sweet.ua.pt

Project 2 - SecureShare | João Paulo Barraca

João Paulo Barraca

15–19 minutes

Project Description

The goal of this project is to design and develop **SecureShare**, a single-tenant secure file transfer web application. This application is intended as a practical assignment for the course. The core focus is on implementing robust security features, including mandatory end-to-end encrypted storage, role-based access control (RBAC), and a formal multi-level security (MLS) scheme, which incorporates departments as security categories (labels).

The project must be developed by a group of **3 students**.

Core Functionality

The platform's primary function is to allow authenticated users to upload files and share them via a unique link, with all operations governed by strict security policies.

- **Organization Management:** The organization is managed by a Administrator, and is composed by users and departments

- The creation of a system implies the creation of a Administrator, than can nominate Security Officers.
- The Administator cannot be modified.
- Administrator can create Departments that represent groups of users.
- Every interaction is logged, in a structure using a hash chain.
- **User Management:** Users interact with the system using a secure (TLS) connection and obeying the clearance they have, according to the following rules.
- Users can be activated without having a clearance. Activation implies providing basic information, defining authentication data and creating key pairs.
- Users store an encrypted blob at the server with their keys. The encryption key is never provided to the server, but users can obtain the blob to interact across devices.
- All public keys are available to all users.
- The Security Officer can issue clearances to users, in the form of time limited, signed object with attributes specifying the clearance level for the entire organization, and for specific departments (a Label).
- Users may provide a clearance object to select the role to be activated.
- Roles are similar to clearances, represented in a similar manner, but related to management actions.
- Roles can be revoked. Revocation consists on creating a Role Revocation object, which is stored at the server.

- **File Upload:** Authenticated users can upload one or more files as a single “transfer.” The uploader must assign a security classification (level and optionally a department) to the transfer. The available levels and departments are constrained by the user’s own security clearance. Transfers can belong to the organization (have no department). All encryption happens client-side before upload.
- **Link Generation:** At request be the user, the system generates a unique, non-guessable URL.
- For **public shares**, the link, created by the client, will include the symmetric decryption key in the URL fragment (#key), which is never sent to the server.
- For **user-specific shares**, the link acts as a pointer to the transfer, with access controlled by the server, based on the MLS system. Users specify who can access the file by encrypting the file key with the target user public key.
- **File Download:** Any individual with a link can attempt to access the resource.
- For **public shares**, the client fetches the encrypted file from the server, reads the key from the URL fragment, and performs decryption in the client side.
- For **user-specific shares**, an authenticated user’s client will fetch the encrypted file and their individually encrypted file key. Decryption is performed client-side using the user’s private key.
- Access by authenticated users is always subject to MLS policy checks before the server releases any data. Therefore, users

may select the adequate role or clearance and send the adequate information in the request.

- **Transfer Expiration:** All transfers must have a configurable expiration time (e.g., 7 days), after which the associated files are permanently and securely deleted from the server.
- **Audit log Access:** Auditors can access the interaction log to validate it.
- The log is only available to Auditors.
- the log is read only.
- Auditors read the log and check the hash chain securing the events.
- Auditors may upload a Verification Object to the log, composed by a document stating the Auditor ID, timestamp, and a signature of the previous entry.

Organizations and Users

The system operates as a single instance with internal segmentation provided by departments.

- **Departments:** The Administrator can create and manage a list of departments (e.g., 'Finance', 'HR', 'Engineering'). These departments function as security labels or categories in the access control model.
- **User Management:** The Administrator can manage, and remove users from the system, while the Security Officer can issue and revoke clearances.

Role-Based Access Control (RBAC)

The system will implement a clear hierarchy of user roles, each with a specific set of permissions.

Role	Description
Administrator	A Role (`ADMINISTRATOR`) that manages the entire application. Appoints Security Officers and creates Departments.
Security Officer	A specialized role (`SECURITY_OFFICER`) responsible for defining security policies and **issuing signed security attribute tokens** (containing clearance levels, departments, and integrity levels) to users. Can appoint Trusted Officers.
Trusted Officer	An elevated role (`TRUSTED_OFFICER`) that can bypass the Bell-LaPadula models for read/write operations when providing a valid justification, which must be logged for audit purposes.
Standard User	The default role (`USER`) for users. Can upload and download files in accordance with the security policies.
Auditor	A role (`AUDITOR`) that can check and validate the interaction log.

Security Requirements

This section outlines the critical security implementations required for the project.

End-to-End Encryption and Storage

All files must be encrypted on the client's machine before being uploaded to the server, ensuring the server operator cannot access plaintext file content. A hybrid encryption scheme with integrity control shall be used. Two ciphers and two cipher modes shall be supported.

Credentials should also be stored in a secure manner.

- **File Encryption (Client-Side):**

1. For each new transfer, generate a strong, random symmetric key (File Key) using an currently secure algorithm defined by the user. Integrity control must be considered.
2. Encrypt the file(s) on the user's machine using this File Key.
3. The client uploads only the encrypted file blob to the server.

- **Key Encryption for User-Specific Shares (Client-Side):**

1. For each intended recipient, the uploader's client fetches the recipient's public key from the server.
2. The uploader's client encrypts the File Key using each recipient's public key.
3. These individually encrypted File Keys are sent to the server along with the transfer metadata.

- **Key Handling for Public Shares (Client-Side):**

1. For public shares, the unencrypted File Key is appended to the generated share URL as a URL fragment (e.g., `.../download/transferId#<base64_encoded_key>`).
2. The server never receives the fragment part of the URL, thus it

never sees the plaintext key.

- **Storage (Server-Side):** The server stores the encrypted file blob and, for user-specific shares, the collection of encrypted File Keys. The server cannot decrypt any of this data.

Multi-Level Security (MLS)

The system must enforce a lattice-based access control model based on security labels assigned to subjects (users) and objects (files). A user's security context is conveyed via a short-lived, cryptographically signed token (e.g., JWT) issued by the **Security Officer**. This token must be presented with every request requiring an MLS check.

A. Bell-LaPadula Model (Confidentiality with Departments)

This model combines hierarchical levels with non-hierarchical categories (departments) to control confidentiality.

- **Security Levels (Clearance/Classification):**

1. Top Secret (Highest)
2. Secret
3. Confidential
4. Unclassified (Lowest)

- **Security Labels (Departments):** A set of non-hierarchical categories, such as `{FINANCE, HR, ENGINEERING}`.
- **User and File Labels:**
- Every **user's token** contains a **Clearance Level** and a **set of authorized Departments**.

- Every **file** is assigned a **Classification Level** and a **set of associated Departments**.
- **Rules to Enforce:**

1. **Simple Security Property (No Read Up):** A user can read a file only if their clearance level is **greater than or equal to** the file's classification level AND the file's set of departments is a **subset of** the user's authorized departments.
- $\text{Level(Subject)} \geq \text{Level(Object)}$ AND $\text{Departments(Subject)} \supseteq \text{Departments(Object)}$
2. ***-Property (No Write Down):** A user can upload a file only if their clearance level is **less than or equal to** the file's classification level AND the user's set of authorized departments is a **subset of** the file's departments.
- $\text{Level(Subject)} \leq \text{Level(Object)}$ AND $\text{Departments(Subject)} \subseteq \text{Departments(Object)}$
- **Upload Constraints:** When uploading a file, a user can only assign:
- A classification level higher or equal to their clearance level.
- A set of departments that is a subset of their own authorized departments.

User Key Management

- **Asymmetric Key Pairs:** Each user must have an asymmetric key pair (e.g., RSA-4096 or a suitable ECC curve) for encryption purposes.
- **Public Key Distribution:** The user's public key must be stored on the server and made available to other users for

encryption.

- **Private Key Security:** The user's private key must be encrypted client-side using a strong key derived from the user's password (e.g., using PBKDF2 or Argon2). The resulting encrypted private key blob is then stored on the server. Upon successful authentication, the client retrieves this encrypted blob and decrypts it locally in the browser's memory for the duration of the session. The plaintext private key **must never** be transmitted to or stored on the server.

System Keys for Signing

The application must possess an asymmetric key pair used exclusively for signing security attribute tokens. The private key must be securely stored on the server, accessible only to logic executed by a **Security Officer**. The public key can be used by any backend service to verify the authenticity of the tokens.

Secure Communication Protocol

All communication between the client and the server **must** be encrypted using Transport Layer Security (TLS) 1.2 or higher. The server must be configured with a valid, trusted TLS certificate to prevent man-in-the-middle attacks. No unencrypted HTTP connections should be permitted.

Suggested Technical Stack

Students are free to choose their technology stack. The following are suggestions:

- **Backend:** Node.js (Express), **Python (Django/Flask)**, Java (Spring Boot)
- **Frontend:** No Frontend is required (but it may exist)
- **Database:** SQLite
- **Client:** A command line tool implemented in any language. **Python** is recommended
- **Encryption Libraries:** Use well-vetted, standard cryptographic libraries for your chosen language (e.g., `Web Crypto API` for frontend, `crypto` for Node.js, `cryptography` for Python).

REST API Specification

The backend must expose a RESTful API for all client-side operations. All endpoints should be prefixed with `/api`. Authentication should be handled via session cookies or JWTs sent in an `Authorization` header.

Authentication

Method	Endpoint	Description
POST	`/auth/login`	Authenticates a user and returns a token/session.
POST	`/auth/logout`	Logs out the current user.
POST	`/auth/activate`	Activates a new account into the system, using a already created username and one time password

Department Management (Administrator Only)

Method	Endpoint	Description
POST	`/departments`	Creates a new department.
GET	`/departments`	Retrieves a list of all departments.
DELETE	`/departments/{deptId}`	Deletes a department.

User Management (Admin OR Security Officer OR Authenticated User)

Method	Endpoint	Description	Authorization
POST	`/users`	Creates a new user to the system. This sets the username and a one time use password	Administrator
GET	`/users`	Retrieves all users in the system.	Administrator, Security Officer
DELETE	`/users/{userId}`	Removes a user from the system.	Administrator
PUT	`/users/{userId}/role`	Updates a user's role (e.g., promote to Trusted Officer). The adequate	Security Officer, Administrator

Method	Endpoint	Description	Authorization
		objects are added to the user account, signed with the caller private key	
GET	`/users/{userId}/clearance`	Gets the clearance tokens for the specified user, containing their assigned clearance, departments, and integrity levels.	Security Officer, Authenticated User
PUT	`/users/{userId}/clearance`	Adds a clearance token for the specified user, containing their assigned clearance, departments, and integrity levels.	Security Officer
PUT	`/users/{userId}/revoke/{tokenId}`	Adds a revocation token for the specified user.	Security Officer
GET	`/users/{userId}/key`	Retrieves a user's **public** key for encryption.	Authenticated User

Method	Endpoint	Description	Authorization
PUT	`/users/me/vault`	Uploads or updates the current user's password-encrypted **private** key blob.	Authenticated User
GET	`/users/me/vault`	Retrieves the current user's password-encrypted **private** key blob.	Authenticated User
GET	`/user/me/info`	Get current user information.	Authenticated User
POST	`/user/me/info`	Updates existing information, such as the password	Authenticated User

File Transfers

Method	Endpoint	Description	Authorization
GET	`/transfers`	Lists existing transfers created by the user.	Authenticated User
POST	`/transfers`	Uploads an encrypted file and its	Authenticated User

Method	Endpoint	Description	Authorization
		metadata. Can be used to allow sharing with new users.	
GET	`/transfers/{transferId}`	Retrieves transfer metadata and the user's encrypted File Key.	Authenticated User
DELETE	`/transfers/{transferId}`	Delete transfer metadata and data.	Authenticated User
GET	`/download/{transferId}`	Downloads the raw encrypted file blob.	Authenticated User
GET	`/audit/log`	Retrieves the complete, ordered hashchain audit log.	Auditor
PUT	`/audit/validate`	Add a validation object to a given log entry.	Auditor

Deliverables

1. **Source Code:** A link to a private Git repository containing the complete, well-commented source code for the server and client.
2. **Documentation:** A `README.md` file in the root of the repository that includes:

- A high-level description of the project and its folder structure.
- Detailed, step-by-step instructions on how to set up the development environment, build the project, and run it locally (Docker containers can be used).
- Clear explanations of any environment variables or configuration files needed.

3. **Project Report (PDF):** A report detailing:

- The final system architecture and technology choices.
- A detailed explanation of how encrypted storage and authentication was implemented.
- A thorough walkthrough of the implementation of the RBAC, MLS, and the hashchain auditing mechanism.
- The findings from the ASVS audit (see item 5).
- **Deadline:** December 5th, 23.59.

4. **ASVS Audit:** Students must perform a security audit of their own application based on the OWASP Application Security Verification Standard (ASVS).

- **Select one chapter** from the ASVS (e.g., V2: Authentication, V4: Access Control, V5: Validation, Sanitization and Encoding).
- **Perform the audit:** Evaluate your application against the

verification requirements in the selected chapter.

- **Report findings:** Document your findings, including which requirements are met, partially met, or not met, and provide a brief justification. This report should be included as a section in the main Project Report.
- **Deadline:** December 19th, 23.59.

5. **Project Defense:** The project developed must be presented and defended during the first weeks of January.

> Faculty assumes that the project is developed in the VM provided for the course.