Segurança e Informação nas Organizações

Project:
Repository for documents for organizations

Bernardo Borges, 103592

Alexandre Regalado, 124572

Antonio Pedrosa, 93279

DETI

Universidade de Aveiro

3rd Delivery, 30th December

1. Introduction

This report explains the SIO (Secure Information and Organizations) project, completed in 2024. The project focuses on building a secure system for managing documents and organizations, with strong protections and role-based access controls.

The goal of this report is to describe how the system works, its main security features, and how it follows standards like OWASP ASVS. The report also shows how the system keeps sensitive information safe and easy to manage.

2. Architectual Decisions

The design of this project reflects various choices aimed to ensure security and easiness of implementation of future features.

This section outlines key decisions and some logic behind them.

Security

Industry Standard Cryptography:

- Cryptographic algorithms such as **AES**, **ChaCha20** and **Camellia** are used for file encryption
- **SHA-256** is employed for hashing and HMAC operations for message integrity.
- **Diffie-Hellman** is used for key exchange, ensuring secure shared key between server and client.

End to end protection:

- All sensitive data is encrypted during communication using **AES** encryption.
- **HMAC** ensures integrity of encrypted message preventing tampering

Server-side Validation

Access control:

- Role based access control is enforced using server-side decorators like `verify_permission`
- All access decisions (permissions, roles, organization checks) are made on the server to prevent client-side bypass.

Data Validation:

- Input validation for sensitive parameters (document names, usernames) is performed on the server

- Database queries include organization and user checks to prevent unauthorized access.

Tokens

Session Management:

- Sessions are encoded using JSON Web Tokens (JWTs).
- Tokens encode an expiration date to reduce the risk of replay attacks.
- Roles and permissions are encoded in the token, removing the server-side session storage.

Stateless Design:

- Reduces the memory management on the server.
- Enables horizontal scalability.

Modular Codebase

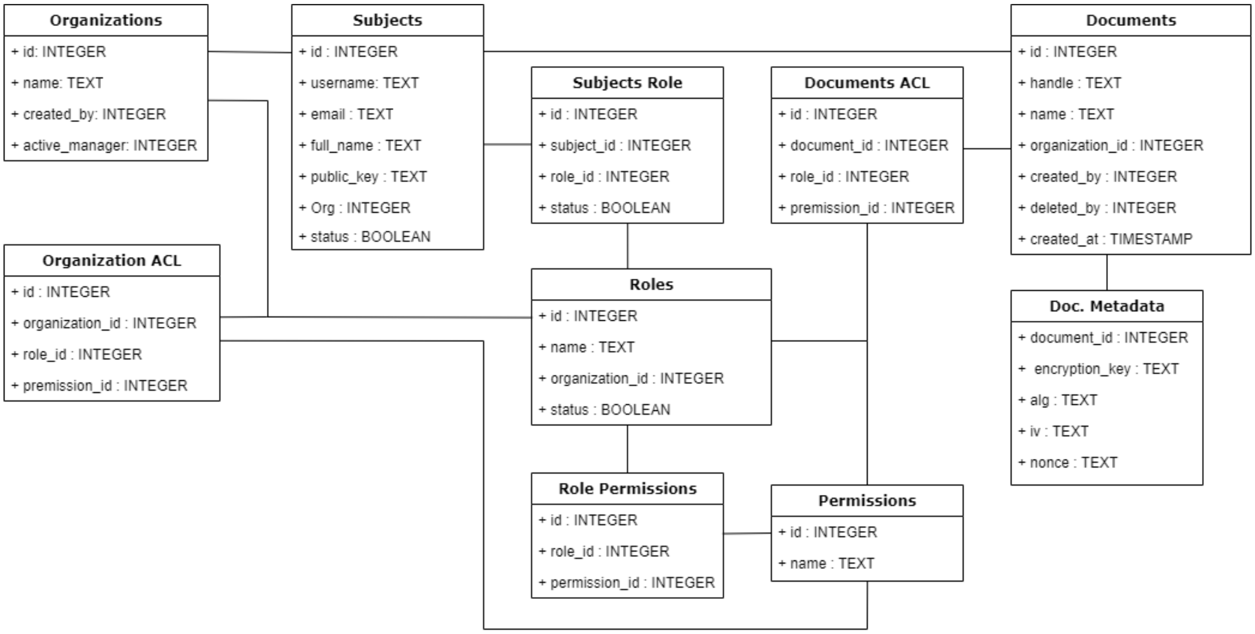
A significant design decision in the codebase is the extensive use of **decorators** to implement various checks like access control, session validation, and secure communication.

This approach centralizes shared functionality while keeping the main application logic clean.

3. Database Schema

The database schema is designed to handle organizations, users, roles, permissions, and documents. Here's a simple description:

The schema organizes information about organizations, users (called subjects), and documents. Each organization has roles that users can take on, and permissions define what these roles can do. Documents are stored securely with metadata for encryption, and access control ensures only certain roles can work with specific documents. There are also tables for linking roles, permissions, and users to keep everything organized and secure.



4. API Side

The API supports managing organizations, users, roles, permissions, and documents. It provides endpoints for creating, reading, updating, and deleting these entities, with robust access control to ensure security and proper role-based management.

Core Features

- **Authentication and Sessions**: To authenticate users and maintain session-based communication.
- **Organizations**: To manage organizations and their related data.
- **Users (Subjects)**: To manage users within an organization.
- **Roles**: To manage roles assigned to users and permissions to these roles.
- **Permissions**: To define and manage actions users can perform through roles.
- **Documents**: To securely manage documents and their metadata.
- **Access Control (ACLs)**: To control which roles have access to documents and organizational features.

Security Features

- **Authentication**: Each session is secured using tokens.
- **Key Exchange**: Diffie-Hellman key exchange is used to establish shared secrets for encrypting communication between clients and the server.
- **Encryption**: Documents are encrypted using AES and other algorithms and can only be accessed with proper metadata and keys.
- **Data Integrity**: HMAC validation ensures that messages are not tampered with during transmission.
- **Access Control**: Role-based permissions ensure that users can only perform actions they are allowed to.

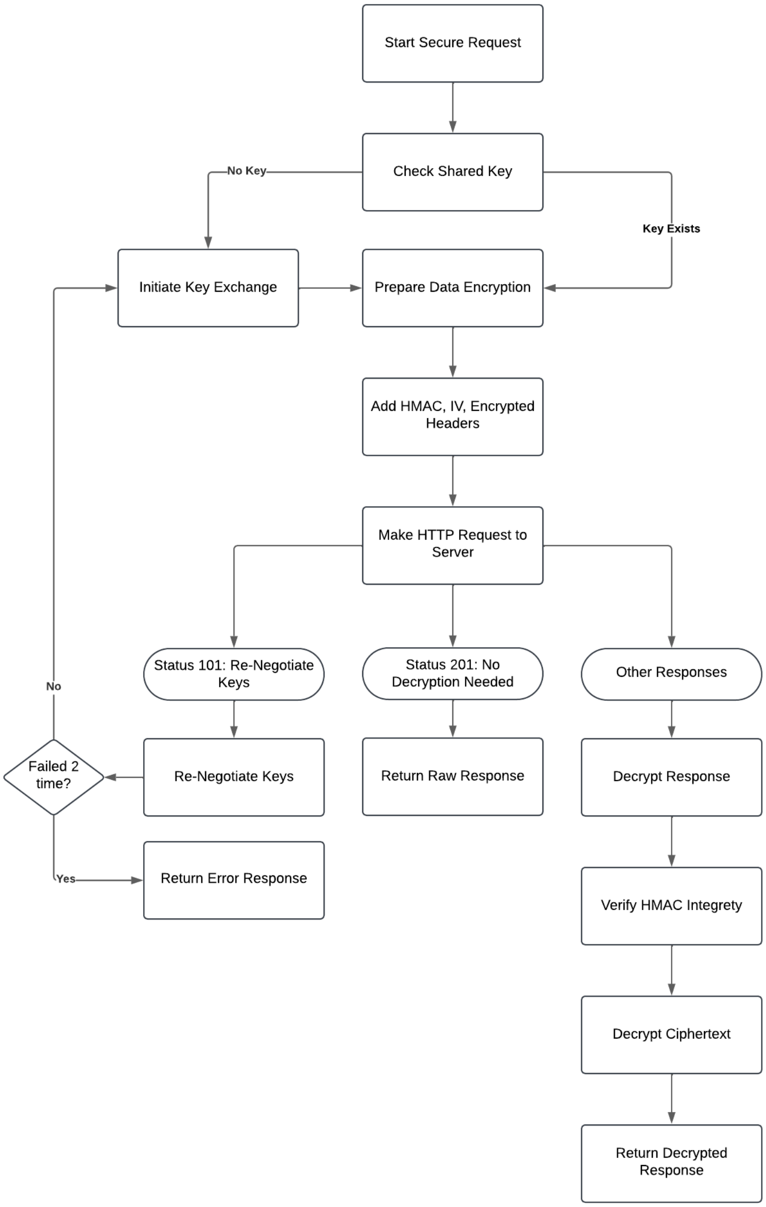
Error Handling Responses

- **200 OK**: Successful operation.
- **201 Created**: Resource created successfully.
- **400 Bad Request**: Invalid input or parameters.
- **401 Unauthorized**: Invalid session token.

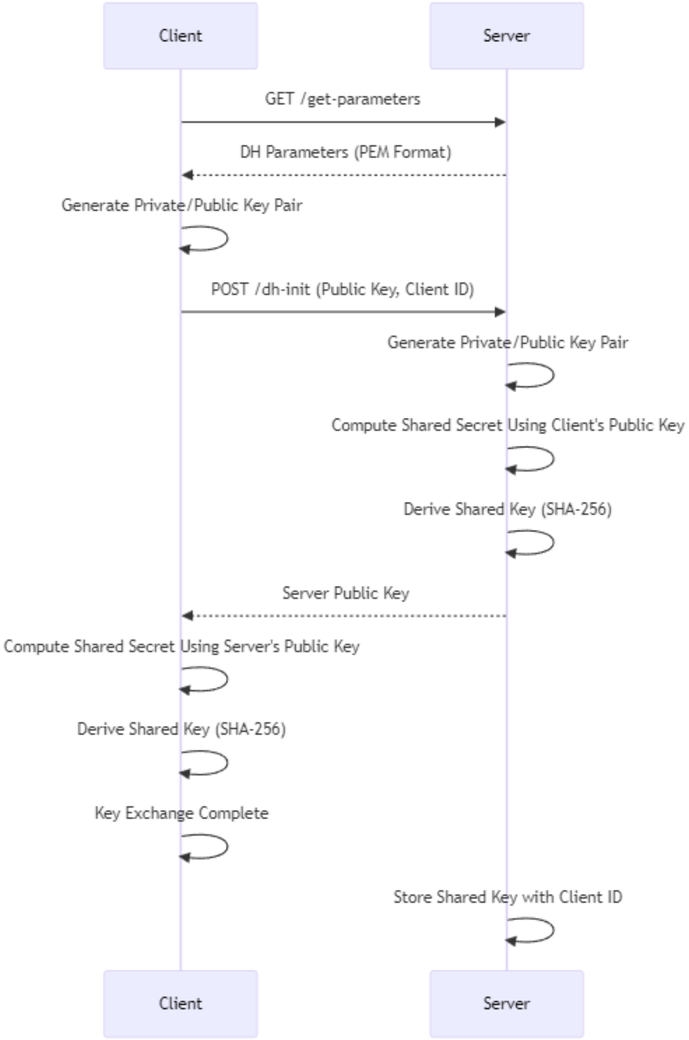
- **403 Forbidden**: User does not have the required permissions.
- **404 Not Found**: Resource not found.
- **500 Internal Server Error**: Unexpected server error.
- **101 Need Key Renegotiation**: The shared secret key needs to be re-establish

5. Secure Communication

Secure Request high level flow chart



Key Exchange flow chart


Client
Server
GET /get-parameters
DH Parameters (PEM Format)
Generate Private/Public Key Pair
POST /dh-init (Public Key, Client ID)
Generate Private/Public Key Pair
Compute Shared Secret Using Client's Public Key
Derive Shared Key (SHA-256)
Server Public Key
Compute Shared Secret Using Server's Public Key
Derive Shared Key (SHA-256)
Key Exchange Complete
Store Shared Key with Client ID
Client
Server

6. Tests

The client-side interface provides command-line tools for securely managing organizations, users, roles, permissions, and documents. These commands interact with the server API to facilitate operations while maintaining encryption and authentication.

Key Functionalities

1. **Key Management**
    a. **`rep_subject_credentials`**: Generates an RSA key pair and encrypts the private key.
    b. **Usage**: rep_subject_credentials <password> <credentials_file>
        i. The private key is securely encrypted with the provided password and saved locally.
        ii. Public and private keys are stored in the `.sio` directory in PEM format.

2. **Organization Management**
    a. **`rep_create_org`**: Creates a new organization by specifying details such as the organization name, user information, and public key.

    b. **Usage**: `rep_create_org <organization> <username> <name> <email> <pub_key_file>`
        i. This command registers the organization and sets the user as the creator.

3. **User Management**
    a. **`rep_add_subject`**: Adds a new subject to the organization.
        i. **Usage**: `rep_add_subject <session_file> <username> <name> <email> <cred_file>`
        ii. Sends the public key and user details to the server for registration.

    b. **`rep_list_subjects`**: Lists all subjects in an organization, with optional filtering by username.
        i. **Usage**: `rep_list_subjects <session_file> [username]`
        ii. Displays the username, full name, and status (Active/Suspended).

4. **Session Management**
    a. **`rep_create_session`**: Creates a new session by authenticating the user.
        i. **Usage**: `rep_create_session <organization> <username> <password> <cred_file> <session_file>`
        ii. Uses a nonce and the user's private key to securely create a session token.

5. **Role and Permission Management**
    a. **`rep_add_role`**: Adds a new role to the organization.
        i. **Usage**: `rep_add_role <session_file> <role>`
        ii. Sends the role name to the server for registration.

    b. **`rep_add_permission`**: Assigns permission to a role.
    c. **Usage**: `rep_add_permission <session_file> <role> <permission>`

    d. **`rep_remove_permission`**: Removes permission from a role.
        i. **Usage**: `rep_remove_permission <session_file> <role> <permission>`

6. **Document Management**
    a. **`rep_add_doc`**: Encrypts and uploads a document.
        i. **Usage**: `rep_add_doc <session_file> <document_name> <file>`
        ii. The document is encrypted before being sent to the server.

    b. **`rep_list_docs`**: Lists documents in the repository, with optional filters for username and date.
        i. **Usage**: `rep_list_docs <session_file>`
        ii. Filters include date criteria (`nt`, `ot`, `et`) and username.

    c. **`rep_get_doc_file`**: Downloads and decrypts a document.
        i. **Usage**: `rep_get_doc_file <session_file> <document_name> [file]`
        ii. The decrypted content can be saved or printed to the console.

d. **`rep_delete_doc`**: Marks a document as deleted.

   i. **Usage**: `rep_delete_doc <session_file> <document_name>`

7. **Access Control (ACL) Management**

   a. **`rep_acl_doc`**: Modifies document permissions for roles.

      i. **Usage**: `rep_acl_doc <session_file> <document_name> <operation> <role> <permission>`

      ii. Operations:

         1. +: Add permission.
         2. -: Remove permission.

7. ASVS Analysis

In the scope of the course, we've decided to make an analysis on a chapter of the ASVS. We've landed on using the "Access Control" chapter to further analyze if there are any possible breaches or holes in the access side of the application.

In this chapter, we'll go by each point, specifying the ASVS Level, its requirement, if its valid (with code references for proof) and a small comment on it.

7.1 General Access Control Design

7.1.1

Verification Requirement: Verify that the application enforces access control rules on a trusted service layer, especially if client-side access control is present and bypassable.

ASVS Level: 1

Validity: Yes

Source: *verify_permission* function (in app.py)

Access control in this application is enforced strictly on the server-side, using robust mechanisms that prevent unauthorized access to sensitive resources. The core of this implementation is centered around the component *verify_permission* decorator.

Importantly, no critical access control decisions are left to client-side logic, which is often prone to manipulation. By enforcing access control rules exclusively on the server, the application ensures that authentication and authorization cannot be tampered with, thus maintaining the integrity of sensitive operations.

7.1.2

Verification Requirement: Verify that all user and data attributes and policy information used by access controls cannot be manipulated by end users unless specifically authorized.

ASVS Level: 1

Validity: Yes

Source: *extrat_token_info* (in costum_auth.py) and *verify_permission* (in app.py)

In this implementation, user roles and permissions are embedded within JWT (JSON Web Tokens), which are cryptographically signed to prevent tampering. These tokens are generated by the server upon successful authentication and are sent to the client for subsequent requests. The integrity of these tokens is protected by a secure RSA signature, which ensures that any unauthorized modification to the token data would invalidate the signature.

All critical access decisions are made server-side, where they cannot be bypassed by modifying or forging tokens on the client side.

7.1.3

Verification Requirement: Verify the principle of least privilege exists - users should only access resources for which they possess specific authorization.

ASVS Level: 1

Validity: Partially

Source: *verify_permission* (in app.py) and the logic behind Role Permissions

The current system partially enforces the principle of least privilege. While roles and permissions are used to restrict user access, there are potential gaps, particularly with role overrides (e.g., for managers) and undefined default access. These issues may allow users to access resources beyond what is necessary for their tasks, violating the least privilege principle.

To fully comply with this verification requirement, the application should consider tightening role-based access controls, ensuring that even privileged roles like manager do not have excessive access to resources. Additionally, undefined access defaults should be eliminated by ensuring that all user's permissions are explicitly configured, and no unnecessary access is granted by default.

7.1.5

Verification Requirement: Verify that access controls fail securely, including when an exception occurs.

ASVS Level: 1

Validity: Yes

Source: *verify_permission* (in app.py)

The access control system effectively fails securely by handling exceptions gracefully and returning appropriate error responses (e.g., 401 Unauthorized and 403 Forbidden) when a failure occurs. These mechanisms ensure that, in the event of an error or exception (such as missing permissions or system issues), users are not given unauthorized access or exposed to unnecessary internal details. This behavior aligns with the ASVS requirement for secure failure handling and minimizes the risk of revealing security vulnerabilities or sensitive information.

7.2 Operational Level Access Control

7.2.1

Verification Requirement: Verify sensitive data and APIs are protected against IDOR attacks.

ASVS Level: 1

Validity: Yes

Source: *list_docs* and *get_doc_metadata* (both in app.py)

IDOR attacks occur when an attacker can manipulate input parameters (such as URLs, request bodies, or API endpoints) to access data or resources they should not be authorized to view or modify. This often happens when user input is used directly to query for resources without proper validation or access checks. To mitigate such vulnerabilities, it is essential to ensure that sensitive data and APIs are protected against unauthorized access by applying proper access control checks at the operational level.

For that reason, the application implements robust protections against IDOR attacks by using role-based access control, secure database queries, and comprehensive user/organization-specific validation. These mechanisms effectively ensure that sensitive data and APIs are accessible only to authorized users, protecting the system from unauthorized access, and preventing attackers from exploiting IDOR vulnerabilities.

7.2.2

Verification Requirement: Verify the application enforces strong anti-CSRF mechanisms.

ASVS Level: 1

Validity: No

Cross-Site Request Forgery (CSRF) is an attack in which a malicious user tricks an authenticated user into unknowingly submitting a request to a web application on which they are authenticated, potentially performing actions such as changing user settings, submitting forms, or making unauthorized transactions. CSRF attacks exploit the trust that a web application has in the user's browser, as the browser automatically includes authentication cookies in any request sent to the application.

To mitigate CSRF attacks, the application must enforce strong anti-CSRF mechanisms, which typically involve ensuring that any state-changing requests (e.g., POST, PUT, DELETE) include a unique, unpredictable token that the server validates. This token helps confirm that the request is coming from a legitimate source and not from a third-party attacker.

The application does not currently enforce any anti-CSRF mechanisms, which makes it vulnerable to CSRF attacks. To ensure robust protection against this threat, the application must implement anti-CSRF tokens for all state-changing requests, ensure that tokens are validated correctly on the server-side.

7.3 Other Access Control Considerations

7.3.1

Verification Requirement: Verify administrative interfaces use multi-factor authentication (MFA).

ASVS Level: 1

Validity: No

Currently, no multi-factor authentication (MFA) is implemented for administrative interfaces, leaving them vulnerable to unauthorized access if an attacker is able to compromise an administrator's credentials. To address this critical security gap, it is highly recommended to implement MFA for all administrative interfaces and privileged actions, using industry-standard solutions such as hardware tokens. This will add a much-needed layer of defense, significantly reducing the likelihood of unauthorized access and enhancing the overall security posture of the application.

7.3.2

Verification Requirement: Verify directory browsing is disabled, and metadata like .git or .svn is not accessible.

ASVS Level: 1

Validity: Yes

Source: Flask Implementation (database.py and secure_communication.py)

The application does not enable directory browsing by default, and sensitive directories like `.git` and `.svn` are not accessible unless the server is misconfigured. Flask's default behavior, in combination with proper server hardening and configuration, ensures that directory browsing is disabled and sensitive metadata files are protected.

To maintain security, it is essential to ensure that the production server (e.g., Apache, Nginx) is properly configured to block access to version control metadata directories like .git and `.svn,` perform regular security audits to verify that no sensitive files are unintentionally exposed and use file permissions and firewall rules to further protect any files that should not be publicly accessible.

By taking these precautions, the application can remain secure from risks associated with directory browsing and exposed metadata.

7.3.3

Verification Requirement: Verify the application enforces additional authorization for lower value systems or segregates duties for high-value applications.

ASVS Level: 1

Validity: No

Currently, the application does not enforce additional authorization for high-value systems or segregate duties for critical actions. The lack of step-up authentication and adaptive authentication means that sensitive operations may be performed without sufficient scrutiny, increasing the risk of unauthorized access and privilege escalation. To improve security for high-value systems, it is recommended to implement multi-layered authentication for sensitive actions, including MFA or email/phone verification, to integrate adaptive authentication that assesses the risk level of the action and adapts the authentication requirements accordingly and enforce segregation of duties for critical actions to ensure that no single user can execute high-risk operations without proper checks.

8. Bibliography

https://cryptography.io/en/latest/

https://github.com/OWASP/ASVS and https://github.com/shenril/owasp-asvs-checklist

Practical Slides and Solved Exercises from Class