Secure Share - Project 2

SIO - Segurança Informática e Nas Organizações

Project Report Submitted by:

João Barreira (120054)

Gonçalo Almeida (119792)

Margarida Ribeiro (119876)

DETI
University of Aveiro

December 7, 2025

Abstract

This report presents Secure Share, a single-tenant secure file transfer web application implementing end-to-end encryption, role-based access control (RBAC) and a formal multi-level security (MLS) scheme based on the Bell-LaPadula confidentiality model with departmental categories. The system was developed using FastAPI backend with SQLite database persistence and a Python-based command-line interface (CLI).

Secure Share's core security architecture ensures that all file content is encrypted client-side using AES-256-GCM before transmission, with server never having access to plain text data. The system employs a hybrid encryption scheme where file encryption keys are protected using RSA-4096-OAEP for private shares. User private keys are secured using PBKDF2-HMAC-SHA256 key derivation with 480,000 iterations and stored as encrypted vault blobs on the server.

The MLS implementation enforces hierarchical security levels (Unclassified, Confidential, Secret, Top Secret) combined with non-hierarchical department categories, using cryptographically signed JWT tokens with RS256 signatures issued by Security Officers. The system implements both the Simple Security Property (no read-up) and the Property (no write-down) of the Bell-LaPadula model.

All system interactions are recorded in an audit log secured by a SHA-256 hash chain, enabling independent verification by designated Auditors. Communication security is enforced through TLS 1.3 using a custom certificate authority for development purposes.

Contents

Abstract **i**

1 Introduction **1**
1.1 Motivation 1
1.2 Project Objectives 1

2 System architecture **2**
2.1 High-Level Architecture and Technology Stack 2

3 Cryptographic Implementation **5**
3.1 Cryptographic Primitives Overview 5
3.2 End-to-End Encryption 6
3.2.1 File Upload Process 6
3.2.2 File Download Process 9
3.3 User Key Management 12
3.3.1 Key Pair Generation 12
3.3.2 Private Key Protection: The Vault System 12
3.3.3 Public Key Distribution 12
3.4 Password Security 13
3.4.1 Authentication Password Hashing with Argon2 13
3.4.2 Vault Password Key Derivation with PBKDF2 13
3.5 Digital Signatures and Token Integrity 14
3.5.1 Signature Algorithm: RS256 14
3.5.2 Token Revocation 14
3.6 Integrity Mechanisms 14
3.6.1 Audit Log Hash Chain 14
3.7 Summary 15

4 Authentication and Session Management **16**

5 Role-Based Access Control **19**
5.1 Roles and Responsibilities 19

5.2 Role Assignment and Storage Model 20
5.3 Presentation and Verification 20
5.4 Revocation and Token Life Cycle 20

6 Multi-Level Security (MLS) 21
6.1 Clearance Levels and Categories 21
6.2 MLS Token Format, Issuance and Storage 21
6.3 MLS Verification 22
6.4 Bell-LaPadula Rules 22
6.4.1 Public Shares 22
6.4.2 User-specific Shares 23
6.5 Trusted Officer bypass 23
6.6 Revocation and Token Life Cycle 23
6.7 Summary 23

7 Hash Chain Auditing Mechanism 24
7.1 Purpose and Security Objectives 24
7.2 Hash Chain Construction 24
7.3 Verification by Auditors 24
7.4 Access Control 25

8 Security Decisions 26
8.1 Self Token Issuance Prevention 26
8.2 Administrator Immutability 26
8.3 Private Key Handling: Zero-Knowledge Architecture 27
8.4 CLI Architecture 27
8.5 Password Peppering 27

9 Static Code Security Analysis with SonarQube 29

10 Conclusion 31
10.1 Key Achievements 31
10.2 Security Analysis and Validation 32
10.3 Final Remarks 32

List of Figures

2.1 Example of a dockerized CLI handshake with the server. 2
2.2 High-Level Architecture 4

3.1 File Upload Process 8
3.2 File Download Process for User-specific Shares 10
3.3 File Download Process for Public Shares 11

4.1 Account Activation Workflow 17

9.1 SonarQube First Analysis 29
9.2 SonarQube After Solving Security Issues 30

List of Tables

3.1 Cryptographic Primitives and Algorithms used in SecureShare 5

4.1 Token Specifications and Storage 18

Chapter 1

Introduction

1.1 Motivation

In modern organizations, secure sharing of sensitive documents is paramount. Traditional file-sharing services often rely on server-side encryption, creating a central point of trust where plain text data is accessible to system administrators or could be compromised through server breaches. Furthermore, many solutions lack formal access control models that can consistently enforce organizational security policies.

1.2 Project Objectives

The primary objectives of this project were to:

- Design and implement a secure file transfer system with mandatory client-side encryption
- Enforce role-based access control (RBAC) with five distinct roles
- Implement a lattice-based MLS scheme combining hierarchical security levels with departmental categories
- Develop a cryptographically signed clearance token system
- Create a tamper-evident audit log using SHA-256 hash chains
- Ensure all communications occur over TLS 1.2+
- Provide a usable command-line interface for all system operations

Chapter 2

System architecture

2.1 High-Level Architecture and Technology Stack

The Secure Share system is built upon a modular, layered architecture that ensures a clear separation of concerns between the client interface, application logic and data persistence. As shown in the architecture diagram (Figure 2.1), the system is organized into three different layers: the Client Layer, the Application Layer and the Persistence Layer.

The Client Layer is the user-facing component, implemented as a Python-based Command-Line interface (CLI) using the typer framework. It communicates with the backend via secure HTTPS/TLS channels and is responsible for local operations, including the End-to-End Encryption of files before transmission.

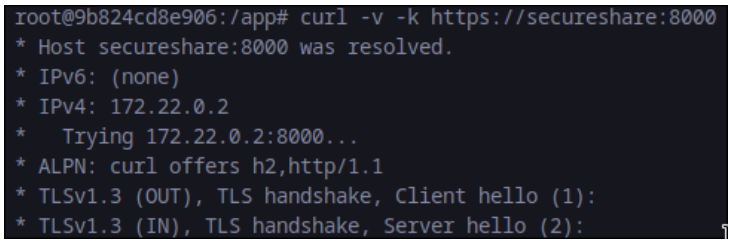

Figure 2.1. Example of a dockerized CLI handshake with the server.

The Application Layer, developed using Python and the FastAPI framework, serves as the system's core. It is structurally divided into two sub-components:

1. **FastAPI Router** - This component handles incoming HTTP requests and routes them to the appropriate modules, including Authentications, Users Management, Transfer operations, Department handling and Audit logging.

2. **Service Layer** - This layer encapsulates the business logic and orchestrates operations. It includes dedicated services for User Management, Department administration, Transfer processing, Audit logging, Authentication, Key Management and the enforcement of Multi-Level Security (MLS) policies.

Supporting these layers is the Persistence Layer, which utilizes SQLite for reliable data storage. This layer maintains the integrity of the system's data entities, including Users, Transfers, Departments, Roles, and Audit logs. It also securely stores various security tokens (Auth, MLS, RBAC, and Revocation) required for the system's authentication and authorization mechanisms.

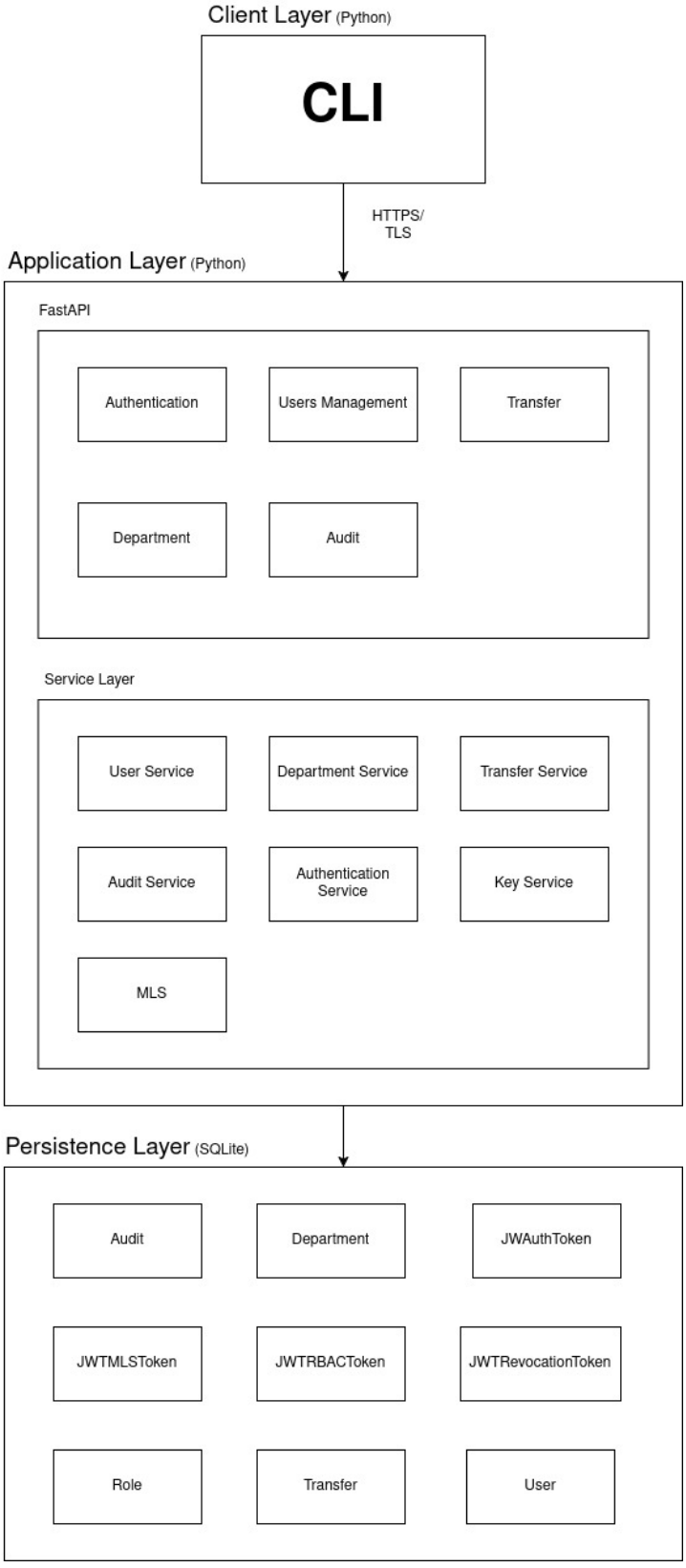

Figure 2.2. High-Level Architecture

Chapter 3

Cryptographic Implementation

All cryptographic operations that handle sensitive plain text data are performed exclusively on the client side. The server receives encrypted artifacts and is architecturally prevented from accessing plain text content. This chapter details cryptographic primitives, their implementation and the rationale behind it.

3.1 Cryptographic Primitives Overview

Secure Share employs a suite of cryptographic algorithms, each chosen for specific security properties and use cases; the implementation relies on the Python cryptographic library.

Table 3.1. Cryptographic Primitives and Algorithms used in SecureShare

Primitive	**Algorithm**	**Purpose**	**Implementation Location**
Symmetric Encryption	AES-GCM	File content encryption, vault encryption	`cli/core/crypto.py`
Asymmetric Encryption	RSA-OAEP	File key encryption for recipients	`cli/core/crypto.py`
Key Derivation	PBKDF2-HMAC-SHA256	Password-based key derivation for vaults	`cli/core/crypto.py`
Password Hashing	Argon2	User password storage	`backend/app/auth/service.py`
Digital Signatures	RSA-PKCS1v15 (RS256)	MLS/RBAC token signing	`backend/app/auth/service.py`
Hashing	SHA-256	Audit log hash chain, OAEP/PBKDF2	`backend/app/audit/service.py`

RSA-4096 was selected over elliptic curve cryptography to ensure long-term security against advances in quantum computing, despite the performance tradeoff. PBKDF2 with 480,000 iter-

ations meets current OWASP recommendations for password-based key derivation, providing resistance against brute-force attacks even if the encrypted vault is compromised.

3.2 End-to-End Encryption

The end-to-end encryption model ensures that the file content remains confidential from the moment it leaves the user's device until it is decrypted by an authorized recipient. This is achieved through a hybrid encryption scheme that combines the efficiency of symmetric encryption (AES-GCM) with the key management flexibility of asymmetric cryptography (RSA-OAEP). For public shares, the file content is still encrypted with a symmetric key, but this key is not encrypted with a specific recipient's public key. Instead, the symmetric key is managed by the system to allow access to any authenticated user within the organization, effectively bypassing the recipient-specific restriction while maintaining encryption at rest.

3.2.1 File Upload Process

When a user uploads a file, the encryption process occurs entirely within the CLI client before any data is transmitted to the server. The process follows these steps:

1. **File Key Generation**

   A cryptographically secure random 256-bit key is generated for each transfer. This key, referred to as the file key, will be used to encrypt the actual file content. The implementation uses Python's `os.urandom()`, which provides cryptographically strong random bytes.

   The use of a unique key per transfer ensures that even if one key is compromised, other transfers remain secure. This also enables efficient re-keying, if a file needs to be shared with additional recipients later, only new encrypted copies of the file key need to be generated, not new encrypted copies of the entire file.

2. **File Content Encryption with AES-256-GCM**

   The file content is encrypted using AES in Galois/Counter Mode. GCM is an authenticated encryption mode that simultaneously provides confidentiality and integrity protection.

   The resulting ciphertext includes an authentication tag (appended by the AESGCM implementation) that is verified during decryption. Any tampering with the ciphertext will cause decryption to fail, providing integrity guarantees. The nonce is transmitted alongside the ciphertext and does not need to be kept secret, but it must never be reused with the same key. Using a random 96-bit nonce ensures negligible collision probability even across billions of encryptions.

3. **File Key Encryption with RSA-OAEP (User-specific Shares)**

   For each authorized recipient, the file key is encrypted using their RSA public key with Optimal Asymmetric Encryption Padding (OAEP). OAEP is a padding scheme that protects against various attacks on textbook RSA, including chose-cipher attacks.

   The OAEP padding uses SHA-256 for both the mask generation function (MGF1) and the main hash function. This ensures that even if the same file key is encrypted for multiple recipients, each ciphertext will be different and unlinkable.

3. **URL Fragment Encoding (Public Share)**

   The raw file key is Base64-encoded and appended to the download URL as a URL fragment (after the #). Since URL fragments are not sent to the server by browsers or standard HTTP clients, the server never sees the decryption key. The key exists only on the up-loader's machine and on the recipient's machine.

4. **Upload Data Structure** After encryption, the client transmits the following data to the server via the `POST /transfers` endpoint. The server stores the encrypted blob as a file in the `storage/` directory, using the transfer id as the filename. The metadata, including the path to the blob and the encrypted file keys (if private share), is stored in the `transfers` table of the SQLite database.

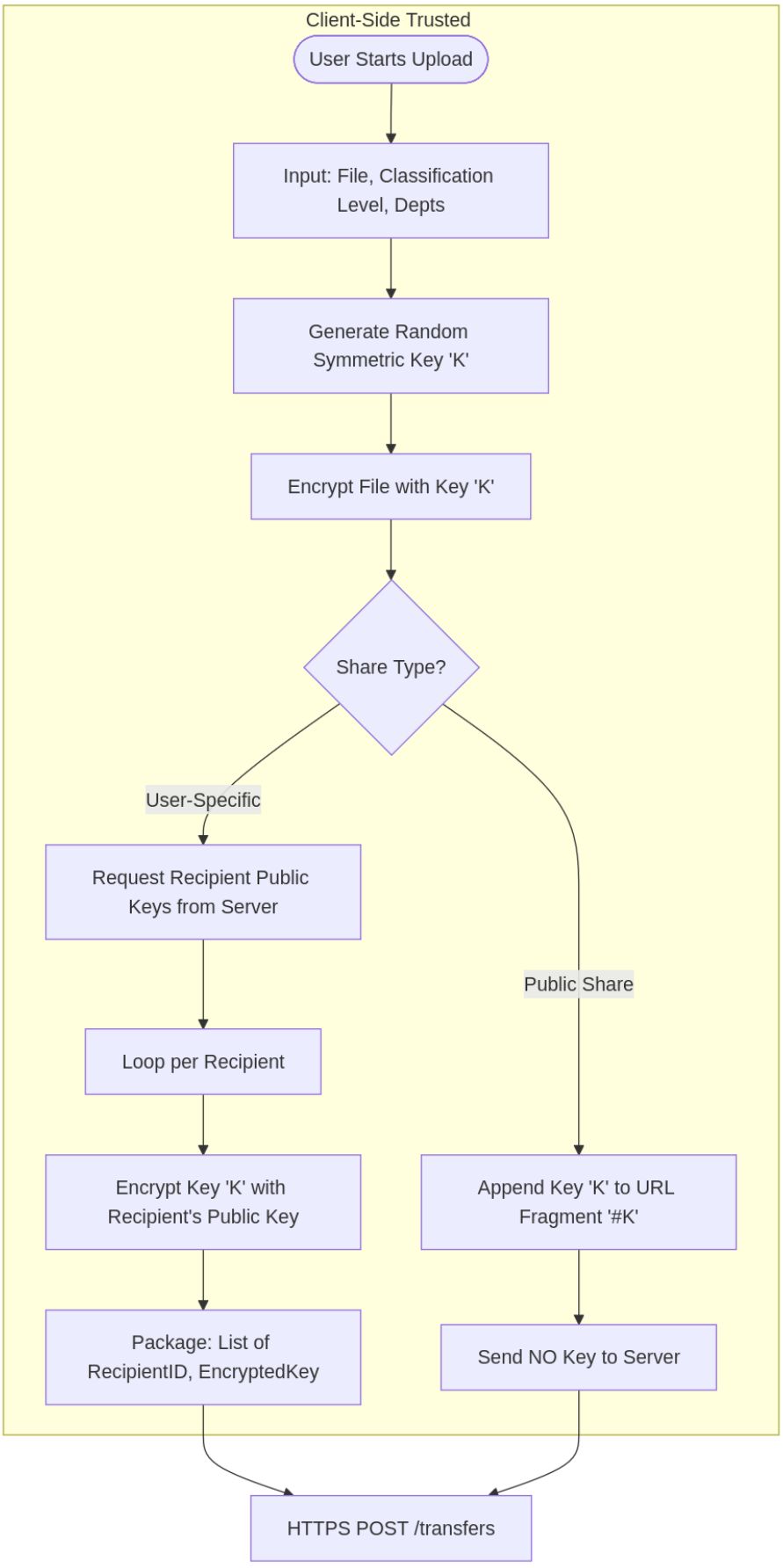

Figure 3.1. File Upload Process

3.2.2 File Download Process

When an authorized user downloads a file, the process reverses the encryption steps, again performing all cryptographic operations on the client side.

1. **Retrieve Encrypted Artifacts** The client makes a `GET /transfers/download/{transfer_id}` request, including authentication and clearance tokens.

   If the transfer is private, the server verifies the user's clearance against the MLS policy and returns:

   - The encrypted file blob (from the `storage/` directory with the nonce prepended
   - The encrypted file key specific to this user

   If the transfer is public it only returns the encrypted file blob with the nonce prepended

2. **Decrypt File Key with RSA Private Key (Private User-specific Shares)** The client loads the user's private key from their local vault and uses it to decrypt the file key. This operation recovers the original 256-bit AES key that was used to encrypt the file. The private key never leaves the client's memory and is never transmitted to the server.

2. **File Key Retrieval (Public Share)** Since the share is public, the server does not return an encrypted file key for the user (because none exists). Instead, the client extracts the file key directly from the URL fragment. The client parses the fragment, Base64-decodes it and obtains the AES file key directly.

3. **Decrypt File Content with AES-256-GCM** Using the recovered file key and the transmitted nonce, the client decrypts the file and verifies the authentication tag. If the cipher text has been tampered with in any way, the GCM authentication check will fail and an exception will be raised, preventing the client from receiving corrupted data. The recovered plain text is then saved to the user's local file system.

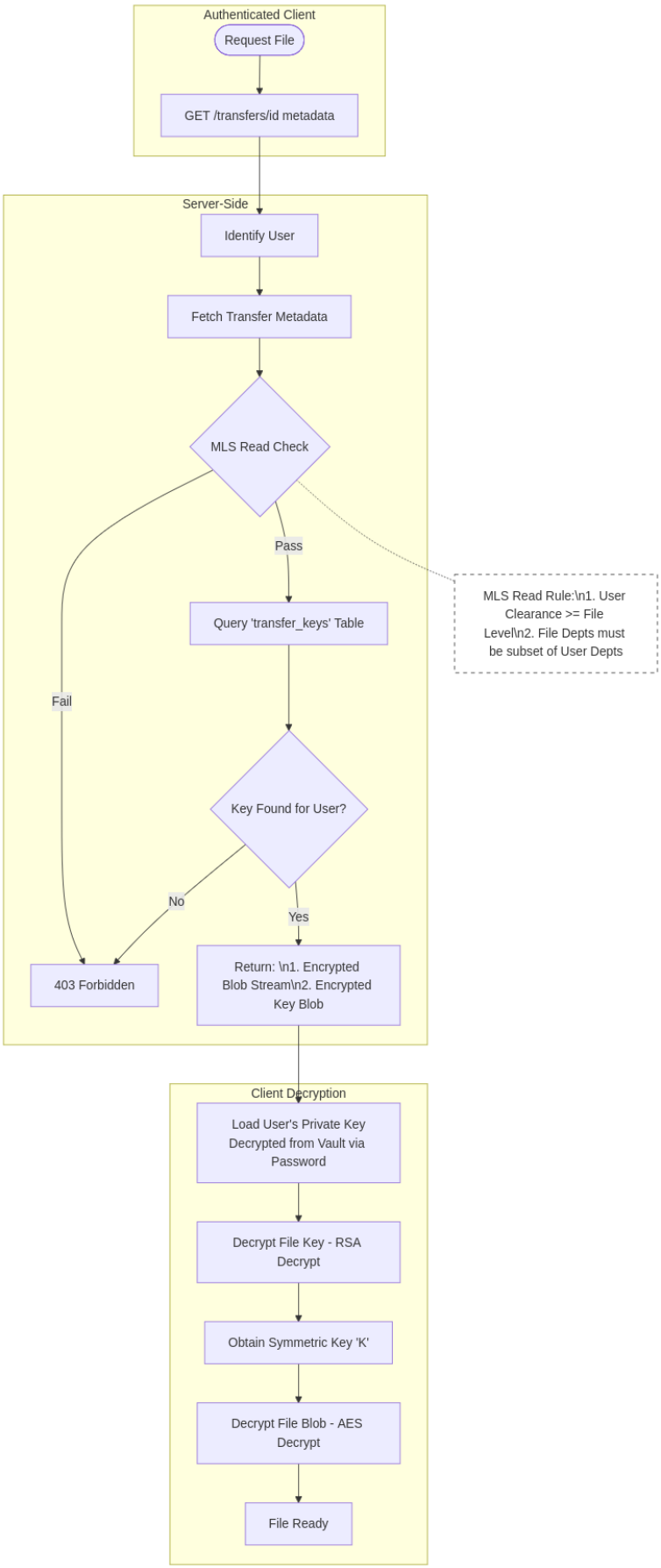

Figure 3.2. File Download Process for User-specific Shares

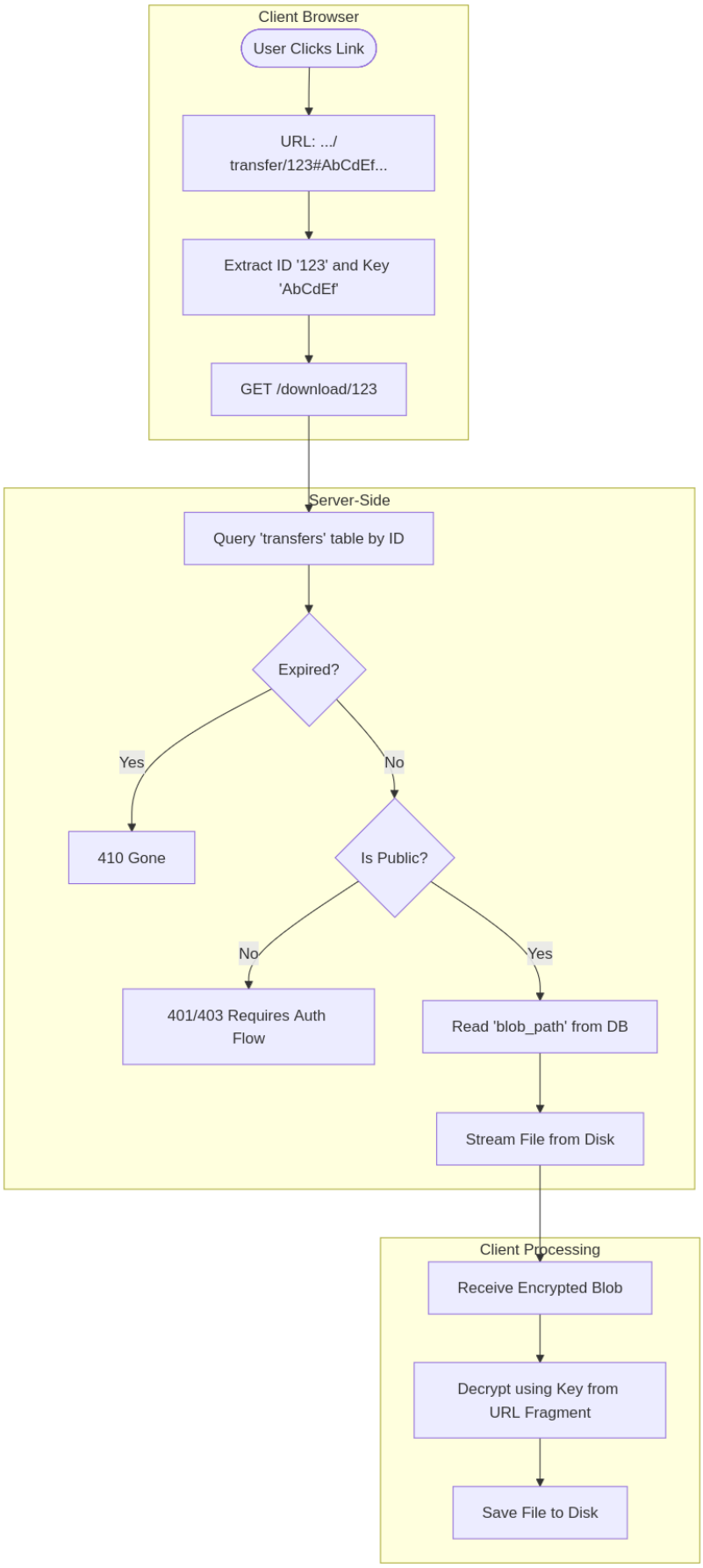

Figure 3.3. File Download Process for Public Shares

3.3 User Key Management

Each user in Secure Share possesses an RSA-4096 asymmetric key pair that serves as their cryptographic identity. The management of these keys presents a fundamental challenge: the keys must be available to the user across multiple devices, yet must remain confidential even if the server is compromised. Secure Share addresses this through a vault-based key management system.

3.3.1 Key Pair Generation

During account activation, the CLI client generates a fresh RSA-4096 key pair locally. The private key is exported in PKCS#8 format without encryption (encryption is applied separately through the vault mechanism, described next). The public key is exported in `SubjectPublicKeyInfo` format, ensuring compatibility with standard cryptographic tools.

3.3.2 Private Key Protection: The Vault System

To enable cross-device access while maintaining zero-knowledge security, the user's private key is encrypted with a password-derived key before being uploaded to the server. This encrypted blob is referred to as a "vault." The vault encryption process employs PBKDF2 for key derivation and AES-256-GCM for encryption.

The iteration count of 480,000 is set according to OWASP guidelines as of 2023. This high iteration count significantly slows down brute-force attacks. The random 128-bit salt ensures that identical passwords produce different derived keys, preventing rainbow table attacks and ensuring that compromising one user's vault does not aid in attacking others.

When a user needs to perform operations requiring their private key (such as decrypting received files), they fetch the vault from the server via `GET /users/me/vault`, enter their password, and decrypt the vault locally.

This design ensures that the server never has access to the user's private key in plaintext, maintaining the zero-knowledge property. Even if an attacker compromises the server and exfiltrates all vault data, they cannot decrypt the vaults without knowing each user's password.

3.3.3 Public Key Distribution

In contrast to private keys, public keys do not require confidentiality protection. Each user's public key is stored in plain text in the `users.public_key` column and is made available to all authenticated users via the `GET /users/{user_id}/key` endpoint. This enables the file sharing workflow: when Alice wants to share a file with Bob, her client fetches Bob's public key and uses it to encrypt a copy of the file key specifically for Bob.

3.4 Password Security

Secure Share employs distinct mechanisms for protecting two types of passwords: user authentication passwords and vault passwords. While these are often the same from the user's perspective, they are processed differently to optimize for their respective threat models.

3.4.1 Authentication Password Hashing with Argon2

User authentication passwords are hashed using Argon2id.

```
pwd_context = CryptContext(
    schemes=["argon2"],
    deprecated="auto",
    argon2__time_cost=2,
    argon2__memory_cost=102400,  # 100 MB
    argon2__parallelism=8
)
```

This algorithm combines resistance to both side-channel attacks and GPU-based cracking. The time cost of 2 iterations provides additional protection, and parallelism of 8 allows the algorithm to utilize multiple CPU cores for hashing while also increasing the cost for attackers.

Additionally, our app implements **a server-side pepper** stored in the environment configuration. The pepper is a secret value stored on the server (not in the database) that is concatenated with the password before hashing. This provides **defense in depth**, even if an attacker exfiltrates the database containing password hashes, they cannot perform offline cracking without also compromising the server's configuration to obtain the pepper value.

3.4.2 Vault Password Key Derivation with PBKDF2

For vault encryption, passwords are processed using PBKDF2-HMAC-SHA256 (described in section 3.3.2). While PBKDF2 is older than Argon2, it was chosen for vault key derivation due to its widespread support across cryptographic libraries and its suitability for client-side execution.

An important security consideration is that vault decryption happens on the client, where the derived key exists in memory, only temporarily. In contrast, authentication password verification happens on the server, where Argon2's memory-hardness provides stronger protection against parallel cracking attempts in the event of database compromise.

3.5 Digital Signatures and Token Integrity

All security tokens are signed using RS256. This is the same RSA-4096 key pair that users possess for encryption, leveraged for a dual purpose. The signature process involves the Security Officer (for MLS tokens) or Administrator (for RBAC tokens) signing the token payload with their private key.

3.5.1 Signature Algorithm: RS256

The signed token is then transmitted to the server, which stores it in the appropriate database table. When a user presents a token to access a protected resource, the server verifies the signature using the issuer's public key.

The use of the Key ID in the JWT header allows the verifier to identify which public key to use without trusting the payload itself. This prevents substitution attacks where an attacker might try to present a token signed by a different, less privileged user.

3.5.2 Token Revocation

To support token revocation, the system maintains a `jwt_revocation_tokens` table. During signature verification, the server checks whether the token's unique identifier, `jti` claim, appears in the revocation table.

Revocation records themselves are signed by the user performing the revocation, creating an auditable chain of custody for privilege changes.

3.6 Integrity Mechanisms

Beyond the authentication provided by AES-GCM for encrypted data and RS256 for tokens, Secure Share employs SHA-256 hashing to ensure the integrity of the audit log through a hash chain mechanism.

3.6.1 Audit Log Hash Chain

Each entry in the audit log contains a SHA-256 hash that is computed over the concatenation of the previous entry's hash and the current entry's data.

The first entry in the log uses a genesis hash of all zeros. Each subsequent entry cryptographically commits to the entire history of the log. Any attempt to modify a past entry will break the chain, as all subsequent hashes depend on it. This provides tamper evidence, as Auditors can recompute all hashes and detect any discrepancies.

This hash chain is computed entirely server-side, every time an event is logged. The use of SHA-256 provides collision resistance, an attacker cannot find two different log entries that produce the same hash.

3.7 Summary

This defense-in-depth approach ensures that compromise of any single component (server storage, database, network traffic) does not compromise the confidentiality of user files or the integrity of the security policy enforcement mechanisms.

Chapter 4

Authentication and Session Management

Authentication is Secure Share is implemented as a password-based, token-driven system that separates credential verification, session management and privilege presentation. User authentication is performed by the FastAPI backend and results in the issuance of short-lived signed session tokens. Elevated privileges and security context are conveyed via separate cryptographically-signed tokens. All token-bearing requests must be sent over the TLS-protected channel.

User passwords are protected in two complementary ways, as explained above (section 3.4). Account activation uses a one-time password (OTP) workflow, as showcased below.

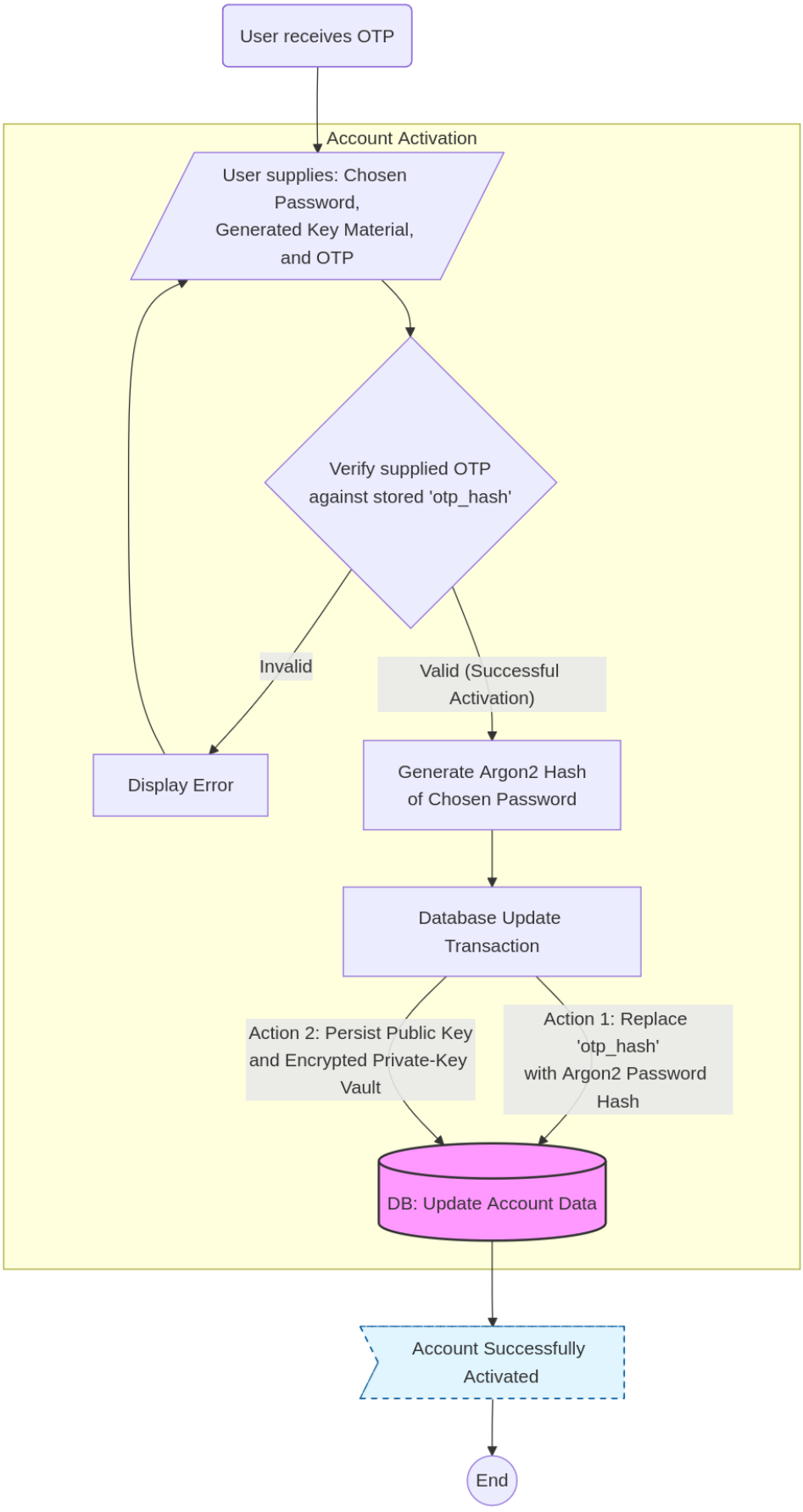

Figure 4.1. Account Activation Workflow

Session is based on JSON Web Tokens (JWTs) signed with RS256. The application is configured to use RS256 and session/auth JWTs are signed with the server's RSA private key and verified with the corresponding public key. The server signing key pair is supplied to the application as PEM-encoded strings through environment variables. The repository's setup script populates the .env file with these keys. Using asymmetric signatures for session tokens simplifies verification and aligns session verification with RBAC/MLS token verification.

When a user authenticates successfully, the backend issues a session JWT containing standard claims and any necessary session metadata. Session tokens are intentionally short-lived. Secure Share treats session tokens as stateful artifacts, where each issued JWT is recorded in the JWTAuthToken table. The token issuance logic first checks for an existing active, unexpired token and returns if present; otherwise it inserts a new token record. This approach enables explicit session life cycle operations such as logout.

Beyond session tokens, the system uses purpose specific signed tokens for conveying elevated authority and MLS context. Role-assignment tokens (RBAC tokens) and MLS clearance tokens are signed by the issuer's private RSA key and stored on the server. Token verification proceeds defensively: the server extracts the issuer identifier from the token header (kid), uses that to fetch the issuer's public key, verifies the RS256 signature, validates required claims and checks for revocation by consulting the JWTRevocationToken table. For RBAC and MLS workflows, the server also verifies token ownership before granting privilege-based operations.

On the client side, the CLI stores one session token locally at a time. When the CLI needs to perform cryptographic operations requiring a private key, it fetches the user's encrypted vault from the server and decrypts it locally using a password-derived key.

Revocation and auditability are integrated into the token life cycle, as well. Revocation records are stored in a table and checked during token verification. All actions that change token state are all logged via the hash-chain protected audit log so that the history of token events is auditable and tamper-evident.

Summary table of token types:

Token Type	**Signature Algorithm**	**Typical Lifetime**	**Storage**	**Presented In**
Session JWT (access)	RS256 (server key)	15 minutes (no refresh)	JWTAuthToken (DB)	Authorization: Bearer
MLS Clearance Token	RS256 (issuer key)	Long-lived (e.g., months–1 year)	jwt_mls_tokens (DB)	X-MLS-Token header
RBAC Role Token	RS256 (issuer key)	Long-lived (e.g., months–1 year)	jwt_rbac_tokens (DB)	X-Role-Token header
Revocation Record	N/A (signed by revoker)	N/A (persistent)	jwt_revocation_tokens (DB)	Checked server-side

Table 4.1. Token Specifications and Storage

Chapter 5

Role-Based Access Control

Secure Share implements an explicit, auditable RBAC model that separates assignment of privileges (role tokens) from their verification and enforcement. Roles are represented as canonical identifiers in the backend and as signed tokens that prove entitlement. Assignment and revocation operations are cryptographically protected and recorded in the audit log.

5.1 Roles and Responsibilities

This application defines five role identifiers in code: ADMINISTRATOR, SECURITY_OFFICER, TRUSTED_OFFICER, AUDITOR and STANDARD_USER. The CLI exposes a subset relevant to token-based assignment and checks (SECURITY_OFFICER, TRUSTED_OFFICER, AUDITOR). ADMINISTRATOR is represented by the is_admin account flag established at initialization and STANDARD_USER is the default role for ordinary accounts.

In policy terms:

- **ADMINISTRATOR**: Persistent built-in administrator account. An Administrator can create user, create departments and may call the role-assignment API to appoint Security Officers and Auditors. The implementation prevents changing the `is_admin` flag by API, so the Administrator cannot be demoted through normal endpoints.
- **SECURITY_OFFICER**: They are authorized to perform security policy management actions. Security Officers may issue RBAC tokens to appoint Trusted Officers. They are also required to sign MLS clearances, however, a SO cannot self-issue an MLS clearance for themselves.
- **TRUSTED_OFFICER**: This is an elevated role that, when accompanied by an explicit, signed role token and a justification, may bypass Bell-LaPadula MLS checks for read or write operations.
- **AUDITOR**: Read-only access to the audit log and ability to upload verification objects. Auditor status is verified via signed RBAC tokens.

- **STANDARD_USER**: Default user role, can upload or download public transfers and user-specific transfers per MLS constraints.

5.2 Role Assignment and Storage Model

Role assignments are represented and stored as signed JWTs (RBAC tokens). The role assignment endpoint is `PUT /users/{user_id}/role` and accepts a JSON payload containing the signed JWT token. Role tokens are created and signed on the client (issuer) using the issuer's private RSA key; the server receives the raw signed JWT and stores it intact in a JWTRBACToken record.

The JWT header includes a kid that identifies the issuer (the numeric user ID) and the server uses it to look up the issuer's public key in the users table and verifies the token signature with RS256.

5.3 Presentation and Verification

When a privileged action is requested, role tokens are presented in the X-Role-Token HTTP header. Dependency functions in the backend retrieve and verify the supplied RBAC token. This process includes:

1. Read unverified header
2. Extract kid (issuer id)
3. Fetch issuer public key from database
4. Decode and verify RS256 signature
5. Validate essential claims
6. Verify ownership
7. Check revocation state

If any of these steps fails, the request is rejected with HTTP 403/400 as appropriate and the failure is logged to the audit log.

5.4 Revocation and Token Life Cycle

Role token revocation is supported and enforced. Revocations are recorded in the JWTRevocationToken table keyed by the revoked token's jti plus token_type and are verified by fetching the issuer's public key to verify the signature. Since revocation is checked on each verification against the authoritative database, revocation takes effect immediately.

Chapter 6

Multi-Level Security (MLS)

Secure Share implements a lattice-based MLS model that combines hierarchical clearance level with non-hierarchical department categories. MLS enforcement is performed server-side before the backend releases any data. Clients must present a signed MLS clearance token (X-MLS-Token) when performing MLS-controlled operations. The design enforces the Bell-LaPadula confidentiality properties and provides auditable Trusted Officer override for exceptional circumstances.

6.1 Clearance Levels and Categories

Clearance levels are represented as strings in MLS tokens and are mapped to integer values for comparison on the server. The numerical mapping used by the service is:

- UNCLASSIFIED -> 1
- CONFIDENTIAL -> 2
- SECRET -> 3
- TOP_SECRET -> 4

Departments are plain string labels. Comparisons of department sets are exact, case-sensitive string matches.

6.2 MLS Token Format, Issuance and Storage

MLS clearances are conveyed in signed JWTs created client-side by a **Security Officer** and submitted to the server. Tokens are signed using RS256 with the issuer's private key. The server verifies signatures by fetching the issuer public key from the users table. The JWT header includes a kid that identifies the issuer. The system stores MLS tokens in the JWTMLSToken

table, however these tokens are revocalbe via entries in JWTRevocationToken with token_type = "MLS".

6.3 MLS Verification

When a request includes a X-MLS-Token, the server validates it by following the same steps described in section 5.3.

For user-specific share:

If a request omits X-MLS-Token, the business logic in `transfers/service.py` treats MLS payload absence as a failure for MLS-protected operations because the service requires either a valid MLS token or an appropriate Trusted Officer bypass.

6.4 Bell-LaPadula Rules

The server implements the two classic rules by mapping level strings to integers and performing set inclusion checks for departments.

- **Simple Security Property (No Read Up)**:

  $\texttt{Level(Subject)} \geq \texttt{Level(Object)}$ **AND** $\texttt{Departments(Subject)} \supseteq \texttt{Departments(Object)}$

- ***-Property (No Write Down)**:

  $\texttt{Level(Subject)} \leq \texttt{Level(Object)}$ **AND** $\texttt{Departments(Subject)} \subseteq \texttt{Departments(Object)}$

These checks are executed inside the MLS dependency functions by the transfers business logic. It requires MLS verification for the following endpoint (user-specific transfers):

- `POST /transfers` - creating/uploading a transfer checks the *-Property and enforces that user clearance permits the chosen classification and departments (or a Trusted Officer bypass is present). For public transfers (is_public = True) MLS checks are not required.
- `GET /transfers/{transfer_id}` - retrieving transfer metadata and the user's encrypted file checks the Simple Security Property for user-specific transfers.
- `GET /transfers/download/{transfer_id}` - Download of the encrypted blob and any per-user encrypted key checks the Simple Security Property for user-specific shares.

6.4.1 Public Shares

Transfers marked with is_public = True are treated as organization wide. The implementation treats this flag as a complete bypass of MLS checks.

6.4.2 User-specific Shares

When a transfer is shared in a user-specific way, the server stores the encrypted recipient file key but does not permit retrieval of those encrypted keys unless MLS checks succeed. In other words, having an encrypted file key stored for you does not grant the ability to download it. The server releases the encrypted copy of the file key only after it has verified the requester's MLS clearance (or accepted a valid Trusted Officer bypass). The decryption of the per-user encrypted key is always performed client-side with the user's private key.

6.5 Trusted Officer bypass

The Trusted Officer role permits an explicit override of both read and write MLS rules. To exercise a bypass the following are required on the request:

- X-Role-Token header with valid, unrevoked TRUSTED_OFFICER JWT that belongs to the caller.
- X-Justification header containing a non-empty textual justification

Server checks for the Trusted Office bypass are:

- Verify X-Role-Token as a TRUSTED_OFFICER.
- Confirm X-Justification is present and not empty

If both conditions are satisfied, skip MLS comparisons for this request and permit the action. Every use of the Trusted Officer bypass is recorded in the audit log.

6.6 Revocation and Token Life Cycle

MLS tokens are revocalbe. Revocations are recorded with token_type = "MLS" in the JWTRevocationToken table and are checked at every MLS token verification. The default MLS token TTL is 365 days, but revocation allows immediate invalidation prior to expiry.

6.7 Summary

MLS in Secure Share is a server-side gate: tokens are created and signed by Security Officers client-side, stored with raw signatures for auditability and verified and revoked by the server on each protected request. Trusted Officer overrides are possible but explicitly controlled with signed role tokens and mandatory justifications that are preserved in the audit hash chain.

Chapter 7

Hash Chain Auditing Mechanism

This chapter documents Secure Share's tamper-evident audit logging design. The implementation is centered in backend service with the AuditLog model.

7.1 Purpose and Security Objectives

The audit subsystem exists to provide an ordered, auditable record of security relevant events. Its goals are:

- **Tamper Evidence**: Any modification to prior entries should be detectable
- **Non-repudiation**: Link actions to actors and keep cryptographic evidence
- **Forensic Support**: Enable auditors to validate the integrity and chronology of events

7.2 Hash Chain Construction

The chain is built on SHA-256. The genesis record uses a fixed previous_hash of 64 zeros. For each new entry:

1. Retrieve previous_hash from the most recent audit_logs row (or genesis if none).
2. Compose the canonical string: previous_hash + timestamp + actor_id + action + details.
3. Compute current_hash = SHA256(canonical_string).hexdigest()
4. Insert the new audit_logs row

7.3 Verification by Auditors

An auditor verifies the chain by re-computing current_hash for each entry and confirming that each `entry.previous_hash` equals the prior `entry.current_hash`. Verification steps:

1. Fetch the full ordered audit log
2. For each entry:
    - Recompute expected_hash
    - Check expected_hash == entry.current_hash
    - Check entry.previous_hash == previous_entry.current_hash
3. Report any mismatch as tampering corruption.

The `GET /audit/log` endpoint returns the complete ordered list to authorized Auditors; the client (auditor) can perform the verification locally.

7.4 Access Control

Only users with the Auditor role (proven via a signed RBAC token in X-Role-Token and session JWT auth) are allowed to call `GET /audit/log/` or `PUT /audit/validate`. The returned log is read-only; auditors cannot alter existing entries.

Chapter 8

Security Decisions

This chapter documents security design decisions made during implementation that were not explicitly specified in the project guidelines, along with their rationale.

8.1 Self Token Issuance Prevention

Users cannot issue MLS clearance tokens or RBAC role tokens to themselves. This implements separation of duties and prevents privilege escalation scenarios.

- **MLS Tokens**: Security Officers cannot grant themselves clearances, requiring a second Security Officer to perform the assignment. This creates a two person rule that prevents a single compromised SO account from granting itself unlimited access to classified information.
- **RBAC Tokens**: Administrators and Security Officers cannot assign roles to themselves, preventing self-elevation of privileges and ensuring that role assignments involve multiple actors.

This design creates an audit trail involving multiple actors and aligns with the principle of least privilege and defense in depth.

8.2 Administrator Immutability

Administrator accounts cannot be deleted, have roles assigned to them or have clearance tokens. Prevents accidental or malicious removal of administrative access and maintains a clear separation between the built-in administrator role and token-based roles.

8.3 Private Key Handling: Zero-Knowledge Architecture

User private keys are never stored in plaintext on the server or persistently on the client. On the Client-side, the decrypted private key exists only in memory during active operation and is immediately discarded after use. It is never written to disk in plaintext.

This approach protects against:

- **Client-side forensics**: No plaintext key to recover from disk
- **Insider threats**: System administrators cannot access user private keys
- **Memory extraction attacks**: Minimizes the time window where plaintext keys exist in RAM

The CLI fetches the encrypted blob via API, decrypts it in memory using the vault password and returns a cryptography object that is used immediately for the current operation and then garbage-collected.

8.4 CLI Architecture

The CLI is implemented as a stateless command-line tool (like Git, AWs CLI) rather than a persistent daemon process running in the background.

The CLI process only runs during command execution and terminates immediately after, minimizing the time window for attacks:

- **Memory Dumping Attacks**: No persistent process to attach debuggers or memory dumpers to
- **Process Injection**: Attackers cannot inject code into a lon-running CLI daemon
- **RAM freeze/cold boot attacks**: Sensitive data exists in RAM only during command execution, not continuously

Each CLI invocation runs with minimal privileges for the specific operation, rather than a daemon holding elevated privileges continuously. Each command runs in a separate process with its own memory space, preventing cross-contamination between operations. To maintain usability without requiring re-authentication for every command, session tokens are persisted to disk (/.secureshare/session.json) with strict file permissions.

8.5 Password Peppering

A server-side pepper is appended to passwords before hashing with Argon2id. Provides defense in depth against database compromise. Even if an attacker steals the database with password

hashes, they cannot crack passwords without also obtaining the pepper from the .env file, which requires server file system access (see more about this on section 3.4.1).

Chapter 9

Static Code Security Analysis with SonarQube

Perform automated static code analysis using SonarQube to identify potential security vulnerabilities, code smells, and security hotspots.

The SecureShare project was analyzed using SonarQube Community Edition with the following security-focused rule sets:

- OWASP Top 10 2021
- OWASP Top 10 2017
- CWE

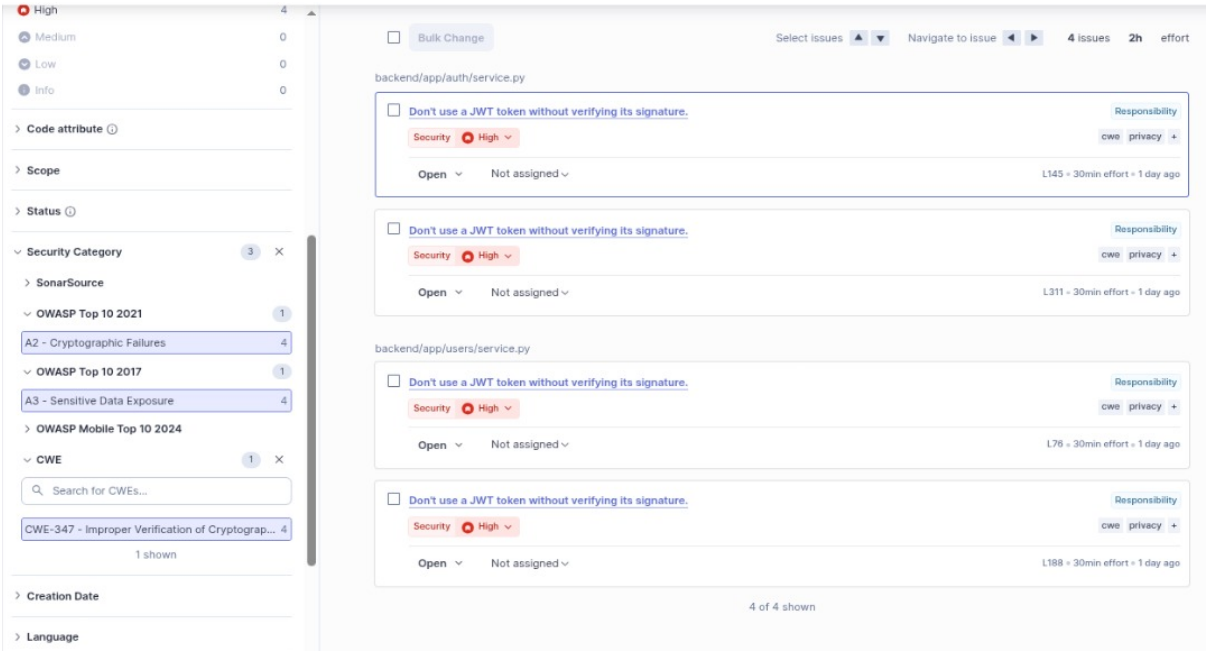

Figure 9.1. SonarQube First Analysis

SonarQube initially detected 4 instances where JWT tokens were being decoded without proper signature verification. This is a critical security vulnerability that could allow attackers to forge tokens. The implementation was reviewed and confirmed to be secure (see more about this on section 3.5.1).

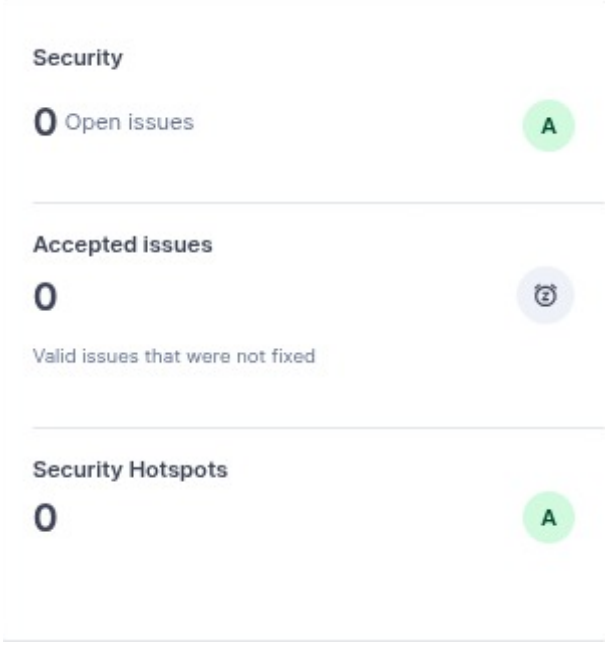

Figure 9.2. SonarQube After Solving Security Issues

The SonarQube analysis is integrated into the development workflow to ensure ongoing security validation. The resolution of all detected issues to achieve a Grade A security rating demonstrates the project's commitment to secure coding practices and thorough vulnerability remediation.

Chapter 10

Conclusion

This project successfully designed and implemented SecureShare, a secure file transfer system that demonstrates the practical application of cryptographic principles, formal access control models, and defense-in-depth security architecture. The system achieves its primary objective of enabling confidential file sharing within organizations while maintaining zero-knowledge guarantees with respect to the server infrastructure.

10.1 Key Achievements

The implementation delivers on all core security requirements:

End-to-End Encryption: All file content is encrypted client-side using AES-256-GCM before transmission, with hybrid encryption ensuring that symmetric file keys are protected using RSA-4096-OAEP. The server never has access to plaintext data, maintaining confidentiality even in the event of server compromise.

Zero-Knowledge Key Management: User private keys are protected through a vault-based system employing PBKDF2-HMAC-SHA256 with 480,000 iterations. The server stores only encrypted key material, ensuring that even database exfiltration does not compromise user cryptographic identities.

Formal Access Control: The system implements both Role-Based Access Control (RBAC) with five distinct roles and a lattice-based Multi-Level Security (MLS) model combining hierarchical clearance levels with non-hierarchical departmental categories. The Bell-LaPadula confidentiality properties (Simple Security and *-Property) are enforced server-side for all file operations.

Cryptographic Token System: All security tokens (session, MLS clearance, RBAC role) are signed using RS256, providing non-repudiable proof of authorization. Token revocation is supported through a centralized revocation table checked at every verification point.

Tamper-Evident Auditing: A SHA-256 hash chain protects the audit log, enabling independent verification by designated Auditors. Every security-relevant action is recorded with

cryptographic linkage to previous events.

Transport Security: All communications occur over TLS 1.3, with a custom certificate authority established for development and testing purposes.

10.2 Security Analysis and Validation

The implementation underwent static security analysis using SonarQube with OWASP Top 10 and CWE rule sets. Initial analysis detected four instances of potential JWT signature verification issues, which were reviewed and confirmed to follow secure patterns for multi-issuer JWT systems. The final security rating achieved Grade A with zero open security issues or hotspots. Several security decisions were made beyond the project requirements, including prevention of self-token issuance (enforcing separation of duties), administrator account immutability, password peppering for defense in depth, and a stateless CLI architecture that minimizes exposure to memory-based attacks.

10.3 Final Remarks

Secure Share demonstrates that strong cryptographic guarantees and formal security models can be implemented in practical systems without sacrificing usability. The zero-knowledge architecture ensures that users retain control over their data even when using centralized infrastructure. The combination of client-side encryption, cryptographically signed tokens, and tamper-evident audit logging provides multiple layers of defense against both external attackers and insider threats.

Bibliography

[1] OWASP Foundation, “Owasp top 10 - 2021,” 2021, accessed: 2024-12-07. [Online]. Available: https://owasp.org/Top10/

[2] MITRE Corporation, “2024 cwe top 25 most dangerous software weaknesses,” 2024, accessed: 2024-12-07. [Online]. Available: https://cwe.mitre.org/top25/

[3] E. Rescorla, “The transport layer security (tls) protocol version 1.3,” Internet Engineering Task Force (IETF), RFC 8446, 2018. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8446

[4] M. Jones, J. Bradley, and N. Sakimura, “Json web token (jwt),” Internet Engineering Task Force (IETF), RFC 7519, 2015. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7519

[5] S. Ramírez, “Fastapi: Modern, fast web framework for building apis with python,” 2024, version 0.100+. [Online]. Available: https://fastapi.tiangolo.com/

[6] SonarSource, “Sonarqube: Continuous code quality and security,” 2024, community Edition. [Online]. Available: https://www.sonarqube.org/