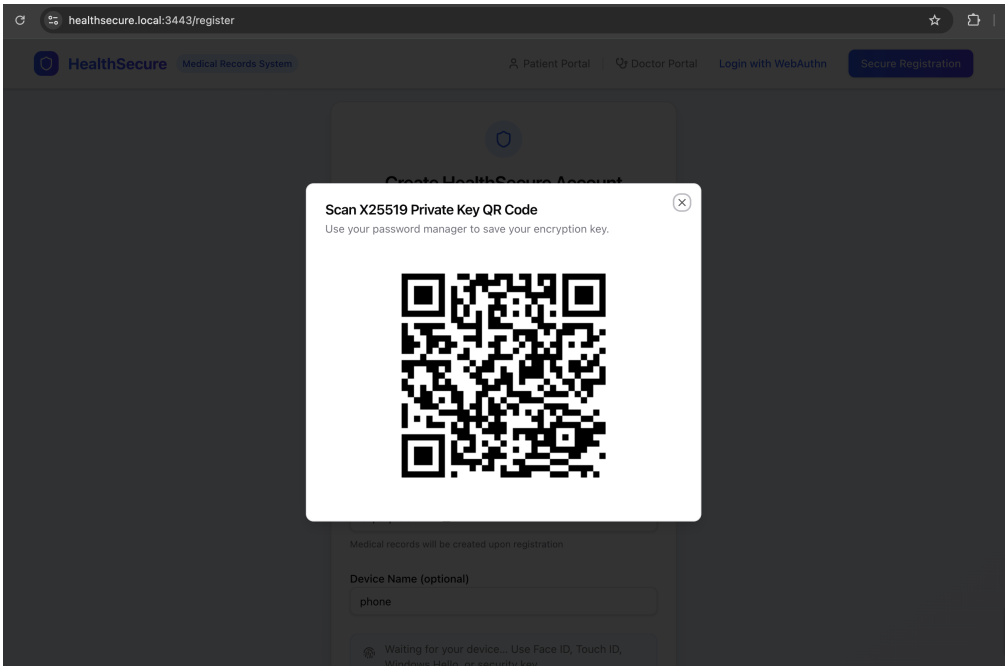


Figure 2

3.5 Core Workflows (Demo Path)

This subsection demonstrates the main security-critical actions implemented in the project: record initialization, doctor appointment, controlled access, and patient approval for doctor-initiated actions.

3.5.1 Initialize a Patient Record (DEK bootstrap)

A patient record is encrypted client-side. A Data Encryption Key (DEK) must be initialized once.

1. Login as a **patient**
2. Open the medical record page:

```
https://healthsecure.local:3443/record
```

3.5.2 Search and Appoint a Doctor (Patient-initiated)

Patients control which doctors can access their medical record.

1. Login as a **patient**
2. Open the medical record interface

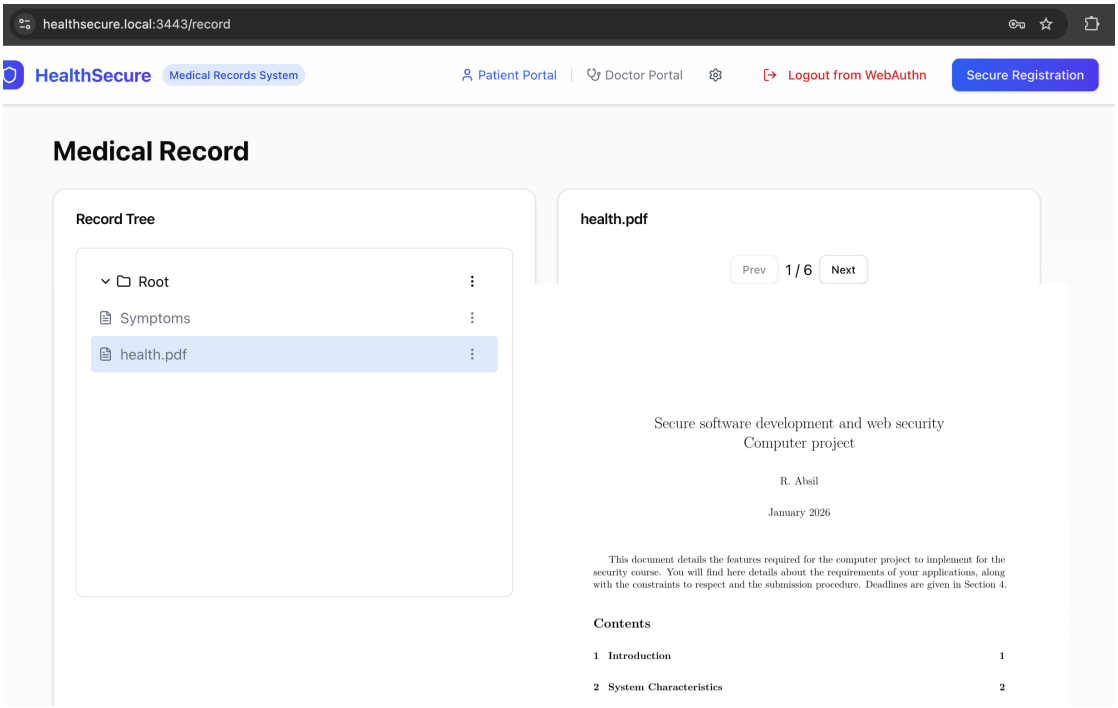


Figure 3: With PDF viewer in the record

3. Search for a doctor and appoint them
4. The client updates the patient's encrypted DEK map so the appointed doctor can decrypt

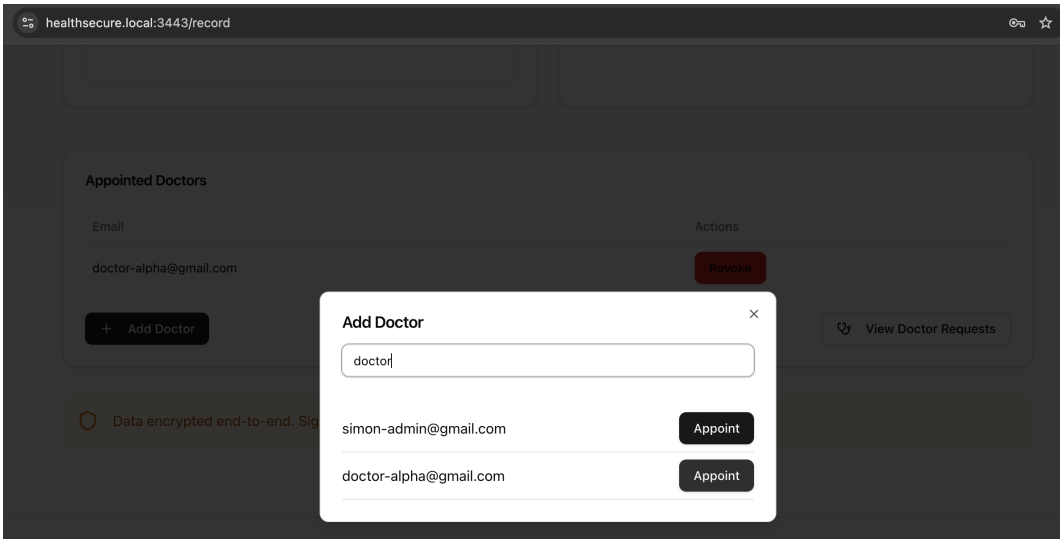


Figure 4: Patient interface for searching and appointing a doctor

3.5.3 Doctor Views an Appointed Patient Record

A doctor can only access patients that have explicitly appointed them.

1. Login as a **doctor**
2. View the list of appointed patients
3. Select a patient and open their encrypted medical record

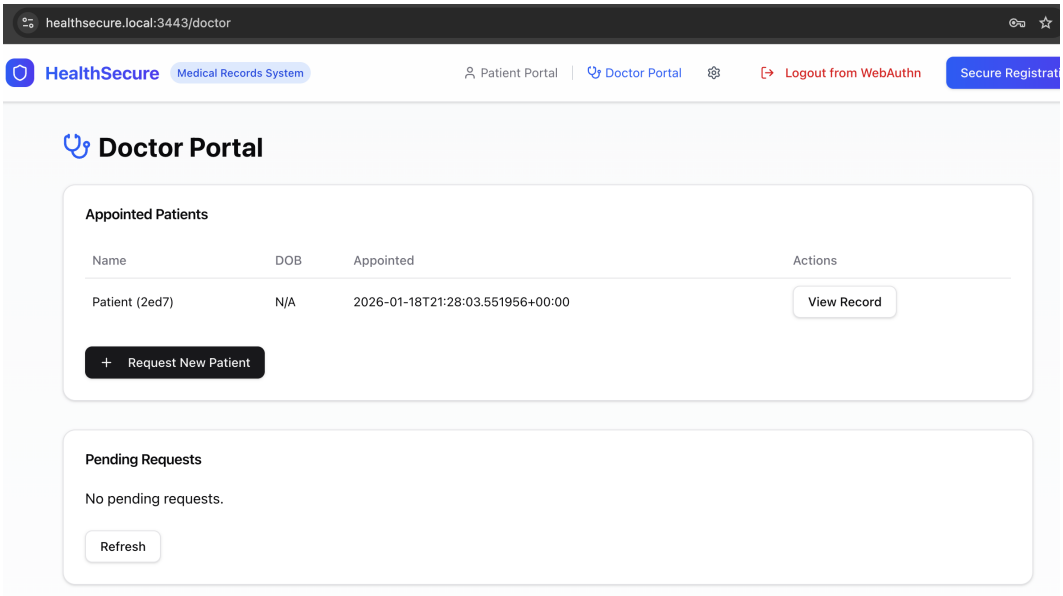
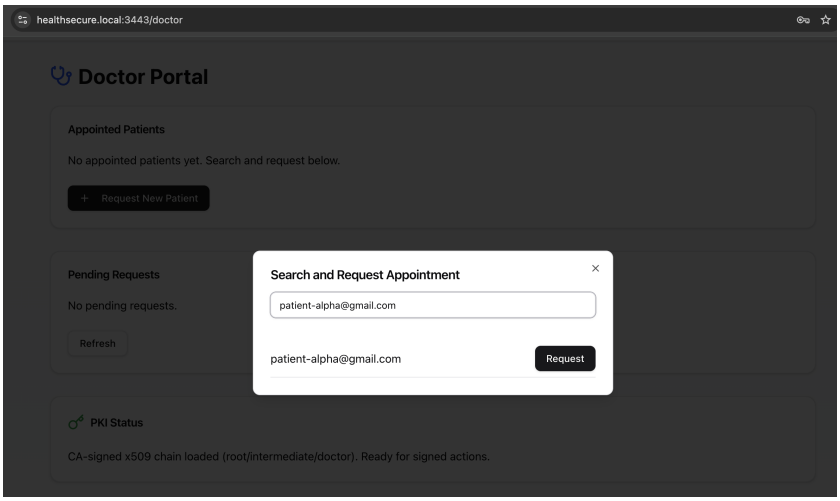


Figure 5

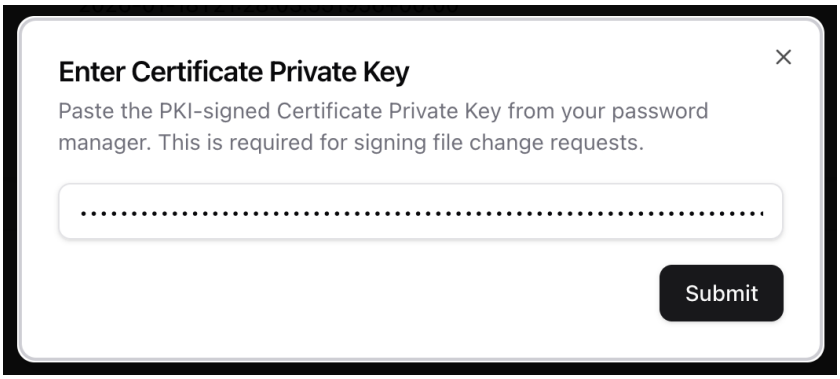
3.5.4 Doctor-Initiated Appointment Request (Pending Approval)

If a doctor requests access first, the patient must explicitly approve.

1. Login as a **doctor**
2. Search a patient and create an appointment request



3. Validate the request as **doctor**



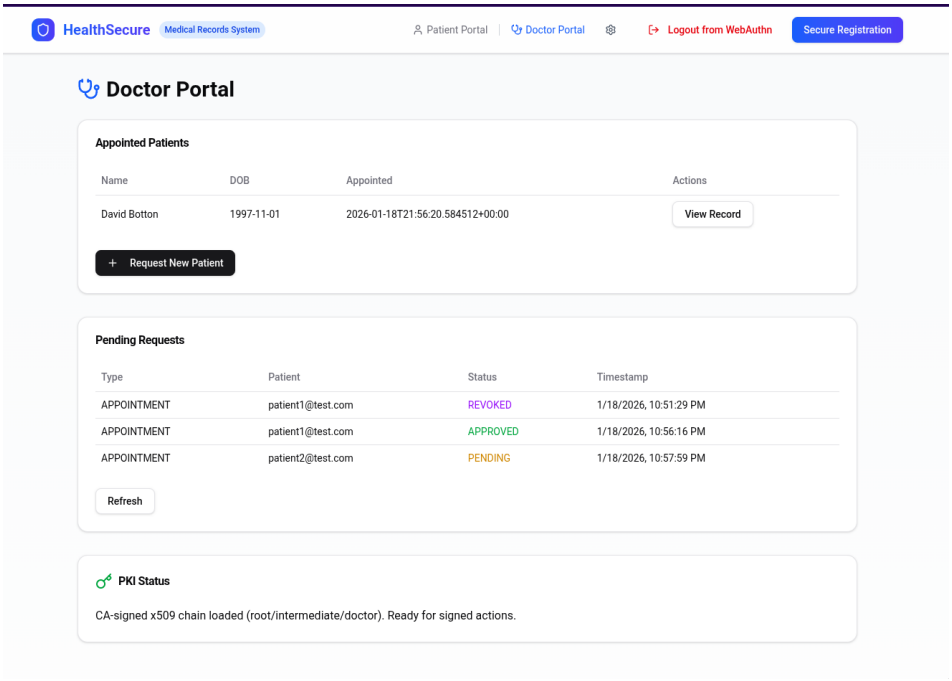


Figure 6

4. Login as the **patient**
5. Open pending appointments
6. Approve the pending request

3.5.5 Doctor Requests a File Change and Patient Approves (Pending Requests)

Doctor actions impacting patient data require patient approval via pending requests.

1. Login as a **doctor**
2. Create a file-related request (e.g., add/update/delete)

3. Login as the **patient**
4. View pending file requests
5. Approve or deny the request

3.5.6 Remove a Doctor (Revocation)

Removing a doctor revokes access by deleting the encrypted DEK entry associated with that doctor.

1. Login as a **patient**
2. Remove an appointed doctor

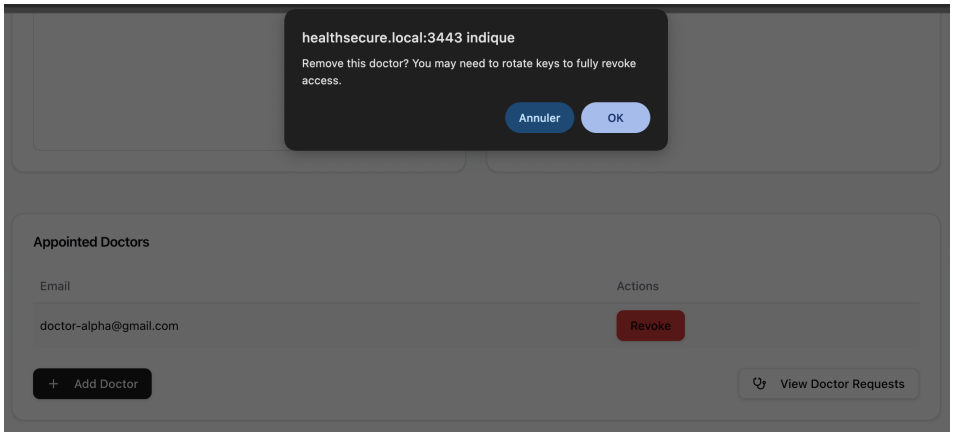


Figure 7

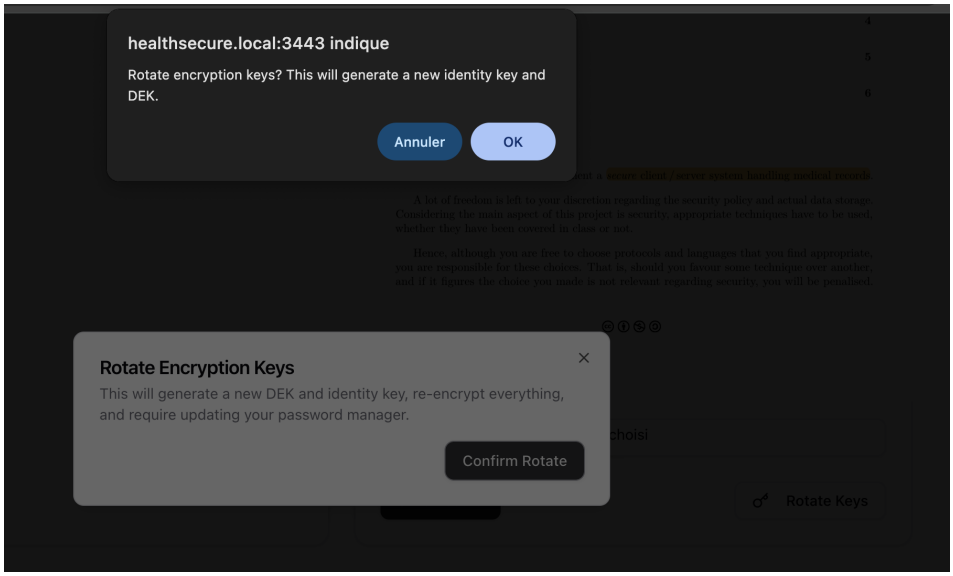


Figure 8

3.6 Verifying Encrypted Storage

To demonstrate that the server stores only ciphertexts:

1. Dump the database content:

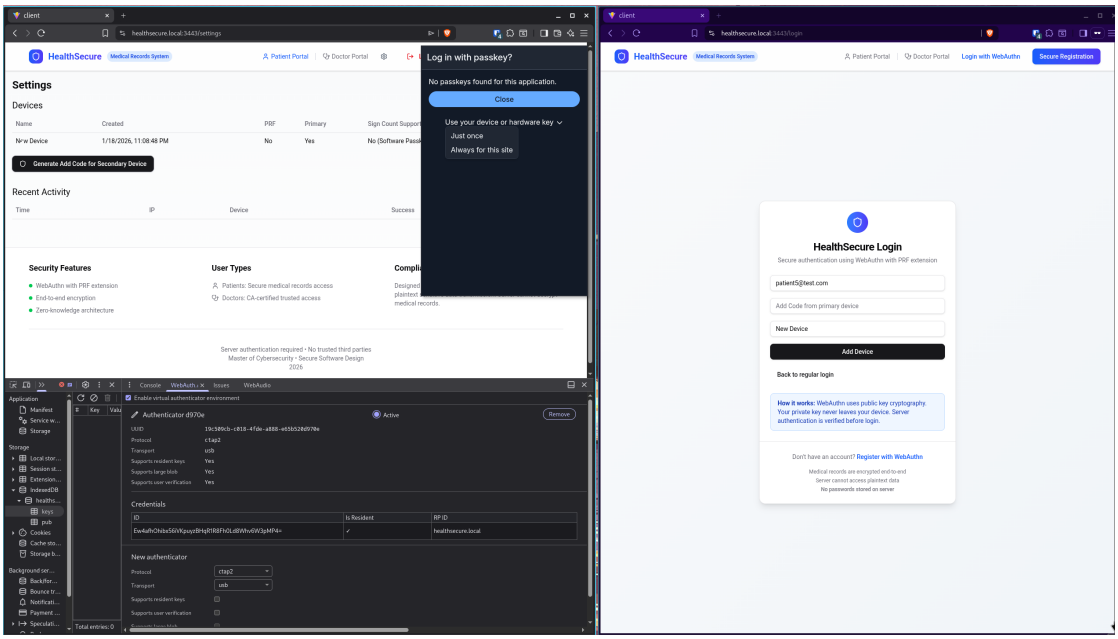
```
docker compose exec server python manage.py dumpdata > data.
    json
```

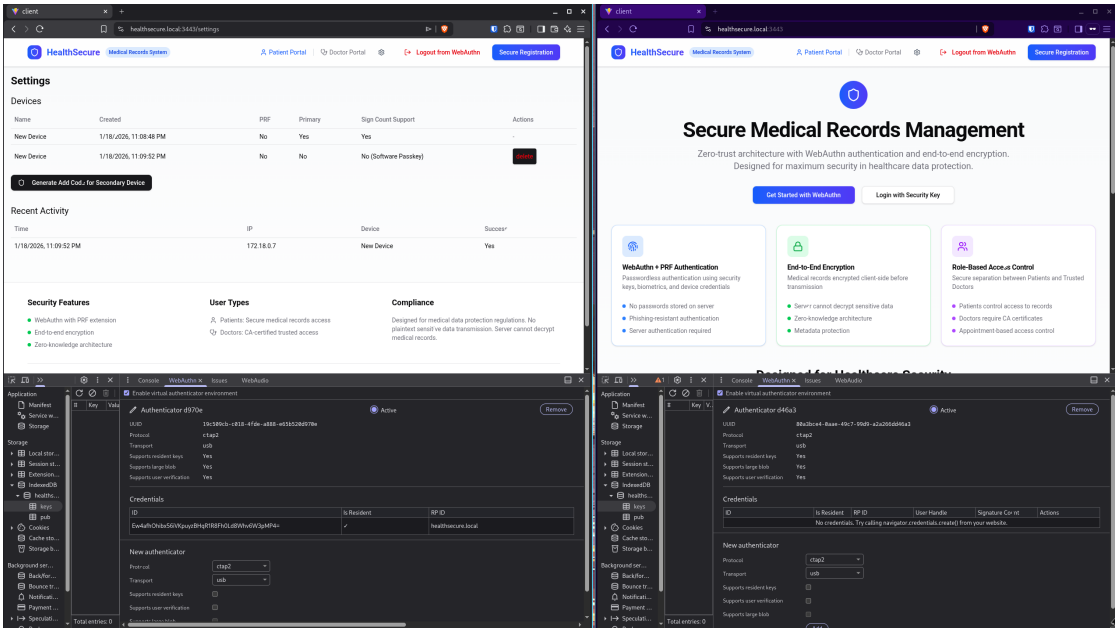
2. Inspect `data.json`. Medical records, keys, and sensitive fields appear exclusively as encrypted blobs.

No plaintext medical content is present in persistent storage.

3.7 Multi device with chrome dev tools

Password managers such as Bitwarden synchronize passkeys across devices, which makes every device act as a primary device. Using a hardware security key would allow a clear primary/secondary device distinction. However, this behavior can be simulated using Chrome DevTools. Additionally, Bitwarden and similar password managers do not support sign counters for passkeys. The sign count value always remains 0.





3.8 Logs and Monitoring

The backend logs all sensitive actions using structured metadata (timestamp, action type, size, privilege level). Logs can be inspected directly from the container output.

```
docker compose logs server
```

Logged metadata enables anomaly detection and post-incident analysis while avoiding disclosure of sensitive content.

4 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).

4.1 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).

4.2 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).

4.3 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).

4.4 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).

4.5 Monitoring and Additional Security

Logs record activity

5 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.

5.1 Doctor Registration & Credential Provisioning

The doctor registration sequence (Figure 9) covers identity onboarding, abuse-prevention (reCAPTCHA), PKI certificate issuance, and WebAuthn credential creation with optional PRF support for client-side key derivation.

1. **Input Validation & Bot Mitigation:** The client validates registration fields and obtains a reCAPTCHA v3 token, which is verified server-side to mitigate automated sign-ups.
2. **Certificate Issuance (PKI):** The client generates a CSR locally (or prepares key material as required) and requests signing through the backend CA API, which forwards the request to the Step-CA signer. The resulting doctor certificate bundle is returned to the client.
3. **WebAuthn Registration Start:** The client starts WebAuthn registration via `/api/webauthn/register/start/` and receives a challenge and PRF salt/options.

4. **Authenticator Attestation & PRF:** The authenticator creates the credential and returns attestation. If PRF results are available, they are used client-side to derive a deterministic KEK for encrypting sensitive key material.

5. **Server Finalization:** The client finishes registration via `/api/webauthn/register/finish/`. The server stores the new credential and public keys, enabling future zero-trust logins and secure key recovery.

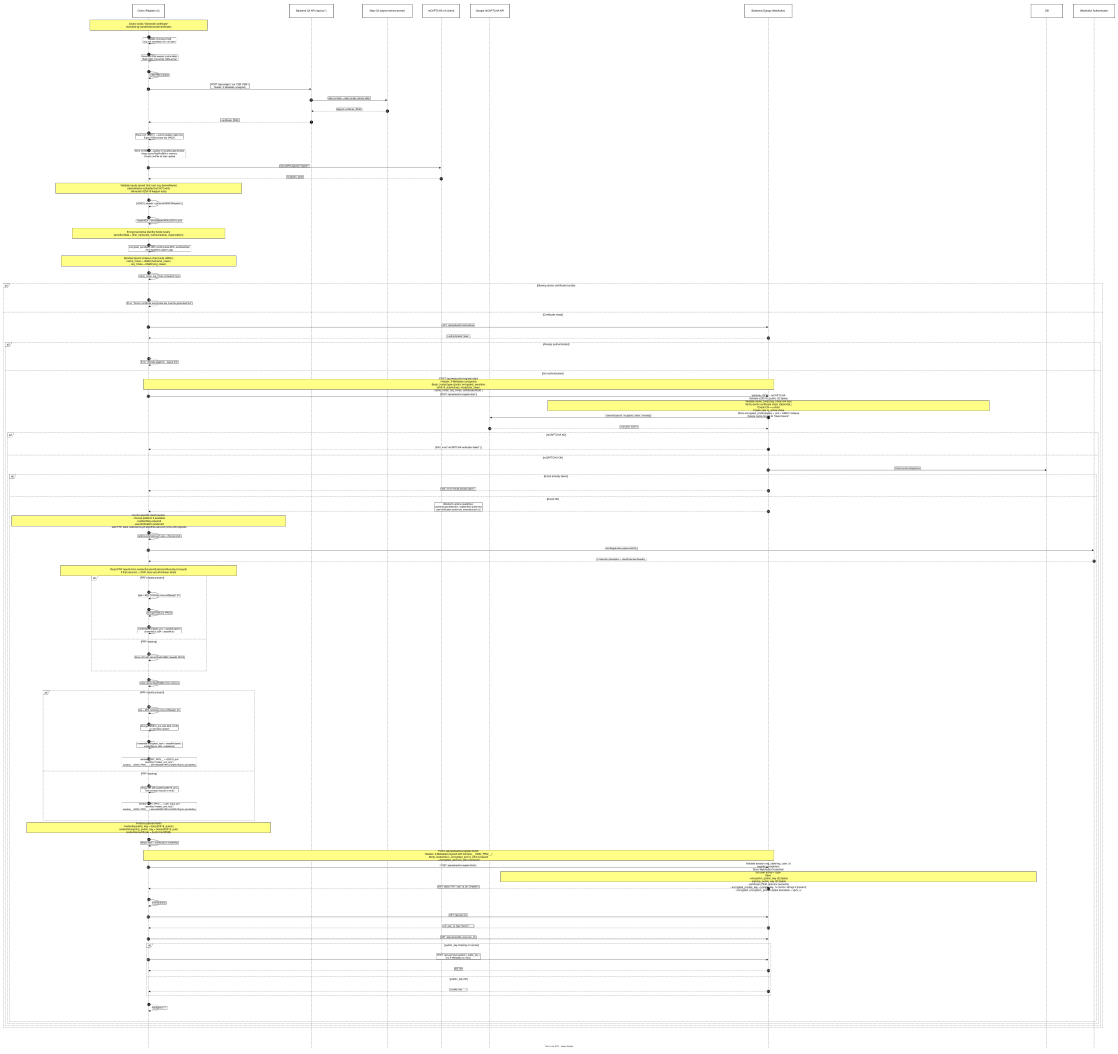


Figure 9: Sequence Diagram: Doctor Registration (PKI, reCAPTCHA, and WebAuthn Credential Creation)

5.2 Patient Registration & Initial Cryptographic Setup

The patient registration sequence (Figure 10) establishes the patient's WebAuthn credential, enables PRF-based client-side KEK derivation when available, and initializes the cryptographic foundation required to protect records end-to-end.

1. **Input Validation & reCAPTCHA:** The client validates inputs and requests a reCAPTCHA v3 token. The backend verifies the token to reduce abuse.

2. **WebAuthn Registration Start:** The client calls `/api/webauthn/register/start/` and receives a challenge and PRF configuration/salt.

3. **Credential Creation:** The authenticator produces an attestation response. If PRF results are present, the client can derive a KEK deterministically for local key wrapping/unwrapping.

4. **Registration Finish:** The client submits `/api/webauthn/register/finish/`. The server persists the credential and public identity material.

5. **Record System Initialization:** The client calls `/api/record/init_dek/` to initialize the patient record cryptographic baseline (e.g., initial DEK/structure), ensuring the record subsystem is ready for encrypted uploads.

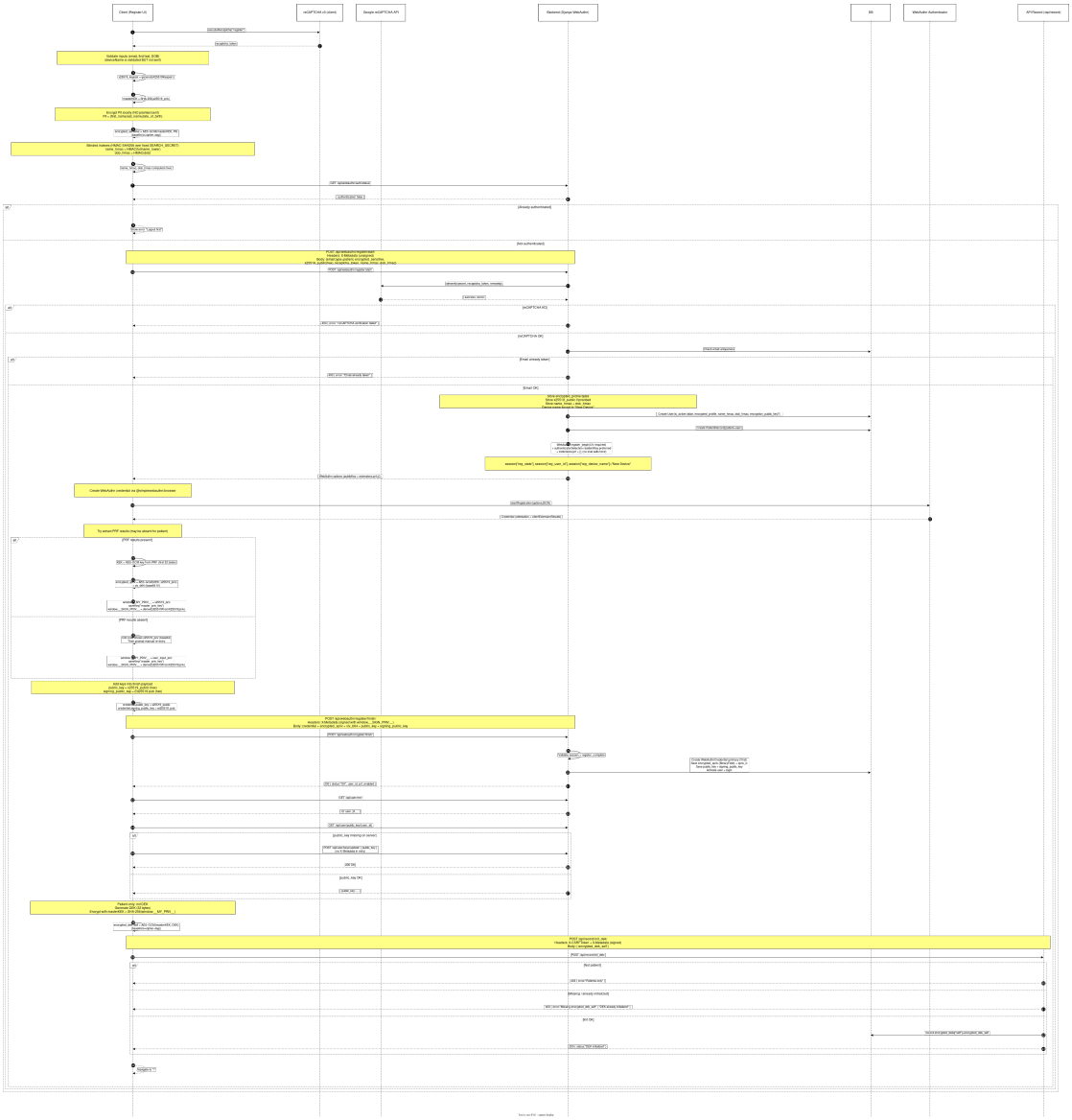


Figure 10: Sequence Diagram: Patient Registration (reCAPTCHA, WebAuthn, and Initial Record Crypto Setup)

5.3 Doctor Login & Key Recovery

The doctor login process, detailed in the sequence diagram (Figure 11), 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.

Figure 11: Sequence Diagram: Doctor Login...

5.4 Doctor Logout

The doctor logout sequence (Figure 12) ensures both server-side session invalidation and client-side key/material cleanup to reduce residual risk after the session ends.

1. **CSRF & Metadata Preparation:** The client retrieves the CSRF token from cookies and prepares request metadata. If a signing private key is available, metadata is signed before sending.
2. **Logout Request:** The client calls `POST /api/webauthn/logout/` with session credentials and CSRF protection.
3. **Server Session Invalidation:** The backend logs the event, executes `django.logout(request)`, and flushes the session store to invalidate the session server-side.
4. **Client Cleanup:** The client clears local sensitive material (e.g., IndexedDB keystore entries and volatile globals such as `window.__KEK__` and other session keys), and removes relevant cookies where applicable.

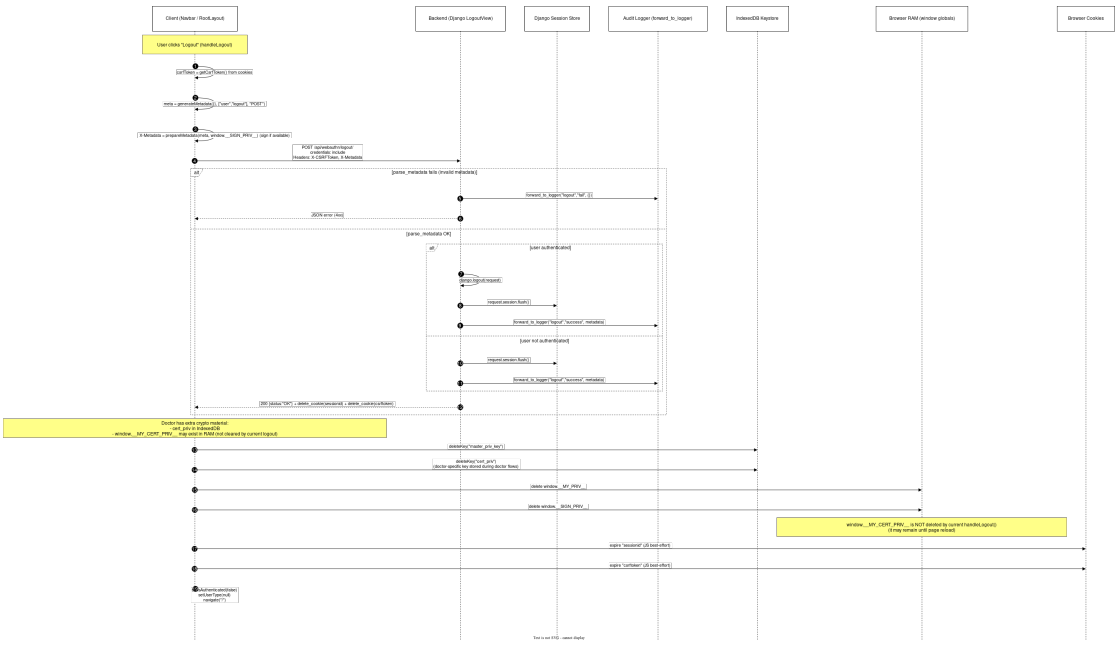


Figure 12: Sequence Diagram: Doctor Logout (Server Session Flush and Client Key Cleanup)

5.5 Patient Login & Key Recovery

The patient login flow (Figure 13) 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.

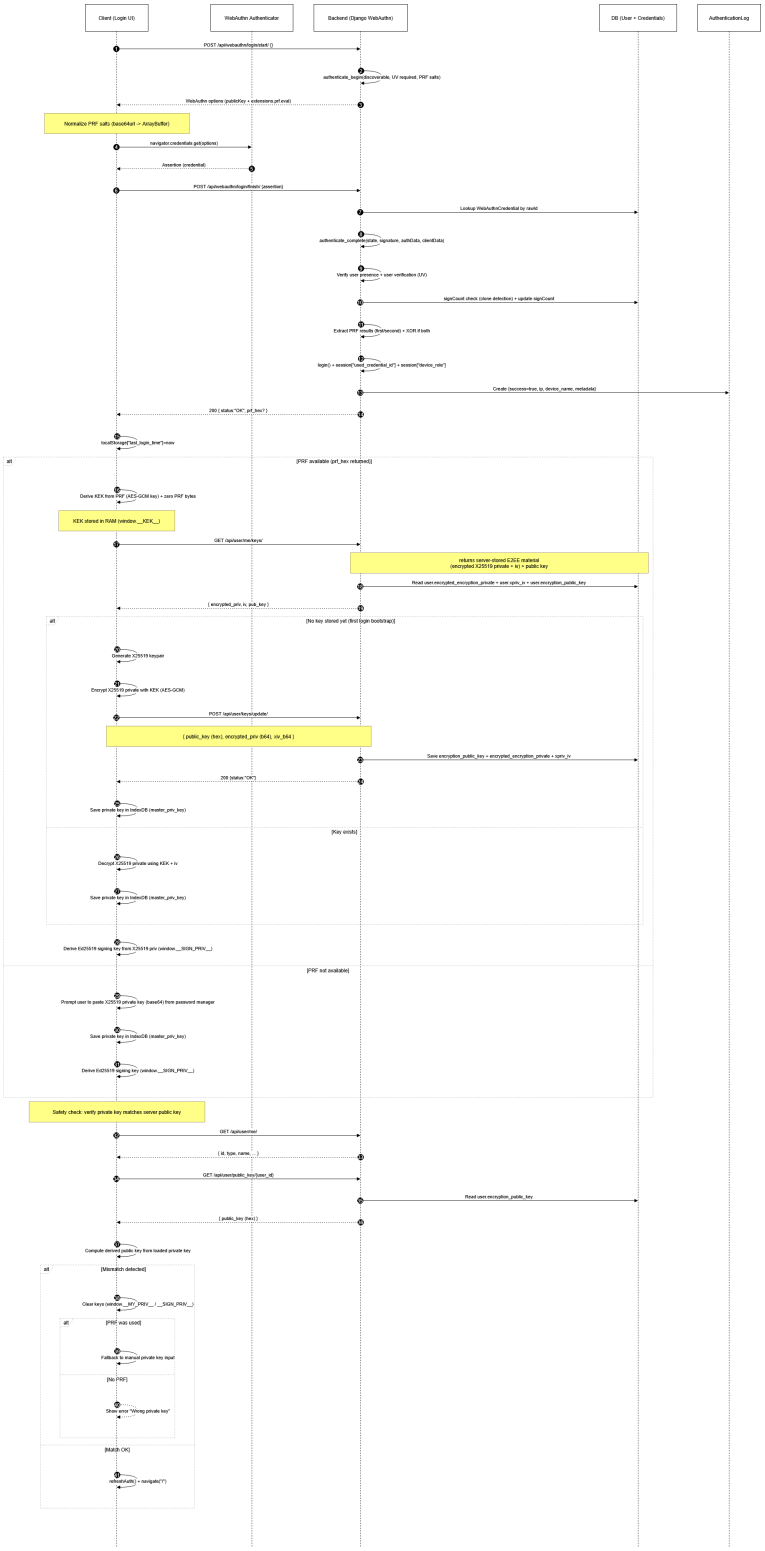


Figure 13: Sequence Diagram: Patient Login and E2EE Material Recovery

5.6 Patient Logout

The patient logout sequence (Figure 14) mirrors the doctor logout guarantees: server-side session revocation and client-side removal of cryptographic context to minimize key persistence.

1. **CSRF & Metadata Preparation:** The client obtains the CSRF token and prepares request metadata (signed when a signing key is available).
2. **Logout Request:** The client calls `POST /api/webauthn/logout/` including session credentials and CSRF protection.
3. **Session Termination:** The backend logs the action and invalidates the session by flushing the server-side session store.
4. **Local Crypto Context Cleanup:** The client clears local encrypted keystore artifacts and volatile session keys, ensuring the E2EE context is not left accessible after logout.

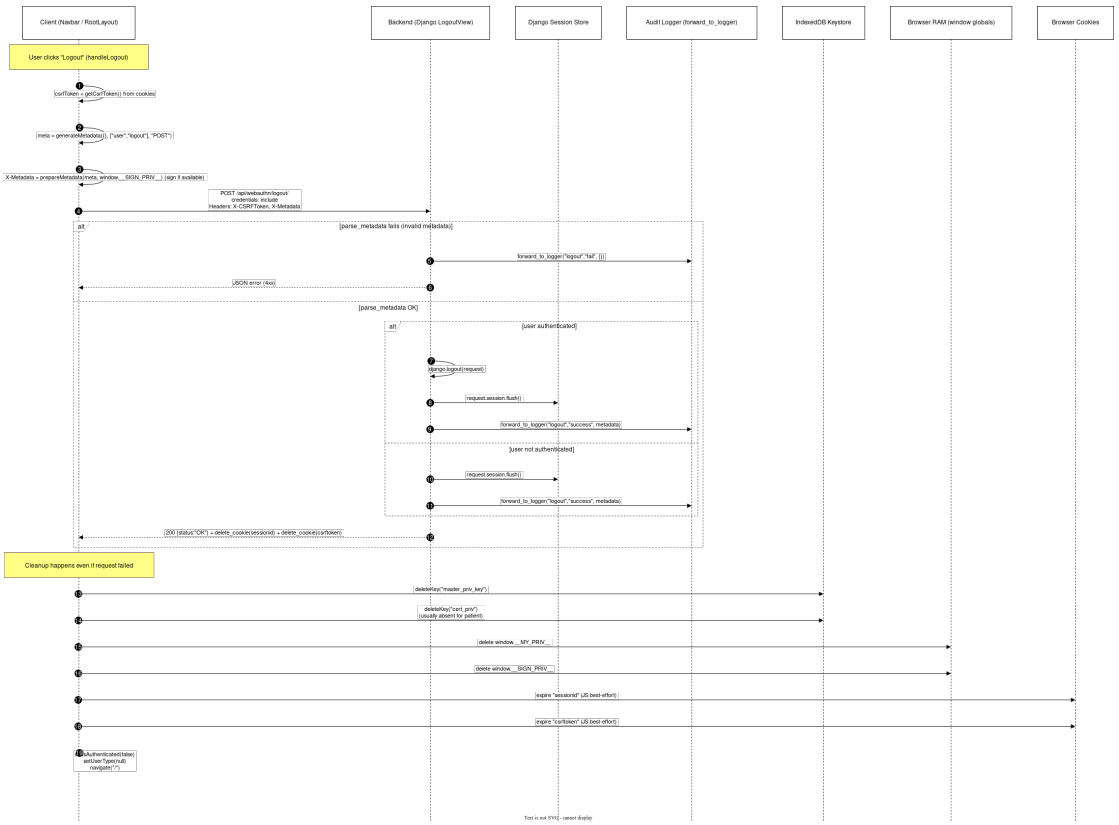


Figure 14: Sequence Diagram: Patient Logout (Session Invalidation and Local Cryptographic Context Cleanup)

5.7 Key Rotation & Revocation

The key rotation protocol (Figure 15) 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.

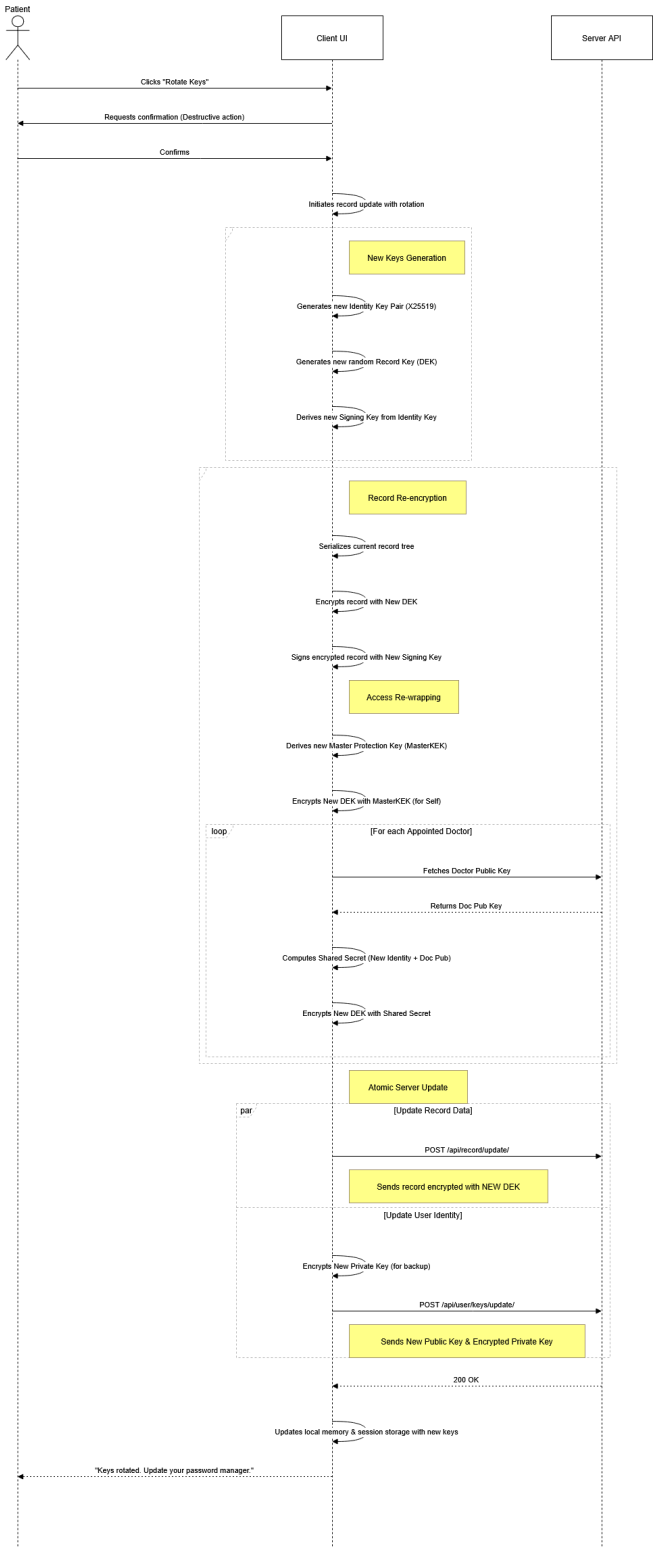


Figure 15: Sequence Diagram: Atomic Key Rotation and Data Re-encryption

5.8 Multi-Device Enrollment & Credential Management

The multi-device sequence (Figure 16) illustrates how additional authenticators/devices can be enrolled (or removed) under user control, with strong verification and audit logging to prevent unauthorized device takeover.

1. **Credential Inventory & Activity:** The primary device retrieves registered credentials and recent activity (e.g., `/api/webauthn/user/credentials/`, `/api/webauthn/user/activi` to provide user visibility and support risk review.
2. **Add-Device Approval Start (Primary):** The primary device initiates an approval flow to add a new credential, producing an approval state (and often a short-lived code or state reference) bound to the authenticated session.
3. **Secondary Device Enrollment Start:** The secondary device submits `/api/webauthn/add/start/` (e.g., {email, code, device_name}) to bootstrap enrollment without granting implicit trust.
4. **Primary Assertion Approval:** The primary device completes approval via a WebAuthn assertion, ensuring user verification and possession before any new credential is accepted.
5. **Secondary Attestation Finish:** The secondary device completes attestation to register its credential; server-side checks (e.g., UV verification / sign counter / clone detection) are enforced.
6. **Credential Removal (Optional):** Removing a credential follows a similar approval pattern to prevent attackers from locking out legitimate devices.

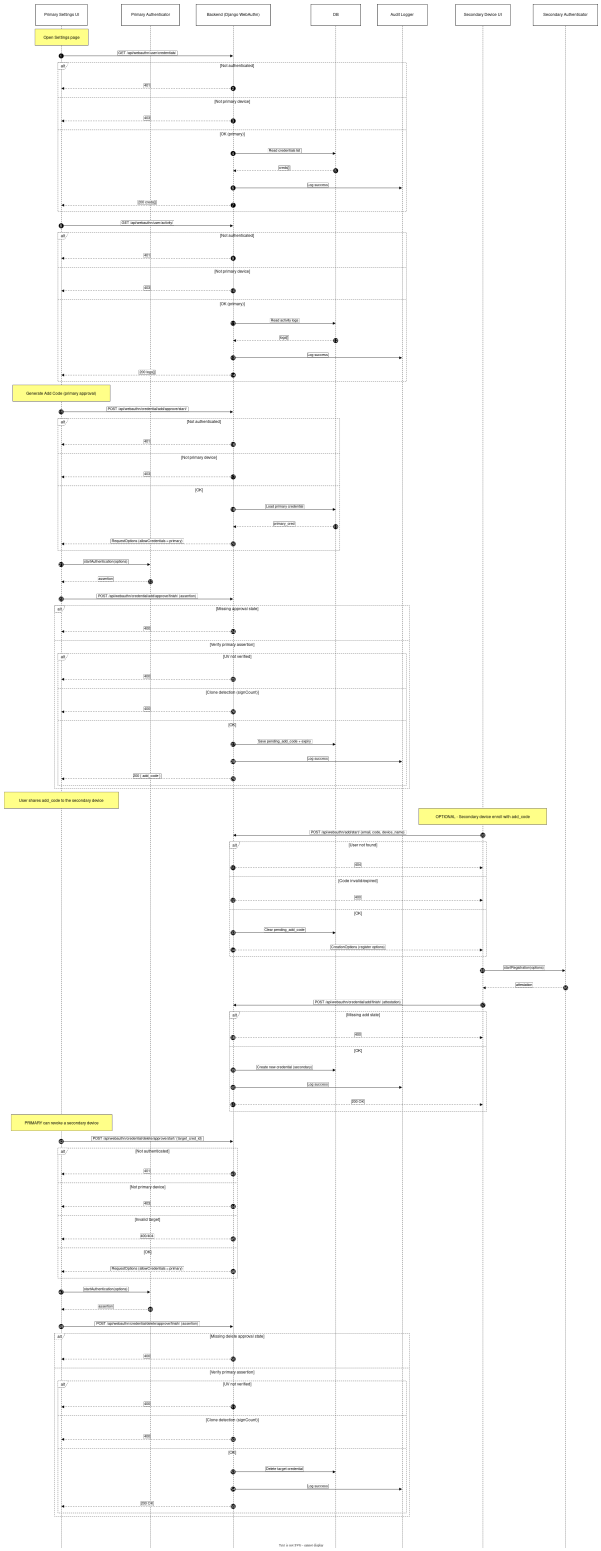


Figure 16: Sequence Diagram: Multi-Device Enrollment, Approval, and Credential Lifecycle Management

5.9 Doctor Patient Search (Authenticated Query & Auditing)

The doctor search sequence (Figure 17) shows how an authenticated doctor searches patient entries with integrity-protected request metadata and comprehensive audit logging.

1. **Search Input Normalization:** The doctor enters a query and selects a search field (e.g., email/name). The client normalizes input prior to request construction.
2. **Request Integrity (Metadata/HMAC):** The client prepares request metadata; where configured, it computes an HMAC over relevant parameters and signs metadata when possible to prevent tampering.
3. **Search Request:** The client calls `GET /api/patients/search/` with the search parameters and attached metadata headers.
4. **Authorization & Validation:** The backend enforces authentication and doctor role checks, validates required parameters, and rejects invalid field selections.
5. **Database Lookup & Response:** The server queries the user table and returns matching results.
6. **Audit Logging:** The server forwards a success/failure event to the audit logger with relevant (non-sensitive) metadata for traceability.

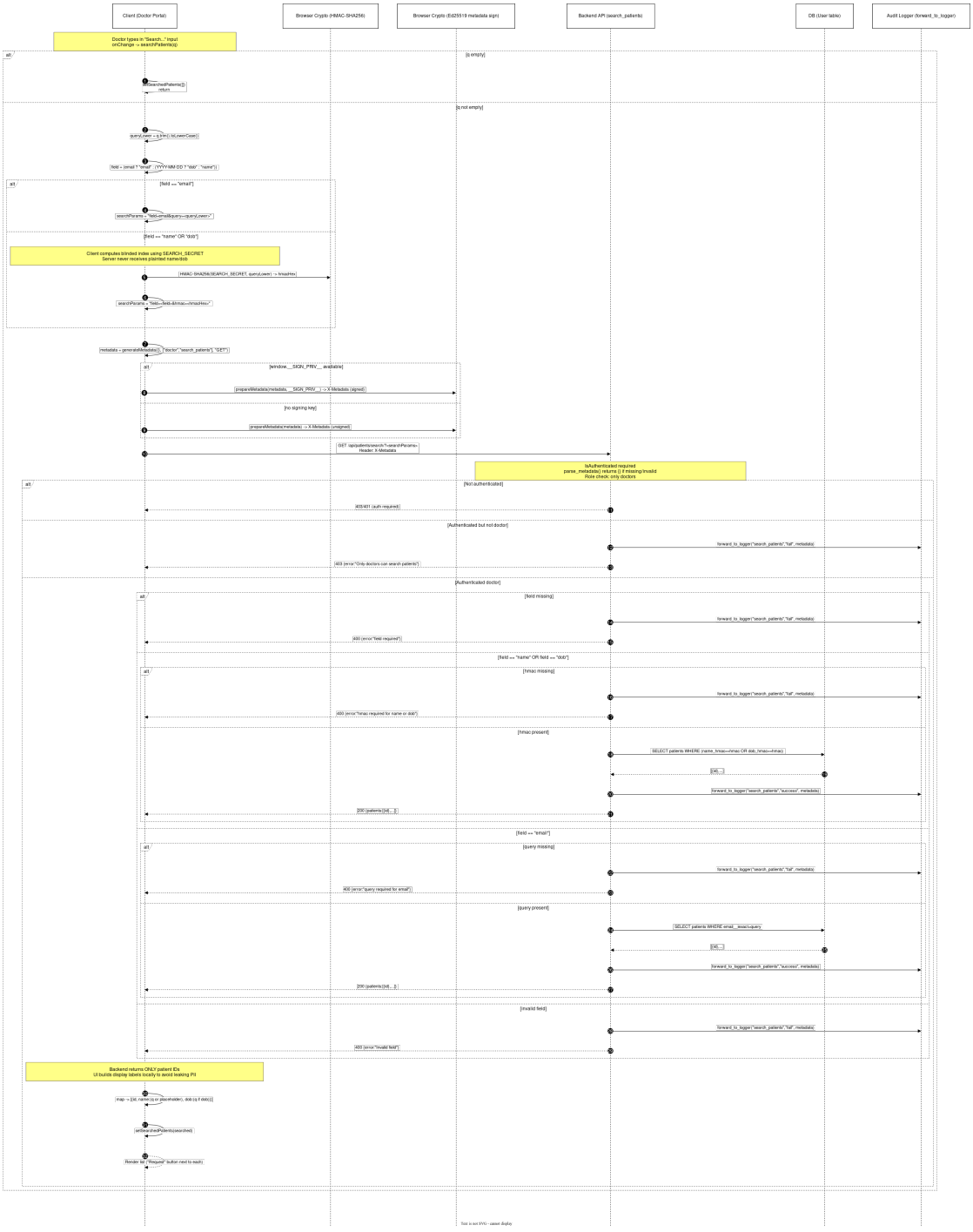


Figure 17: Sequence Diagram: Doctor Searches Patients (Role-Gated API, Integrity-Protected Metadata, and Audit Trail)

5.10 Patient Doctor Search (Role-Gated Discovery)

The patient search sequence (Figure 18) illustrates how a patient discovers doctors through an authenticated endpoint with optional request metadata signing and audit logging.

1. **Query Entry:** The patient types a query in the UI; empty queries may short-circuit client-side to avoid unnecessary requests.
2. **Metadata Signing (Optional):** If a signing key is available in the session, the client signs request metadata to provide integrity guarantees; otherwise it sends unsigned metadata.
3. **Search Request:** The client calls `GET /api/doctors/search/?q=<q>` with metadata headers.
4. **Authorization & Role Check:** The backend enforces authentication and ensures the caller is a patient before proceeding.
5. **Database Query & Result List:** The server queries the doctor directory/user table and returns a list of doctor identifiers for UI rendering.
6. **Audit Logging:** The outcome is forwarded to the audit logger for traceability and anomaly detection.

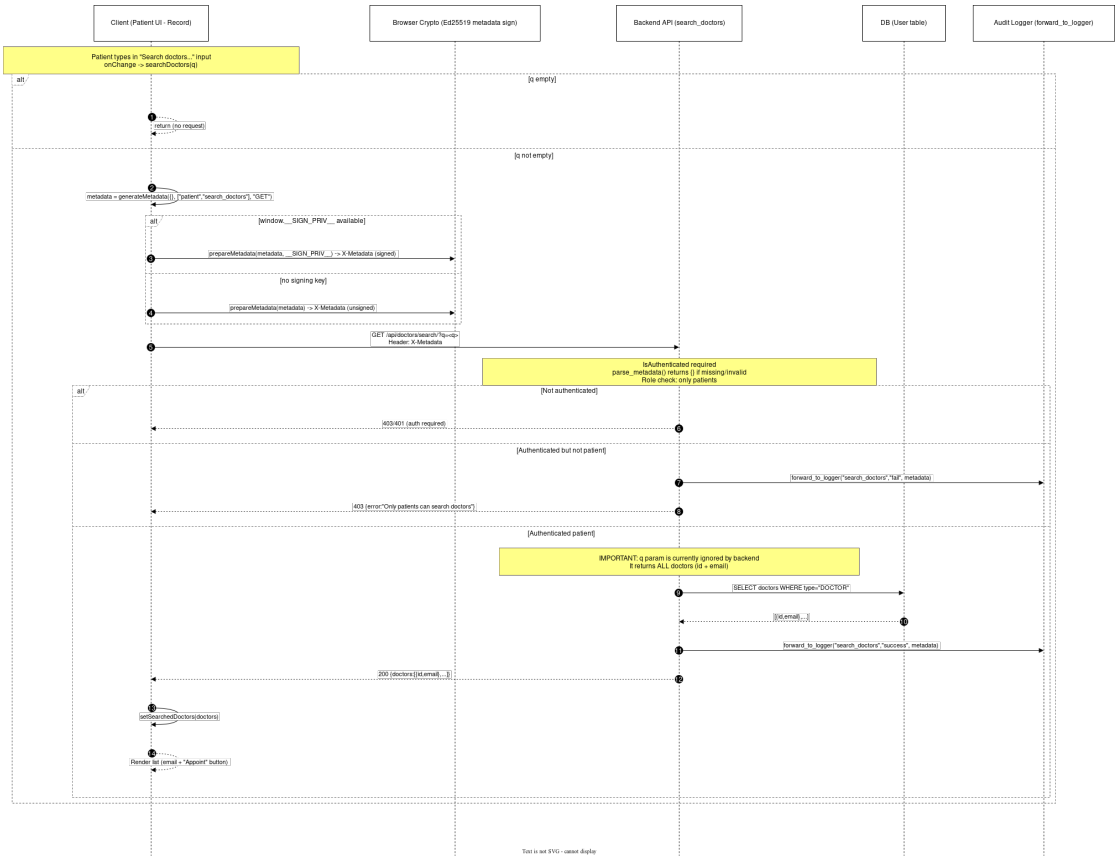


Figure 18: Sequence Diagram: Patient Searches Doctors (Authenticated Endpoint, Optional Signed Metadata, and Auditing)

5.11 Doctor-Initiated Appointment & Approval

This protocol (Figure 19) 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."

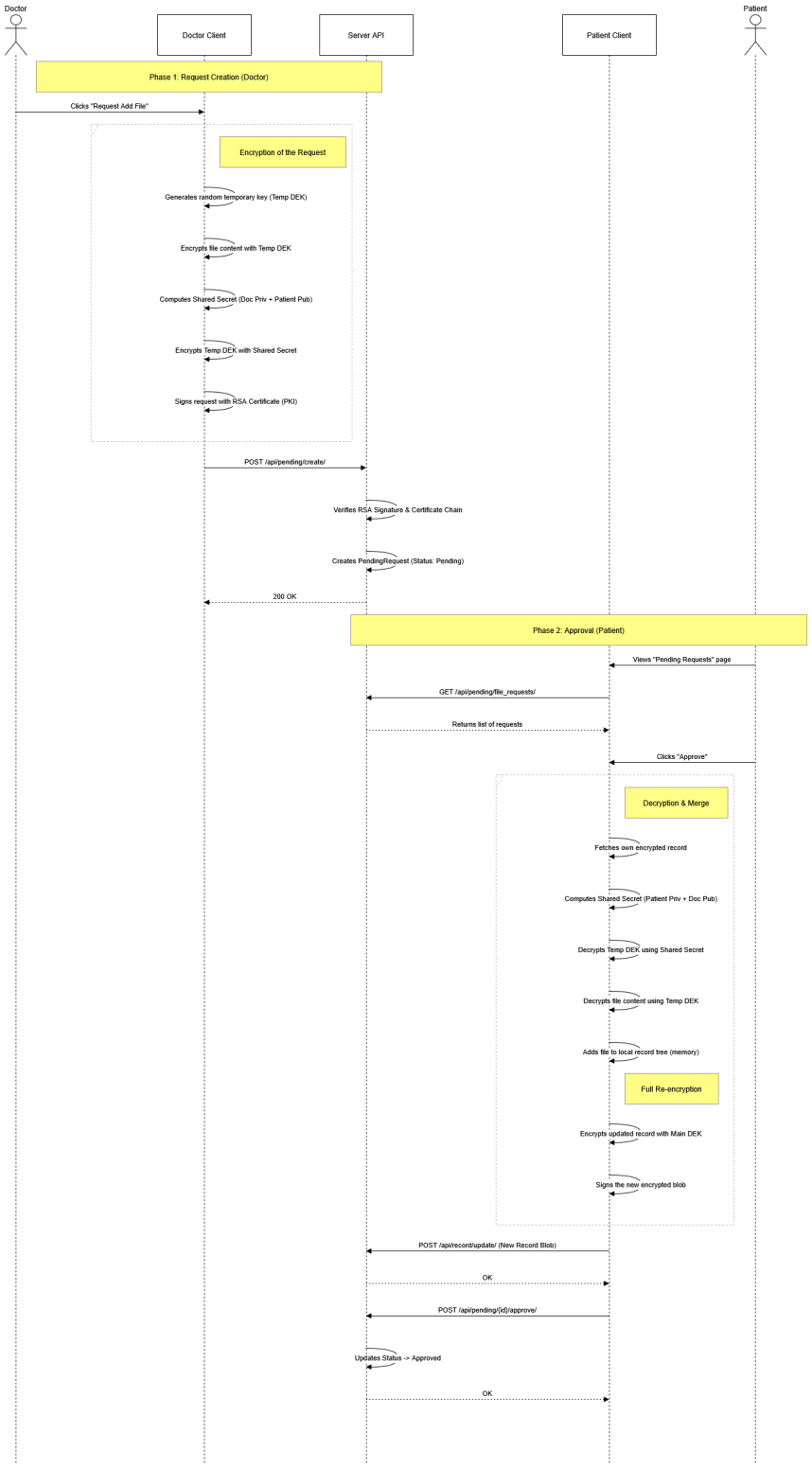


Figure 19: Sequence Diagram: Doctor-Initiated Request and Patient Approval

5.12 Patient Record Creation & Upload

This sequence (Figure 20) 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).

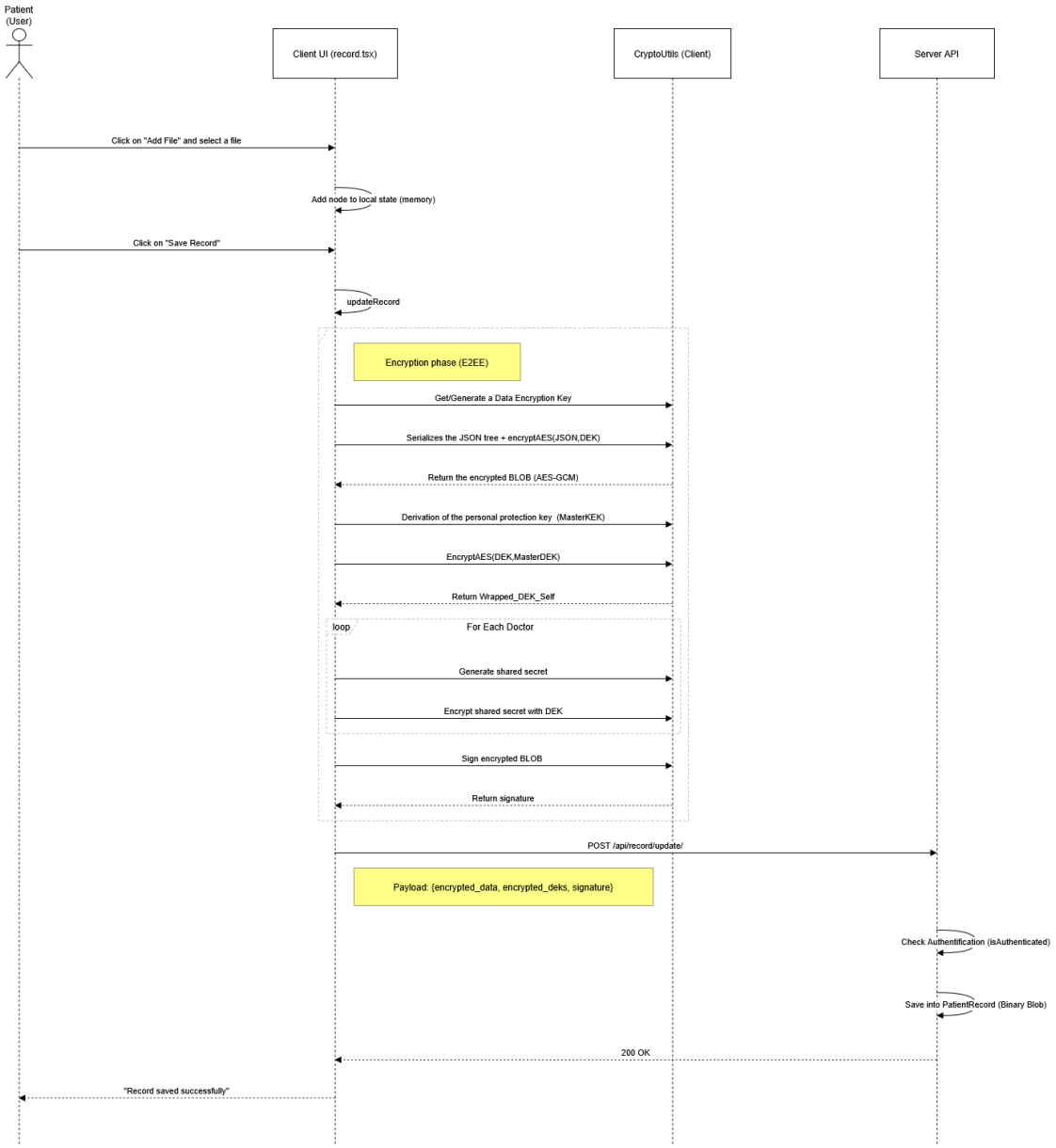


Figure 20: Sequence Diagram: Secure Record Creation and Upload by Patient

6 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

7 Security Checklist Analysis

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

7.1 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)

7.2 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)

7.3 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)

7.4 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`).