universidade de aveiro
deti
departamento de eletrónica, telecomunicações e informática

Licenciatura em Engenharia Informática
3º Ano
Ano Letivo 2025/2026

Personal Security Assessment

November 2025

Security of Information and Organizations

João Pereira **120010**
Diogo Duarte **120482**
Thiago Vicente **121497**

Contents

1 Introduction

This report presents ***SecureShare***, an end-to-end encrypted file sharing system. The project was developed to practice and apply concepts learned during both theoretical lectures and practical sessions of the **SIO** course, focusing on secure file exchange.

The system was designed with security as a priority, though it has some scalability limitations, which will be discussed later in the report.

In addition to the backend server, a command-line interface (*CLI*) was also developed to test and interact with the system's functionalities.

During development, AI-based tools were used as a supplementary aid for code exploration, debugging assistance, and documentation refinement. All architectural decisions and security mechanisms were designed and validated by the authors.

The report presents selected code snippets and concept explanations; the complete source code is available on GitHub.

2 Architecture and Technology Stack

2.1 Architecture

SecureShare is implemented as a layered backend system that follows a clear separation of concerns. The server is structured into three primary layers:

- **Routers**, Define HTTPS endpoints, handle request/response models, validate input, and delegate business logic to services.
- **Services**, Contain the application's core logic, authorization workflows, role-management procedures, and integrations.
- **Models**, Define database entities and relationships
- **Database**, relational database

This organization improves maintainability, encourages modularity, and allows each layer to evolve independently without affecting the others.

2.2 Technology stack

Technology	Purpose
FastAPI	High performance and easy to configure library used for routing
Pydantic	data validation
SQLAlchemy	ORM for database modeling and queries
Uvicorn	Run FastAPI application
Cryptography	Signatures and key handling
Bcrypt	Better hashing
Requests	Http(s) requests
Mkcert	Certificate creation
Docker	Containerization

Table 1: Tecnology choise

2.3 DB Implementation

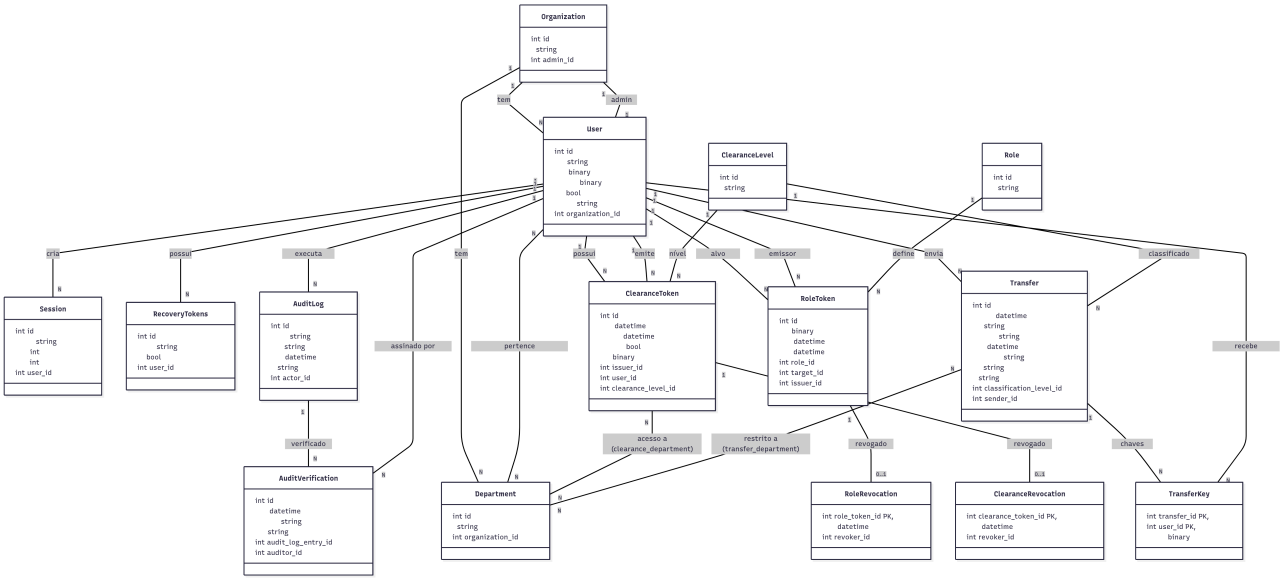

Figure 1: SecureShare's DB Schema

2.4 Endpoints

Figure 2: SecureShare's Endpoints Schema

3 Security Implementation

3.1 End-to-End Encryption

The SecureShare system was implemented with end-to-end encryption at it's core, and we can say it was sucessfull, as the files, their keys and the user's private key are never sent in plaintext (unless the transfer is explicitly public, but even then the server never gets the key, only the ones with the link). This makes our system very robust and secure for file sharing.

3.1.1 Client Side File Encryption

SecureShare implements client side file encryption with integrity control to ensure that any possible leaks on the server won't compromise the client's files. When sending the file the user can choose how the file will be encrypted with the current cyphers:

- **AES-GCM**
- **ChaCha20-Poly1305**

Both this methods are considered secure and robust,and both of them generate an authentication tag when encrypting, which means that any attempt at modifying the encrypted blob to modify the decrypted result won't work and the decryption will fail. These methods also have a feature that allows the system to add AssociatedData (will be refered as AAD) to the encryption. This AAD won't be encrypted and won't be included on the encrypted blob, it will be used to generate the authentication tag for the files. This means that the AAD is needed to decrypt the files, even a different bit on the AAD will make the decryption fail.

The workflow for the encryption goes like this (note that this entire workflow happens on client-side):

- User chooses the files that will sent, the encryption method, the security clearance, expiration-date, sets the transfer as public or not and sets the users that will have access (if not public)
- The files are zipped into a .zip file, this .zip file is the one that will be encrypted and sent to the server.
- A random symetric key and an Initialization Vector (also refered as nonce) are randomly generated
- Metadata for the transfer, (which includes the clearance level, the departments, expiration days, whether the transfer is public or not, users allowed, nonce and the encryption method) is generated
- Using the encryption method chosen by the user, the nonce and the symetric key randomly generated the files are encrypted. The metadata is also used in the encryption as the AAD, which means that the exact metadata that was sent is needed in order to decrypt the file. This means that if the metadata is tampered with, the file can't be accessed, even with the symetric key and the nonce.
- The metadata and the encrypted blob are sent to the server.

3.1.2 Key Encryption for User Specific Shares

SecureShare implements user specific shares. This works by encrypting the files as stated above and then use asymetric encryption to encrypt the symetric key with the recipients public key,ensuring that the symetric key can only be decrypted with the recipients private key, which ensures the only the correct users can decrypt the key and consequently the files.

The workflow for this type of shares goes like this(note that this entire workflow happens on client-side):

- User chooses the files and all the options shown in the previous encryption workflow and the users that will have access to the transfer
- For each user that will have access to the transfer, the system gets their public key from the server and encripts the symetric key with it
- The encrypted symetric key for each user is put on metadata
- Encryption occurs as explained in the previous workflow (Client Side File Encryption)
- The metadata and the encrypted blob are sent to the server.

3.1.3 Key handling for public shares

SecureShare implements public shares. In these shares, anyone with the transfer link can access it, but nevertheless the files content are still encrypted and the server can never access them. The file is accessible for anyone with the link because the key that allow for decryption is in the URL, but the process of putting the key in the URL is done on the client-side, ensuring that the server never sees the key and can never decrypt the file.

The workflow for this type of shares goes like this:

- User chooses the files and all the options shown in the Client Side File Encryption workflow and sets the transfer as public
- Encryption happens as stated in the Client Side File Encryption workflow and the files and their metadata are sent to the server
- Server responds with the transfer link for the file
- Client appends key to the URL
- Client shares the URL

3.1.4 Downloading transfer

SecureShare is a filesharing system built around security, and it has to implement a way of getting the transfers and downloading them. As all transfers are encrypted, we had to implement the decryption methods acording to the cyphers and the cypher modes we used.

For public transfer and User Specific shares the workflow is similar, the only thing that changes is how the system gets the symetric key to decrypt the files. For a public transfer, the user gets the transfer link, inserts it into the system and the system gets the key from the URL.

For a User Specific Share, the user puts the transfer id into the system, and the system will ask the server for the transfer. The server will provide the transfer and the encrypted symetric key. The symetric key will then be decrypted with the user's private key.

The workflow for downloading a transfer will go like this:

- User puts transfer (ID for user specific or URL for public transfer) into the Client
- Client will download the encrypted blob and metadata from the Server
- Client will get the symetric key as explained above
- From metadata, Client will get the IV and the correct cypher and cypher mode
- Client decrypt the file using the correct method and checking integrity also with the metadata
- Client will write the decrypted zip into a file

Important Note : The output file will have the name the user chooses or the name will be the file UUID. To check the file, the user will then have to rename it to a .zip file and then unzip.

3.1.5 Encrypted storage on server

SecureShare is implemented in a way that never allows the server to have access to the files uploaded to it. The client always sends the files encrypted and never sends the key to the server. This along with the private key also never being sent in plaintext to the server, makes it impossible for the server to access the user's uploads. The encrypted files are stored in the server machine and the metadata along with the encrypted blob path are stored in a Database in the server. The security methods implement ensure that even if an attacker gains access to the server, it's impossible to access uploads, and the integrity control applied to the encryption makes it useless to tamper with the encrypted files or their metadata.

3.2 Authentication and Key Management

The system implements a robust authentication mechanism combined with decentralized cryptographic key management.

3.2.1 Session Management

Authentication relies on stateful sessions backed by the database. The login process follows these steps:

1. The user provides credentials (username and password).
2. The server verifies the password using **bcrypt** (with a per-user salt) against the stored hash.
3. Upon success, a cryptographically secure random session token is generated using the `secrets` module, producing a 64-byte URL-safe string.
4. This token is hashed and stored in the **sessions** table with an expiration time of one hour.
5. The token is returned to the client and must be included in the `Authorization` header (Bearer scheme) for all subsequent protected requests.

Session validation occurs on every request via dependency injection using `Depends(get_db)`, verifying token existence and expiration.

3.2.2 Key Pair Generation and Storage

SecureShare uses asymmetric cryptography (RSA-4096) for identity and file encryption. A critical design choice was to perform all private key operations on the client side, ensuring the server never has access to unencrypted private keys.

Activation and Key Generation Flow:

- During account activation (via the `/activate` endpoint), the client generates an RSA-4096 key pair locally.
- The **public key** is sent to the server in PEM format and stored in the `users` table. It is publicly accessible and used by other users to encrypt shared files through a digital envelope mechanism.
- The **private key** is encrypted locally by the client using the user's password (e.g., via AES-GCM with a password-based key derivation function) before being sent to the server.
- This encrypted blob (`private_key_blob`) functions as a "cloud vault", allowing users to recover their keys on different devices without the server ever being able to decrypt them.

This hybrid approach balances strong security guarantees (zero-knowledge server model) with usability through secure key portability across devices.

4 Access Control Mechanisms

4.1 Role-Based Access Control (RBAC)

4.1.1 Overview

SecureShare implements a role-based access control system where users can hold multiple roles simultaneously but must specify which role they are acting as for each operation. Every user has the *Standard User* role by default and may be assigned one additional privileged.

Roles can be assign and revoked. For each assign, the issuer must sign the data, that signature is then validated on the server before applying changes.

The system enforces role-based permissions through the `X-Acting-Role` HTTP header, which specifies the role context for each request.

4.1.2 Role Descriptions

- **Administrator**: Manages the organization, departments and users, and assigns Security Officers.
- **Security Officer**: Issues and revokes clearances, assigns Trusted Officers and Auditors.
- **Trusted Officer**: Can bypass MLS restrictions with logged justification.
- **Auditor**: Access and validate the audit log, add verification objects.
- **Standard User**: Can upload and download files with MLS enforcement.

4.1.3 Permission Matrix

Administrative Operations

Operation	Admin	Sec. Officer	Trusted Off.	Auditor	Std. User
Create Department	✓	✗	✗	✗	✗
Delete Department	✓	✗	✗	✗	✗
List Departments	✓	✓	✓	✓	✓
Create User	✓	✗	✗	✗	✗
Delete User	✓	✗	✗	✗	✗
List Users	✓	✓	✗	✗	✗
Assign Security Officer	✓	✗	✗	✗	✗
Assign Trusted Officer	✗	✓	✗	✗	✗
Assign Auditor	✗	✓	✗	✗	✗
Revoke Role	✓	✓	✗	✗	✗
Assign Clearance	✗	✓	✗	✗	✗
Revoke Clearance	✗	✓	✗	✗	✗

Table 2: Administrative operations permissions

Security-Critical Operations

Operation	Admin	Sec. Officer	Trusted Off.	Auditor	Std. User
View Audit Log	✗	✗	✗	✓	✗
Verify Audit Chain	✗	✗	✗	✓	✗
Add Verification	✗	✗	✗	✓	✗
Upload	✗	✗	✓	✗	✓
Download	✗	✗	✓	✗	✓
Bypass MLS	✗	✗	✓	✗	✗
Delete Own Transfer	✗	✗	✗	✗	✓

Table 3: Security-critical operations

4.1.4 Implementation

The RBAC system is enforced through a `require_role()` function that acts as a FastAPI dependency. This function validates both user authentication and role authorization for each endpoint.

Role Enforcement Function:

```
def require_role(required_roles: list[str]):
    def role_checker(
        authorization: str = Header(...),
        x_acting_role: Optional[str] = Header(None),
        db: Session = Depends(get_db)
    ):
        # 1. Extract and validate session token
        token = authorization.replace("Bearer ", "")
        session = db.query(SessionModel).filter(
            SessionModel.session_token == token
        ).first()

        if not session or session.expires_at < time.time():
            raise HTTPException(401, ...)

        # 2. Get user and acting role
        user = db.query(User).filter(User.id ==
           session.user_id).first()
        acting_role = x_acting_role or "Standard User"

        # 3. Verify user has the claimed role
        user_roles = get_active_user_roles(db, user.id)
        if acting_role not in user_roles:
            raise HTTPException(403, ...)
```

```
        # 4. Verify role is authorized for this endpoint
        if acting_role not in required_roles:
            raise HTTPException(403,...)

        return db

    return role_checker
```

Usage Example:

```
@router.put("/{user_id}/clearance")
async def add_user_clearance(
    ...
    db: Session = Depends(require_role(["Security Officer"]))
):
    # Only accessible when acting as Security Officer
    ...
```

In the cases where multiple roles can access the separation of what they can do was done using *if statements*:

```
...
@router.get("/public/{access_token}")
async def download_public_transfer(
  ...
):
  trusted = is_trusted_officer(db, user.id, acting_role)
  if not trusted:
      # enforce MLS
      ...
  result = TransferService.get_transfer_file(db, transfer.id)
  # log and return file
  ...
```

4.2 Multi-Level Security (MLS)

4.2.1 Overview

SecureShare follows Bell-LaPadula model. Two rules are then enforced:

- **No read up**, users cannot access files from levels above
- **No write down**, users cannot write information to lower levels

In this implementation, users aren't associated to a single department. The association between departments and users is done using clearances.

A clearance can be applied to a set of departments or to the entire organization, and the are cryptografically signed by the issuer. The signature, as with roles, is verified before applying action.

Similar to roles, users can have multiple clearances and need to specify the clearance they want to use. This is enforced through the `X-Acting-Clearance HTTP` header.

4.2.2 Implementation

Two function were created to apply the rules of the Bell-LaPadula model:

```
def can_read(...) -> bool:
    ...
    # No Read up: can only read at or bellow clearance level
    if user_level_value < object_level_value:
        return False

    if not object_depts.issubset(user_depts):
        return False

    return True

def can_write(...) -> bool:
    ...
    # No Write Down: can only write at or above clearance level
    if user_level_value > object_level_value:
        return False

    if not object_depts.issubset(user_depts):
        return False

    return True
```

5 Audit System

The SecureShare Audit Log module is the cornerstone of the system's security architecture, designed to ensure **forensic readiness**, **immutability**, and **non-repudiation** of all sensitive operations. By implementing a secure ledger inspired by blockchain principles, the system ensures that any retrospective alteration of history becomes computationally evident and immediately detectable.

5.1 Cryptographic Hash Chain Architecture

The core of the audit system relies on a *Hash Chain* data structure stored within the `audit_log` relation. This structure guarantees that the integrity of any given record is mathematically dependent on the integrity of its predecessor, creating a continuous chain of trust.

5.1.1 Chain Initialization (Genesis Block)

The chain begins with a designated **Genesis Block** ($i = 1$). Since it has no predecessor, its 'previousHash' field is initialized with a standard null value (a string of 64 zeros). Figure 3 visualizes how the second block cryptographically binds itself to this genesis block.

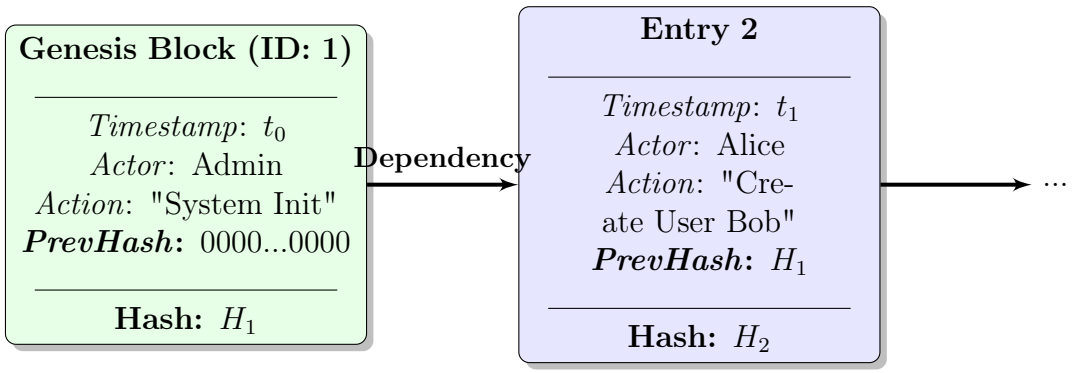

Figure 3: Initialization of the Hash Chain. Entry 2 explicitly includes H_1 in its payload calculations, creating an unbreakable link.

5.1.2 Hashing Algorithm

Each audit entry E_i is composed of a payload (timestamp, actor, action, details) and cryptographic metadata. The integrity is enforced using the **SHA-256** hashing algorithm.

The calculation of the hash for Entry 2 (H_2) is derived as follows:

$$H_2 = \text{SHA-256}\left(\text{JSON}\left(\{\text{timestamp} : t_1, \text{actor} : ..., \text{action} : ..., \textbf{previous\_hash} : H_1\}\right)\right) \quad (1)$$

By embedding H_1 directly into the data payload of Entry 2, the system ensures that any modification to the Genesis Block (which would change H_1) would immediately invalidate Entry 2, as its stored `previous_hash` would no longer match the recomputed SHA-256 hash of the Genesis Block.

5.2 Verification & Validation Mechanisms

To operationalize integrity checks, the system provides two distinct verification modes and a sophisticated temporal authorization check.

5.2.1 Verification Modes

1. **Incremental Verification (Signed Checkpoints) - `/audit/validate`:** This endpoint allows official Auditors to "seal" the log's current state. It performs the following operations:
   - Verifies the integrity of all entries generated since the last checkpoint.
   - Generates a digital signature (S) over the latest entry's hash (H_{last}) using the Auditor's RSA private key.
   - Stores the signature in the `AuditVerification` table.

   This effectively "freezes" history. In future audits, the system only needs to verify the cryptographic link back to this signed checkpoint, significantly optimizing performance for large datasets.

2. **Full Verification (Deep Audit) - `/audit/verify`:** This mode ignores all checkpoints and recalculates the hash chain from the Genesis Block ($ID = 1$) to the present. It is essential for disaster recovery and for detecting *post-signing tampering* (e.g., if an attacker modifies an old record that had already been signed in a previous year).

5.2.2 Temporal Semantic Validation

Integrity checks prove that the *bytes* are unchanged. However, SecureShare also enforces **semantic validity** — verifying if an action was legitimate **at the specific moment it occurred**.

The system reconstructs the authorization state at time t_{action} by checking:

- **Role History:** Did the actor hold a valid `RoleToken` at t_{action}?
- **Revocations:** Was a revocation issued *before* t_{action}?

5.3 Testing and Security Capability Verification

To rigorously validate these security guarantees, a specialized testing framework was developed. This framework bypasses the application layer and interacts directly with the database to simulate sophisticated insider attacks.

5.3.1 Test Environment Automation

The script `setup_test_env.sh` is used to bootstrap a consistent forensic environment. It automates:

- Organization and Department creation.
- Provisioning of actors (Admin, Auditors, Users) with cryptographic keypairs.
- Generation of baseline audit traffic and establishment of an initial signed checkpoint.

5.3.2 Tampering Simulation Scenarios

The `test_tampering_scenario.sh` script executes specific attack vectors documented in `adulteration_tests.md`. These tests verify the system's ability to detect the three primary categories of ledger compromise:

Scenario A: Content Integrity Violation An attacker modifies the payload of an event (e.g., changing the action description).

```
UPDATE audit_log SET action = 'Unauthorized Access' WHERE id = 5;
```

Result: The calculated hash H'_5 differs from the stored `entryHash`, triggering a *Content Modification Alert*.

Scenario B: Chain Continuity Attack An attacker calculates a new valid hash for a tampered entry and updates the database, attempting to hide the modification.

```
UPDATE audit_log SET entryHash = 'NEW_VALID_HASH' WHERE id = 5;
```

Result: The link to the next block is broken. Block 6 expects specific content in Block 5 to satisfy the condition $previousHash_6 \equiv entryHash_5$. The system flags a *Chain Broken Alert* at index 6.

Scenario C: Record Suppression (Deletion) An attacker deletes a compromising record entirely.

```
DELETE FROM audit_log WHERE id = 5;
```

Result: The verifier identifies a gap in the sequence. Block 6 references a predecessor that no longer exists in the relational store, triggering a *Missing Entry Alert*.

5.3.3 Summary of Detection Capabilities

Table 4 summarizes the executed test scenarios and results.

Table 4: Tampering Scenarios and Forensic Detection Mechanisms

Attack Vector	Simulation Method	Detection Mechanism
Content Integrity	SQL Update of `action`	Hash Mismatch ($H'_i \neq H_i$)
Chain Continuity	SQL Update of `entryHash`	Broken Link ($PrevHash_{i+1} \neq H'_i$)
Suppression	SQL `DELETE`	Sequence Gap (Reference not found)

5.4 Conclusion

The SecureShare Audit Log system successfully implements a forensic-grade logging mechanism. By combining a **SHA-256 Hash Chain** for data integrity with **RSA Digital Signatures** for checkpoint validation, the system achieves a high degree of resistance against modification and repudiation.

While no system can prevent all forms of physical or root-level compromise, the implemented solution ensures that any such breach leaves an undeniable cryptographic trace, fulfilling the requirement for a tamper-evident system.

6 Conclusion

The SecureShare project successfully achieved its main objective of designing and implementing a secure, end-to-end encrypted file sharing system, with a strong focus on confidentiality, integrity, and access control. Throughout the development process, the system incorporated modern cryptographic primitives and security design principles, ensuring that sensitive data remains protected even in the event of server compromise.

The implementation of client-side encryption, hybrid cryptography, role-based access control (RBAC), and multi-level security (MLS) demonstrated the practical application of theoretical security models in a real-world context. Additionally, the audit logging system, based on cryptographic hash chains and digital signatures, provided a robust and tamper-evident mechanism for accountability and forensic analysis.

Although the system presents some scalability limitations due to its security-first architecture, these design choices were deliberate and aligned with the project's academic and technical goals. The modular architecture, clear separation of responsibilities, and use of well-established technologies contributed to a maintainable and extensible codebase.

Overall, SecureShare proved to be a valuable learning experience, allowing the team to deepen their understanding of applied cryptography, secure system design, and organizational security principles. The project demonstrates that it is possible to combine usability and strong security guarantees in a practical file-sharing solution, laying a solid foundation for future improvements and real-world adaptations.

A PKI Certificate Validation

A.1 Overview

The TLS implementation was improved to use a hierarchical PKI (Public Key Infrastructure) instead of self-signed certificates. This change provides proper certificate chain validation and protection against Man-in-the-Middle (MITM) attacks.

A.2 Previous Implementation

The original implementation used self-signed certificates generated with `mkcert`:

```
# Client (api_client.py) - INSECURE
self.session.verify = False
urllib3.disable_warnings()
```

This approach had several security issues. There was no chain of trust validation, the client accepted any certificate including those from attackers, and as a result the system was vulnerable to man-in-the-middle attacks.

A.3 Current Implementation

Multi-level PKI hierarchy was implemented using XCA:

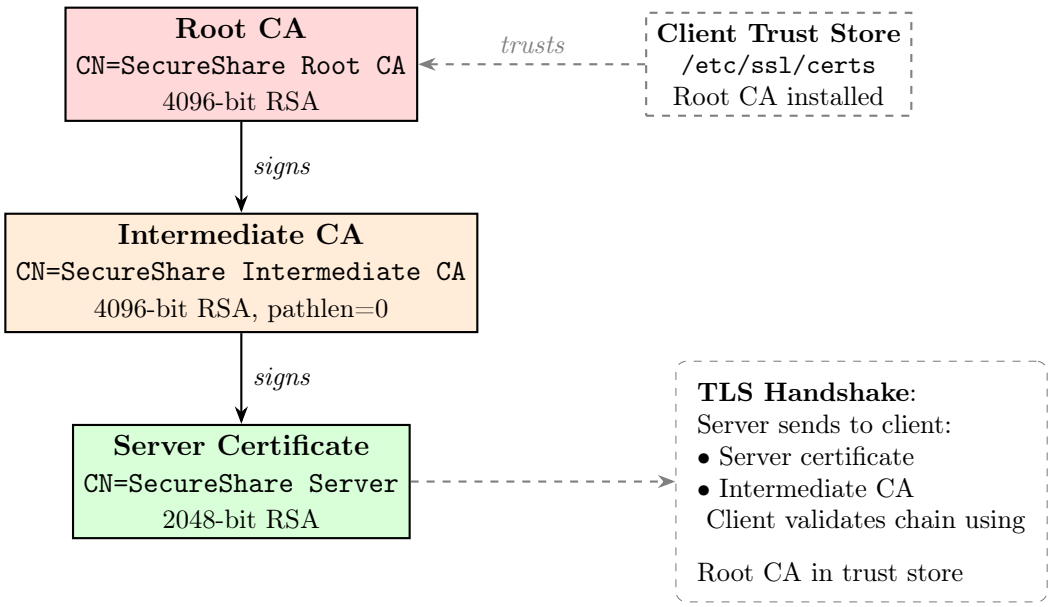

Figure 4: SecureShare hierarchical Public Key Infrastructure (PKI) architecture. The Root CA certificate is installed in the client's system trust store, enabling validation of the complete certificate chain during TLS handshake.

A.3.1 Certificate Chain Validation

The client validates the server's certificate chain against its system trust store. This is achieved through two complementary mechanisms:

1. Python requests configuration (`client/api_client.py`):

```
class APIClient:
  def init(self, base_url: str, ...):
      self.session = requests.Session()
      # Enable certificate validation using system trust store
      self.session.verify = True
```

2. System trust store installation (`client/Dockerfile`):

```
# Install Root CA in system trust store
COPY certs/root.crt
   /usr/local/share/ca-certificates/secureshare-root.crt
RUN update-ca-certificates

# Explicitly configure Python to use system CA bundle
ENV SSL_CERT_FILE=/etc/ssl/certs/ca-certificates.crt
ENV REQUESTS_CA_BUNDLE=/etc/ssl/certs/ca-certificates.crt
```

The `verify=True` directive instructs the `requests` library to validate the server's certificate against the trusted CA certificates. The environment variables ensure that Python's SSL module correctly locates the system CA bundle at `/etc/ssl/certs/ca-certificates.crt`, which is generated by `update-ca-certificates` during container build.

A.3.2 TLS Handshake and Certificate Validation

During the TLS handshake, the server transmits its certificate chain consisting of `server.crt` and `intermediate.crt`. The client validates this chain by confirming that: (1) the server certificate is signed by the Intermediate CA, (2) the Intermediate CA is signed by the Root CA, and (3) the Root CA is present in the system trust store. Only upon successful validation of all three conditions is the secure connection established.

A.4 Security Improvements

Aspect	Before	After
Certificate type	Self-signed	CA-signed
Verification	`verify=False`	`verify=True`
MITM protection	None	Full
Trust hierarchy	None	Root → Intermediate → Server

Table 5: Comparison of TLS implementations

A.5 Testing

Certificate chain validation can be verified using OpenSSL. The `-untrusted` flag provides the intermediate certificates from `chain.crt`:

```
$ openssl verify -CAfile client/certs/root.crt \
    -untrusted server/certs/chain.crt server/certs/chain.crt
server/certs/chain.crt: OK
```

Fake certificates are rejected:

```
$ openssl verify -CAfile client/certs/root.crt /tmp/fake.crt
error 18 at 0 depth lookup: self-signed certificate
```

A.6 Certificate Generation with XCA

The certificates were generated using XCA, a graphical certificate management tool. The following steps describe the creation process.

A.6.1 Step 1: Create Root CA

Open XCA and create a new database. Click on `New Certificate` and configure as follows:

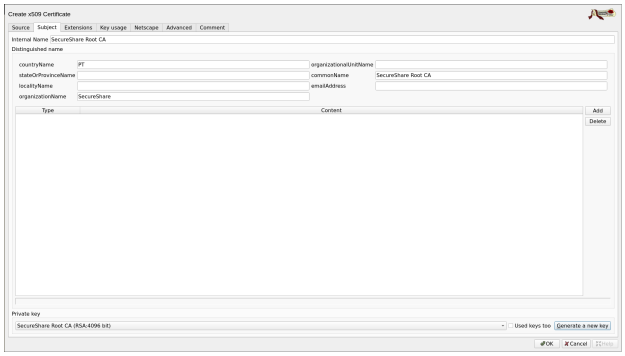

Figure 5: Root CA - Source tab: Select "Create a self signed certificate"

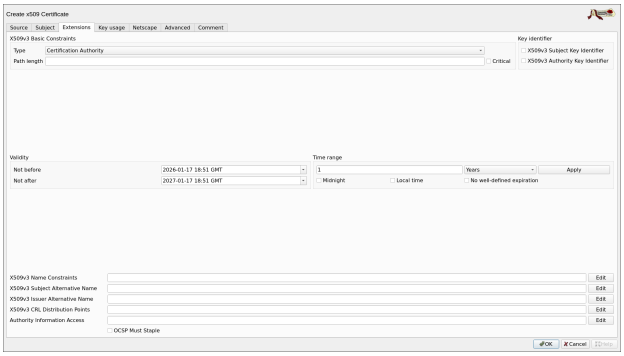

Figure 6: Root CA - Subject tab: Set Common Name to "SecureShare Root CA"

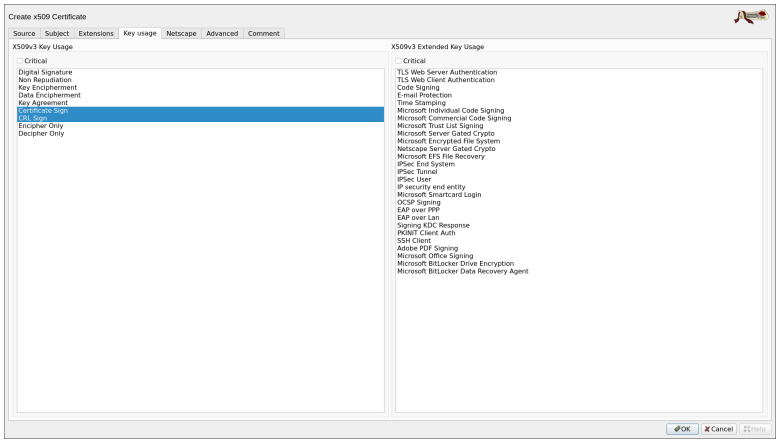

Figure 7: Root CA - Key Usage: Enable Certificate Sign and CRL Sign only

A.6.2 Step 2: Create Intermediate CA

Click on `New Certificate` and configure the Intermediate CA:

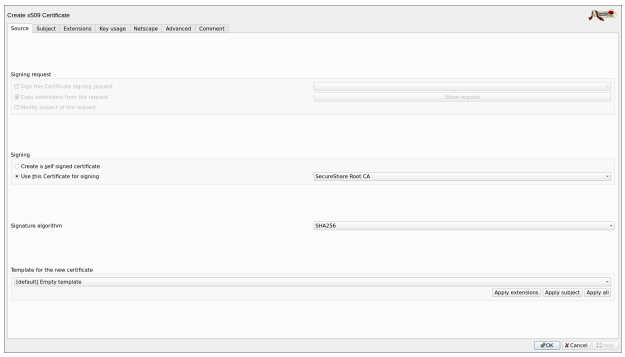

Figure 8: Intermediate CA - Source: Sign with Root CA

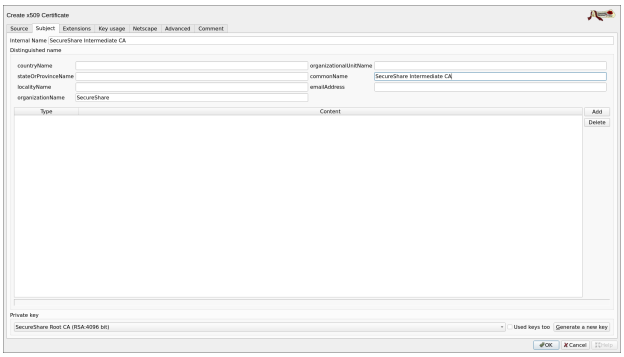

Figure 9: Intermediate CA - Subject: Set CN to “SecureShare Intermediate CA”

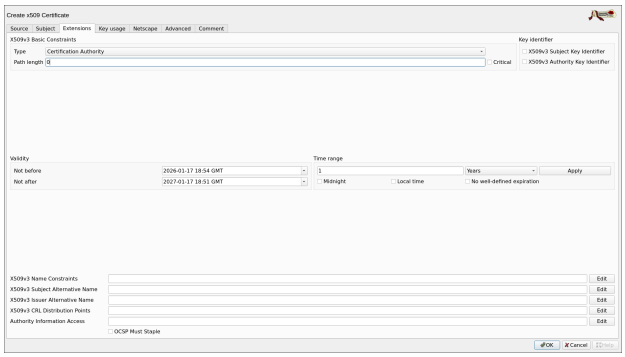

Figure 10: Intermediate CA - Extensions: Set as CA with path length 0

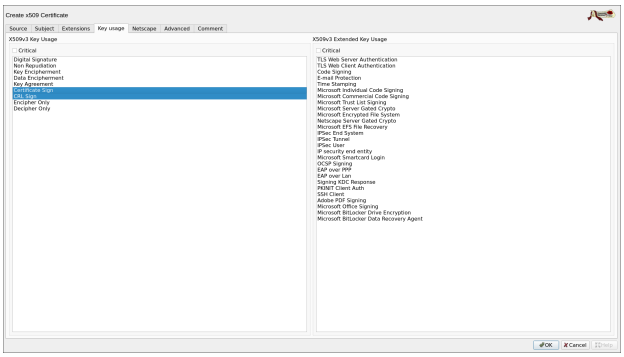

Figure 11: Intermediate CA - Key Usage configuration

A.6.3 Step 3: Create Server Certificate

Click on `New Certificate` and configure the Server Certificate:

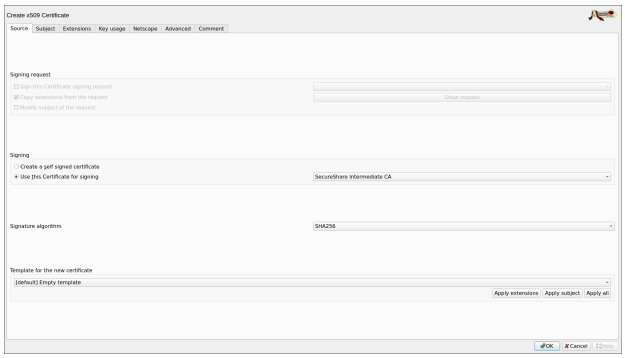

Figure 12: Server Certificate - Source: Sign with Intermediate CA

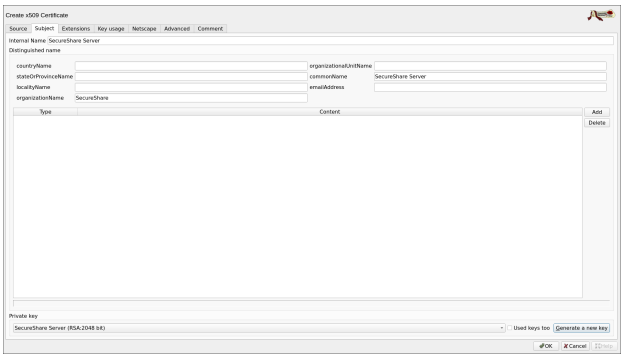

Figure 13: Server Certificate - Subject: Set CN to “localhost”

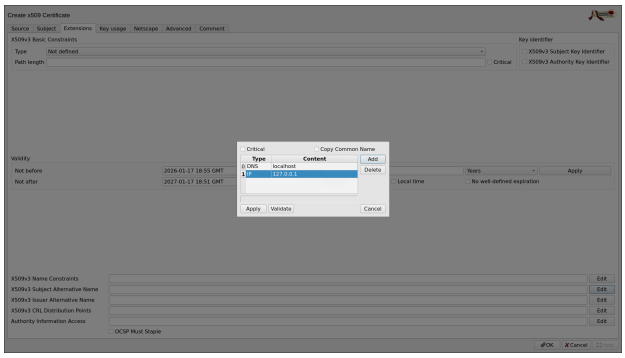

Figure 14: Server Certificate - Extensions: Set as End Entity

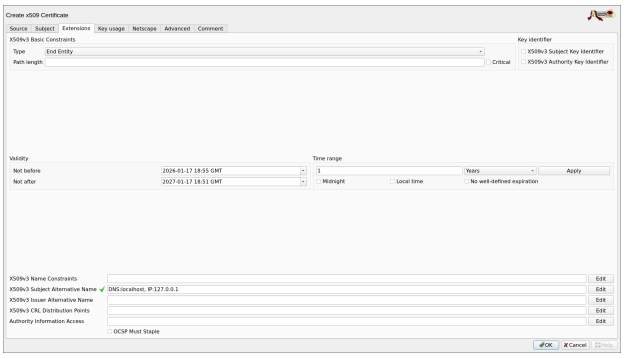

Figure 15: Server Certificate - Subject Alternative Names: Add localhost and 127.0.0.1

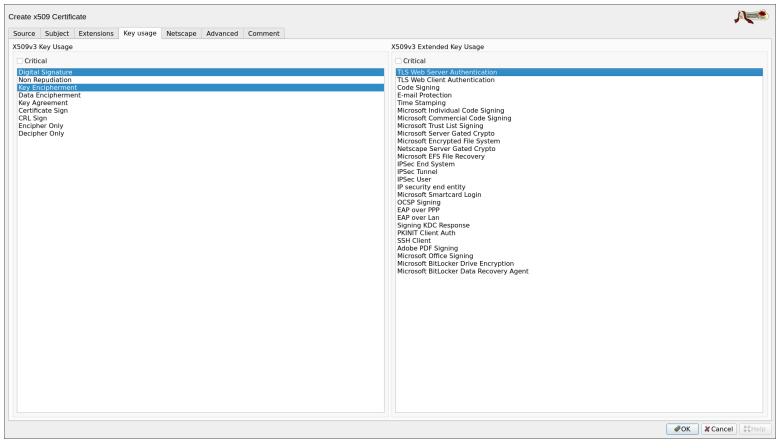

Figure 16: Server Certificate - Key Usage: Digital Signature and Key Encipherment

A.6.4 Step 4: Export Certificates

Export the certificates to the appropriate locations:

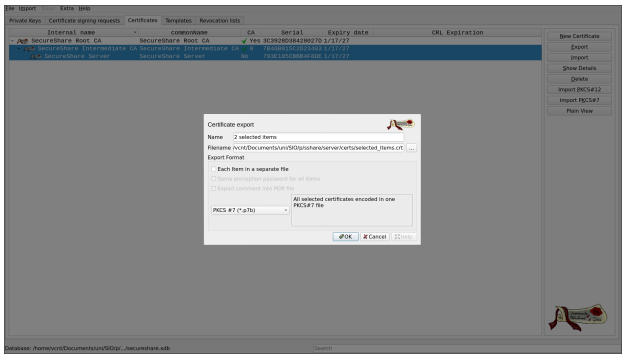

Figure 17: Export server certificate chain to `server/certs/chain.crt`

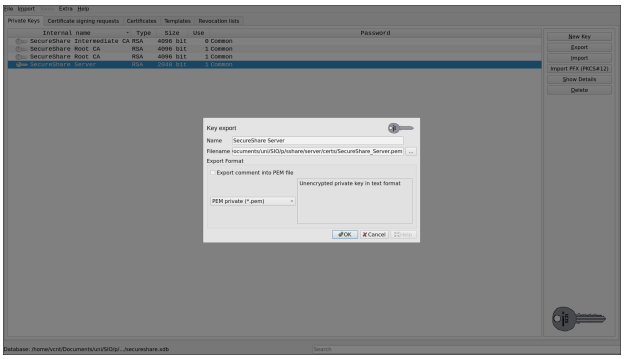

Figure 18: Export server private key to `server/certs/key.pem`

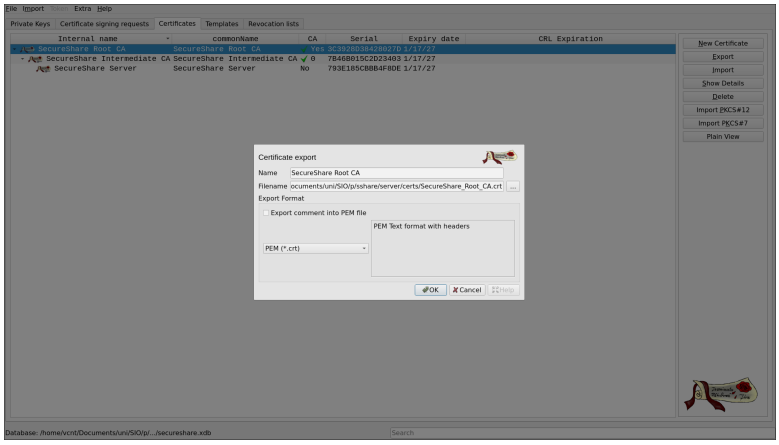

Figure 19: Export root certificate to `client/certs/root.crt`