SIO Final Project - Repository

Pedro Pinto nº115304, João Pinto nº104384, Guilherme Santos nº113893

1. Introduction

This project aims to develop a secure repository for managing organizational documents, addressing the critical need for confidentiality, integrity, and accessibility in sensitive data management. Organizations face increasing challenges in protecting their documents from breaches, unauthorized access, and data loss, which underscores the importance of secure systems. Our repository leverages **MongoDB** for its flexibility with document-oriented data and **Docker** for consistent deployment. To further enhance security, files are stored encrypted in the filesystem under the project directory in a dedicated `/vault` directory. The repository integrates cryptographic mechanisms such as ECDSA for signing and AES-GCM for encryption, ensuring robust protection against common security threats. Additionally, a full analysis of Chapter 6 of OWASP Application Security Verification Standard (ASVS) was conducted, ensuring compliance with best practices for cryptographic implementations and secure data management.

To Run the System

Server: Launch the server by running:

```
docker compose up --build
```

Client: Execute commands by running:

```
python3 subject.py <command> <args>
```

2. Security Decisions

This section outlines the security measures implemented in the project, focusing on workflows designed to ensure data confidentiality, integrity, and availability. Sequence diagrams are used to illustrate key workflows, such as creating an organization, creating a session, and managing document access.

Session Robustness Against Attacks

The following table summarizes how the system's session management is designed to mitigate common security threats:

Threat	**Description and Mitigation**
Eavesdropping	A passive attacker cannot access the content being exchanged. Encryption ensures that all contents remain confidential.
Impersonation	An active attacker cannot pose as a victim subject or the repository. Authentication of sessions (e.g., login) and interactions are implemented to verify identities.
Manipulation	An active attacker cannot manipulate data exchanged within a session. Cryptographic integrity controls ensure that modifications are detected.
Replay	An attacker cannot replay interactions that occurred during a session. The system detects out-of-order or past messages using mechanisms like `msg_id`.

Table 1: Session Robustness Against Common Attacks

Cryptographic Details

The project employs modern cryptographic algorithms to ensure robust security:

- **ECDSA (Elliptic Curve Digital Signature Algorithm):** Used for signing responses, ensuring the integrity and authenticity of transmitted data.
- **AES-GCM (Advanced Encryption Standard - Galois/Counter Mode):** Provides confidentiality and integrity for session data and document storage.
- **HKDF (HMAC-based Key Derivation Function):** Ensures secure and unique session keys derived from the shared secret during key exchanges.

Key Management: Session keys are generated for each session and are never exposed. On the server-side, session information is kept in memory to enhance security, while on the client-side, it is stored in a designated file referred to as the `session_file`.

2.1. Local commands

Generating Subject Credentials

Derives a private key from the password using ECC (Elliptic Curve Cryptography), ensuring the password is not directly stored. Generates a public key, which is serialized in PEM format and securely stored in a credentials file.

Decrypting Files

Supports AES-GCM for decryption, ensuring both confidentiality and integrity of file contents. Verifies decryption integrity using encryption tags, rejecting tampered or invalid files.

2.2. Commands that use the anonymous API

We chose to explain the workflow of creating an organization, creating a session, and downloading a file as examples of the commands that use the anonymous API.

Creating an Organization

The workflow for creating an organization includes the following security measures:

1. **Server-Side Validation:** The server validates the received data, ensuring all required fields (`organization`, `username`, `name`, `email`, and `public_key`) are present.
2. **Master Key Usage:** The server's master key is used to derive a private key, which signs the response to ensure its integrity and authenticity.
3. **Signed Response:** The server returns a response containing the organization details and a cryptographic signature generated using ECDSA with SHA-256. This allows the client to verify the authenticity of the data.
4. **Database Security:** The organization data, including roles and permissions, is securely stored in the database. Sensitive metadata of documents is encrypted, ensuring protection against unauthorized access and maintaining data confidentiality.

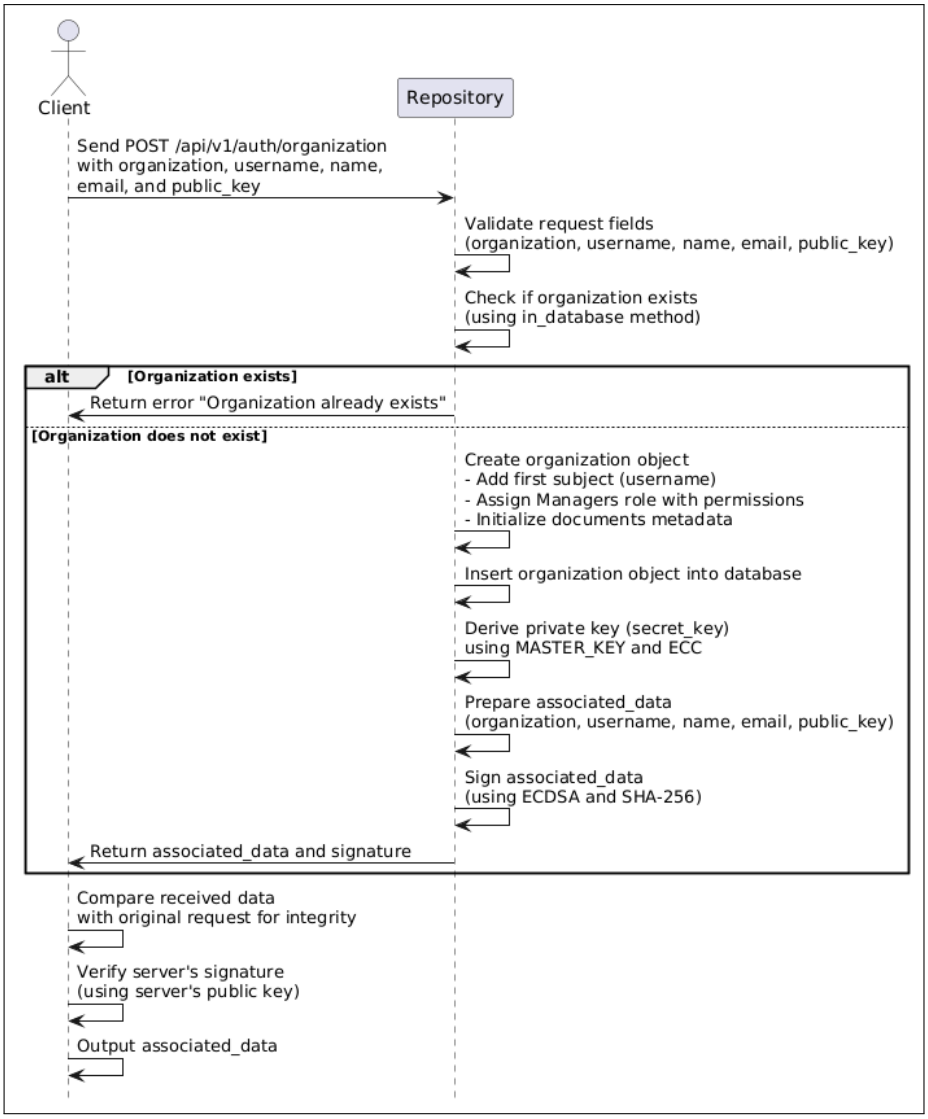

Figure 1: Create a Organization.

Creating a Session

The process of creating a session ensures secure communication between the client and the server using a combination of cryptographic processes:

1. **Client-Side Key Generation:**
   The client generates an ephemeral private key derived from their private key (itself derived from the password) and computes the corresponding ephemeral public key.

2. **Wrapping and Signing Data:**
   The client creates the `associated_data`, which includes the `organization`, `username`, and the `client_ephemeral_public_key`. This data is signed using **ECDSA** (Elliptic Curve Digital Signature Algorithm) with **SHA-256** to ensure authenticity and integrity. The signed data, along with the `associated_data`, is sent to the server.

3. **Server-Side Verification and Key Exchange:**
   The server verifies the client's signature using the client's public key. It then generates its own ephemeral private and public keys. Using **ECDH** (Elliptic Curve Diffie-Hellman), the server calculates a `shared_key` based on its ephemeral private key and the client's ephemeral public key.

4. **Key Derivation for Encryption:**
   The `shared_key` is processed with **HKDF** (HMAC-based Key Derivation Function) to derive a `derived_key`, adding an additional layer of cryptographic protection for the session.

5. **Session Information and Response:**
   The server stores the session details and sends back the `session_id` and its `ephemeral_public_key`, signed with the server's private key to ensure authenticity.

6. **Client Verification and Storage:**
   The client verifies the server's signature using the server's public key. It computes the `shared_key` and derives the same `derived_key` using the server's ephemeral public key. Finally, the session details are saved to a `session_file` on the client side for future use.

These cryptographic processes are illustrated in the sequence diagram below, Figure 2.

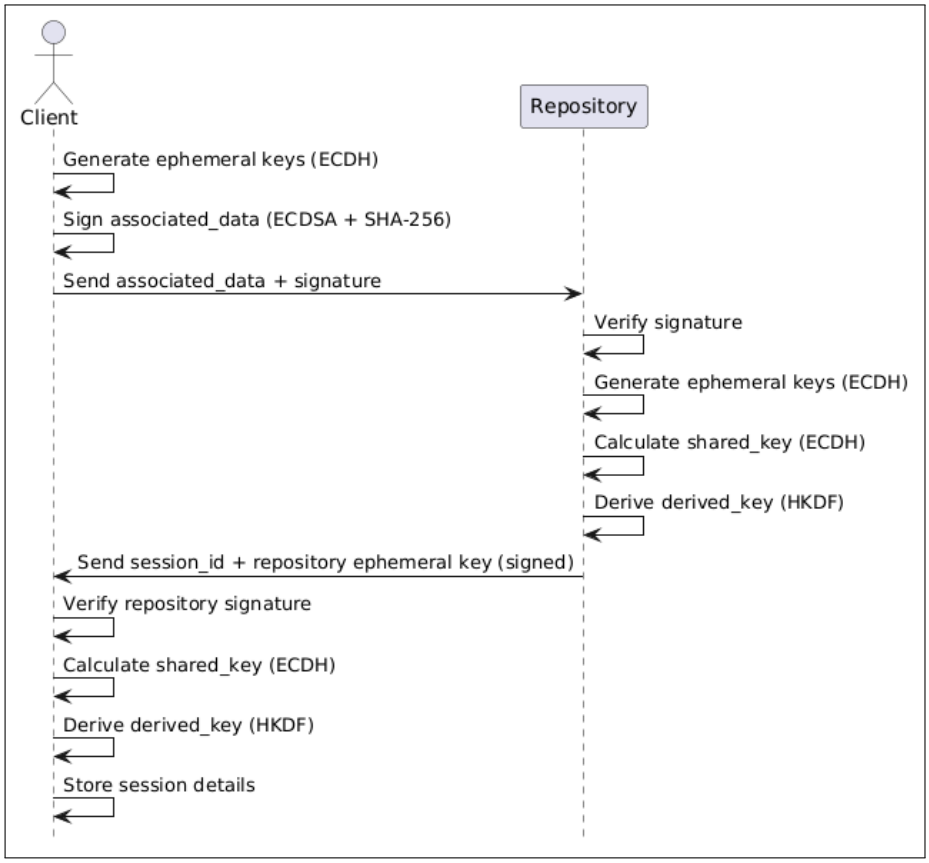

Figure 2: Create a Session.

Communicating with the Repository

To illustrate the secure communication mechanisms in the Repository, we provide an example of how a file is downloaded using its handle. The workflow for downloading a file ensures the integrity and authenticity of the retrieved content. The client requests the file using its handle, and the server locates the file (in `/vault`) and returns its content along with a cryptographic signature. The signature, generated using the server's private key, allows the client to verify the authenticity and integrity of the file, ensuring it has not been tampered with during transmission. Upon successful validation, the client can write the file content to a specified location or output it to stdout.

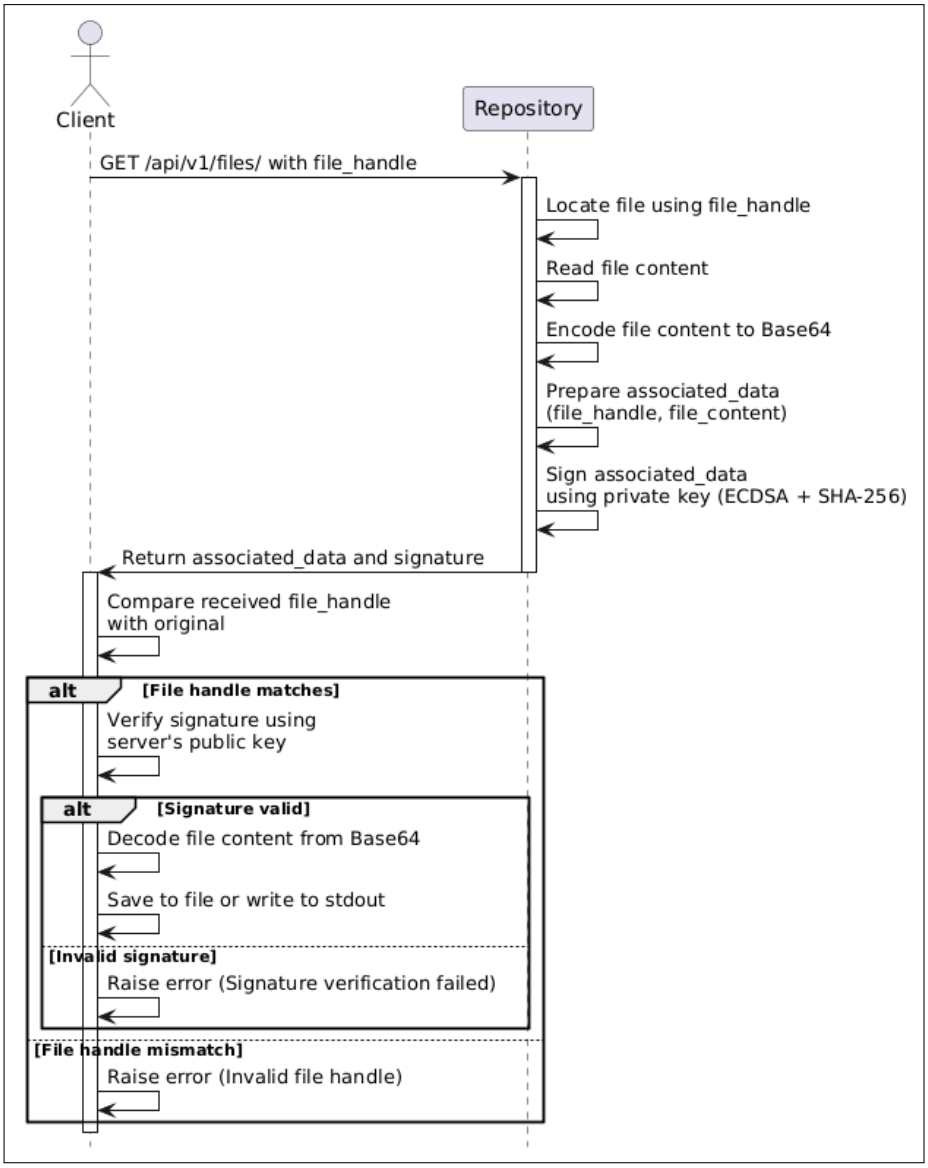

Figure 3: Downloading a File

2.3. Commands that use the authenticated API

Secure Communication Workflow

The secure communication between the client and server is achieved through a systematic process involving **data encapsulation**, **data transmission**, and **data decapsulation**. This ensures confidentiality, integrity, and protection against replay attacks in a simple and maintainable structure.

Session Communication Details

- **Algorithm**: **AES-GCM** (Advanced Encryption Standard in Galois/Counter Mode) is used for both **encryption** and **decryption**. It also ensures data **confidentiality** and **integrity**. This versatility helped in each session communication.
- **Keys**: The `derived_key` for encryption and decryption is established securely during session creation using **ECDH** and **HKDF**. This is a shared key between the repository and the client.
- **Associated Data**: This additional data (e.g., `msg_id` and `session_id`) is authenticated during

encryption but **not encrypted** itself, preventing **tampering**. When a message is received it helps a quicker decapsulation and verification, they work as a header.

Here is how it works, in a more detailed description:

1. **Client-Side Data Encapsulation:**
   The client begins by preparing the data (`plaintext`) to be sent.
   - Session-related metadata (`msg_id` and `session_id`) are grouped into `associated_data`, which is authenticated but not encrypted.
   - The `plaintext` is serialized into bytes and encrypted using the **AES-GCM** algorithm with the `derived_key`. The encryption process generates a random `nonce` and produces the `ciphertext`, ensuring confidentiality and integrity.
   - Both the `associated_data` and the encrypted data (`nonce` and `ciphertext`) are grouped into a JSON structure and sent to the server.
2. **Server-Side Data Decapsulation:**
   Upon receiving the data, the server extracts and validates the `associated_data` to authenticate the session and verify the `msg_id` for replay attack prevention.
   - Using the stored session details and `session_id`, the server decrypts the `ciphertext` with **AES-GCM** using the `derived_key` and verifies the integrity of the associated data. If the integrity check fails, the process halts with an error.
   - The decrypted `plaintext` is then processed, and any required operations are performed on the server side.
3. **Server Response and Encapsulation:**
   After completing the requested operation, the server prepares a response in plaintext.
   - The response is encapsulated similarly: `associated_data` (updated `msg_id` and `session_id`) is combined with the encrypted response (`nonce` and `ciphertext`) using **AES-GCM** to ensure the confidentiality and integrity of the response.
   - The encapsulated data is sent back to the client.
4. **Client-Side Data Decapsulation:**
   Upon receiving the server's response, the client verifies the `associated_data`, ensuring the `msg_id` matches and the session is valid.
   - The encrypted response is decrypted using the `derived_key`, and the integrity of the response is checked. If validation is successful, the client processes the plaintext response.

This process ensures that every step of the communication is secure, authenticated, and resistant to attacks, such as replay and man-in-the-middle attacks.

We choose to explain the workflow of assuming a role as an example of the commands that use the authenticated API. In this chapter, we will also explain how the communication between the client and the server occurs.

Assume a Role

The workflow for assuming a role allows a user to assume a specific role associated with their session. The client sends a request to the server, including the desired role. The server validates the user's active status and checks whether the user holds the requested role in the organization. If authorized, the server updates the session to include the role, and the client reflects this change in the local session file. This ensures role-based access control and prevents unauthorized role assignments.

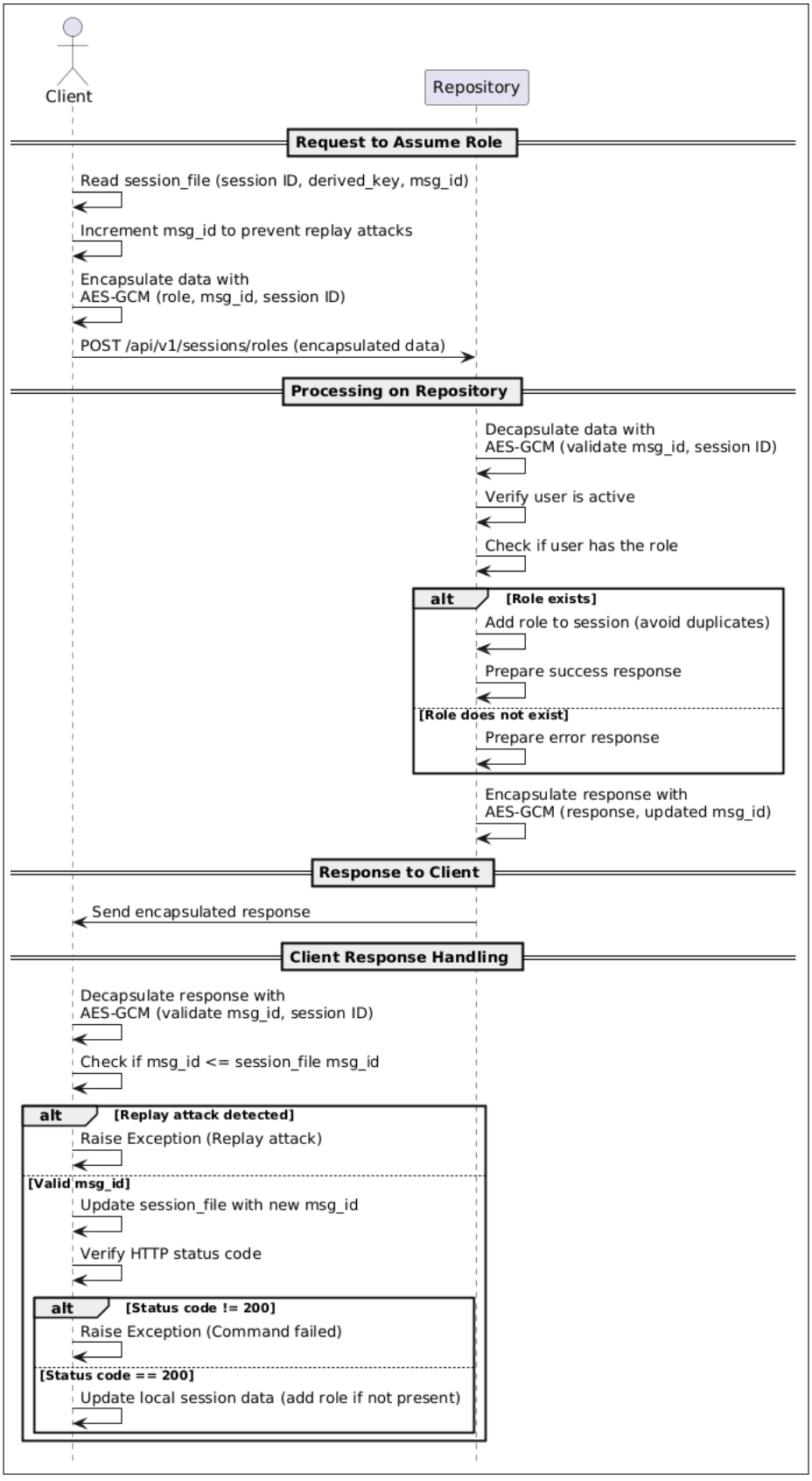

Figure 4: Assume a Role

2.4. Commands that use the authorized API

This subpart of commands are within a session and also requires some kind of permission, assumed inside the session. Therefore, all these commands use as first parameter a file with the session key. For that session, the subject must assume/resign one or more roles. We chose to explain the workflow of adding a subject to an organization as an example of the commands that use the authorized API.

Add a New Subject to the Organization

The workflow for adding a new subject to an organization begins with the client preparing the required data, including the subject's `username`, `name`, `email`, and `public key`. The public key is read from a credentials file, serialized, and included in the request. This information is encapsulated and encrypted using the session's `derived_key` for confidentiality and integrity. Once the server receives the data, it validates the session, checks the user's active state, and ensures that the session includes the `SUBJECT_NEW` permission. If these checks pass, the server adds the new subject to the organization's database with default properties, such as an "active" state. Finally, the server responds with a confirmation message, encapsulated and encrypted for secure transmission back to the client.

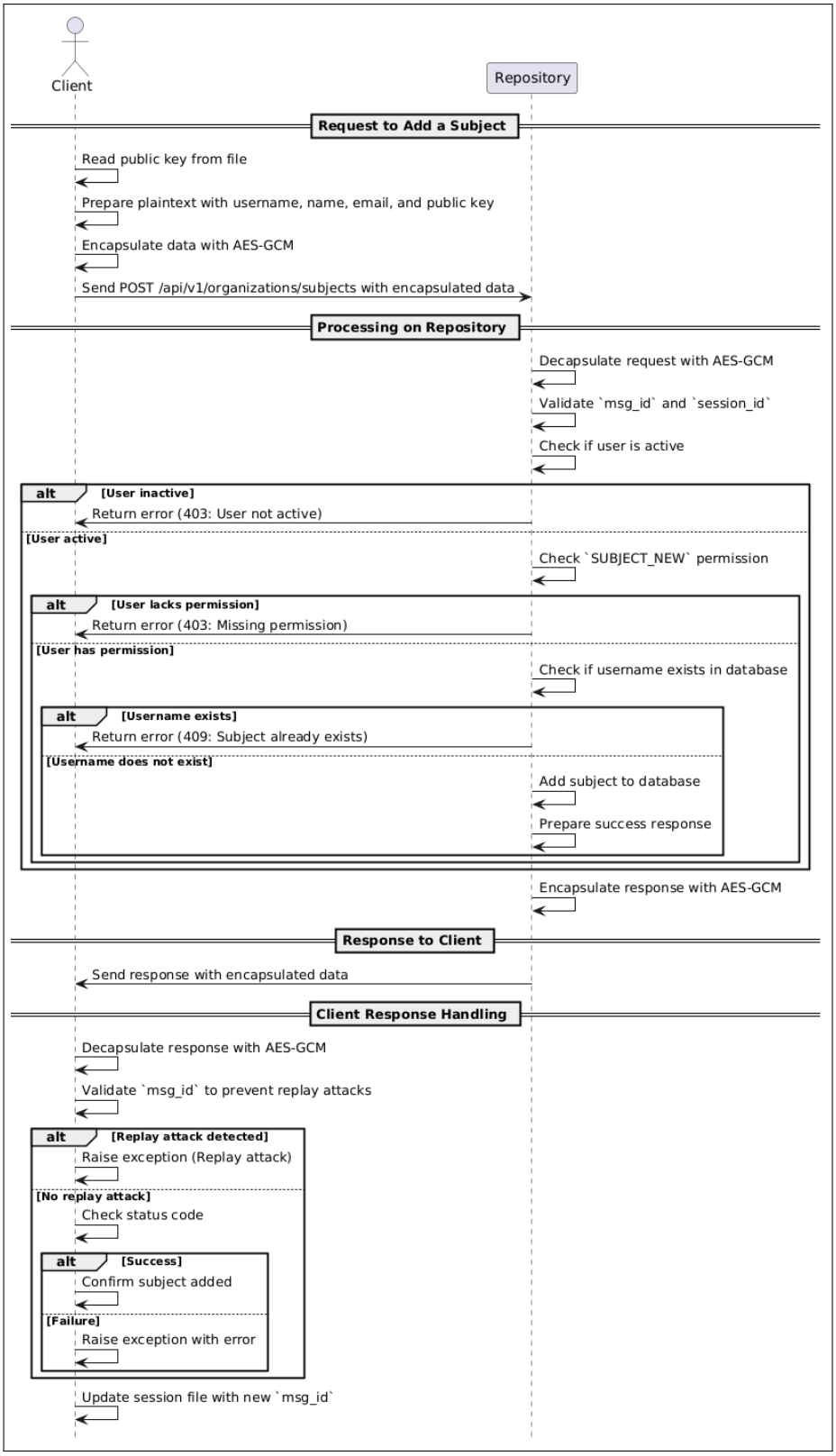

Figure 5: Adding a Subject to an Organization

3. Analysis of the software

3.1. Data Classification

Compliance with V6.1.1

V6.1.1 requires that regulated private data be encrypted while at rest. In our application, sensitive private data such as `usernames`, `names`, and `emails` are considered Personally Identifiable Information (PII). Currently, these fields are publicly accessible during the initialization of an organization. To address this, we could implement a hybrid encryption scheme at the point of organization creation to ensure secure handling of this data. However, as this information is not deemed critical within the context of our business logic, we have not implemented this feature. Within a session, sensitive information added (e.g., when a subject is added) is securely managed and remains private. The following outlines how our implementation aligns with V6.1.1, despite some minor setback:

Encryption of Data:

- **Document Storage:** Sensitive file content is encrypted before transmission to the server using `AES-GCM` (Advanced Encryption Standard - Galois/Counter Mode), providing both confidentiality and integrity protection. For instance:

```
plaintext_bytes = json.dumps(plaintext).encode("utf-8")
derived_key_bytes = bytes.fromhex(derived_key_hex)
nonce, ciphertext = symmetric.encrypt(derived_key_bytes, plaintext_bytes,
associated_data_bytes)
```

  The encrypted file content is then securely stored in the repository.

- **Encryption of Metadata:** Document metadata, including sensitive information such as the encryption key, is encrypted using a `secret_derived_key` derived from the `master_key` via `PBKDF2` (Password-Based Key Derivation Function). For example:

```
key_salt = os.urandom(16)
secret_derived_key = PBKDF2HMAC(
    algorithm=hashes.SHA256(), # output is 256 bits -> 32 bytes
    length=32,
    salt=key_salt,
    iterations=480000,
).derive(current_app.MASTER_KEY.encode("utf-8"))
key_nonce, encrypted_key = symmetric.encrypt(secret_derived_key, key, None)

document_metadata = {
    'name': name,
    'create_date': datetime.datetime.now().strftime("%d-%m-%Y %H:%M:%S"),
    'creator': username,
    "file_handle": file_handle_hex,
    "document_acl": document_acl,
    "deleter": None,
    "alg": alg,
    "key": encrypted_key.hex(),
    "key_salt": key_salt.hex(),
    "key_nonce": key_nonce.hex(),
}
```

- **Session Data Security:** Session-related data such as the `session_id` and `derived_key` are encrypted during transmission, ensuring confidentiality and integrity. This information can only be accessed by repository, is implemented using `AES-GCM` encryption and follows this structure:

```
encrypted_data = {
    'nonce': nonce.hex(),
    'ciphertext': ciphertext.hex()
}
```

Addressing CWE 311 - Missing Encryption of Sensitive Data: The Common Weakness Enumeration (CWE) 311 refers to the lack of encryption for sensitive data during storage or transmission. Our implementation mitigates this risk as follows:

- **Encryption Before Transmission:** All sensitive data, such as emails and document contents, is encrypted before transmission to the repository.
- **Encryption at Rest:** Sensitive data, including file content and metadata, is securely encrypted while stored.

Conclusion: Our implementation fully complies with V6.1.1 by ensuring that sensitive data is encrypted both during transmission and while at rest.

Compliance with V6.1.2

V6.1.2 requires that regulated health data be encrypted while at rest to maintain confidentiality and prevent unauthorized access.

Applicability of V6.1.2: Our application does not process, store, or transmit regulated health data.

Addressing CWE 311 - Missing Encryption of Sensitive Data: As health data is not present in our application, the specific risks associated with CWE 311 do not apply.

Compliance with V6.1.3

V6.1.3 requires that regulated financial data be encrypted while at rest to ensure confidentiality and prevent unauthorized access.

Applicability of V6.1.3: Our application does not process, store, or transmit regulated financial data.

Addressing CWE 311 - Missing Encryption of Sensitive Data: As financial data is not present in our application, the risks described by CWE 311 are not applicable.

3.2. Algorithms

Compliance with V6.2.1

V6.2.1 mandates that cryptographic modules fail securely and handle errors in a way that prevents Padding Oracle attacks.

Padding Oracle Attacks: This attack exploits vulnerabilities in block cipher modes like CBC (Cipher Block Chaining), where plaintext must be padded to match the cipher block size. If the application responds differently based on whether the padding is valid or invalid, an attacker can use this feedback as an "oracle" to deduce sensitive information, such as encryption keys or plaintext, through repeated trial-and-error attempts.

To prevent Padding Oracle Attacks:

1. Avoid cryptographic modes that rely on padding or ensure padding is validated securely.
2. Handle errors generically, without exposing cryptographic details.

Applicability of V6.2.1: Our application adheres to V6.2.1 by implementing cryptographic mechanisms that inherently mitigate the risk of Padding Oracle Attacks.

Evidence of Compliance:

1. **Use of AES-GCM:**
    - AES-GCM (Galois/Counter Mode) is utilized for all encryption and decryption operations. This mode does not require padding, as it operates on plaintext of arbitrary lengths.

- AES-GCM integrates encryption and integrity verification, ensuring tampered messages are rejected during decryption.

2. **Error Handling:**

   - Cryptographic errors are handled securely by catching exceptions and returning generic error messages to clients. For example:

     ```
     try:
         plaintext_bytes = symmetric.decrypt(
             bytes.fromhex(derived_key_hex),
             bytes.fromhex(nonce_hex),
             bytes.fromhex(ciphertext_hex),
             json.dumps(associated_data).encode("utf-8")
         )
     except InvalidTag:
         raise Exception(f"Error decrypting data for session {session_id} (InvalidTag)")
     except Exception as e:
         raise Exception(f"Error decrypting data for session {session_id} ({e})")
     ```

     If decryption fails (e.g., due to tampered ciphertext), the exceptions are handled securely without revealing cryptographic details. This prevents attackers from inferring the cause of failure.

3. **Fail-Secure Behavior:**

   - All cryptographic operations fail securely. Invalid data or incorrect keys result in decryption failure without compromising system integrity or exposing sensitive information.

Addressing CWE-310 - Cryptographic Issues: CWE-310 refers to cryptographic issues, including misuse of algorithms, insecure key management, and improper error handling. By utilizing AES-GCM and securely handling errors, our application adheres to best practices.

Conclusion: The cryptographic modules in the application comply with V6.2.1, ensuring resilience against Padding Oracle Attacks.

Compliance with V6.2.2

V6.2.2 requires the use of industry-proven or government-approved cryptographic algorithms and libraries. It also mandates avoiding deprecated algorithms like MD5 or SHA-1.

Applicability of V6.2.2: Our application complies with V6.2.2 by employing secure, modern cryptographic standards.

Evidence of Compliance:

1. **Use of AES-GCM:**

   - AES-GCM is used for encryption, ensuring both confidentiality and integrity. It is recognized as a secure and efficient standard.
   - Example:

     ```
     aesgcm = AESGCM(key)
     nonce = os.urandom(12)
     ciphertext = aesgcm.encrypt(nonce, plaintext, associated_data)
     ```

2. **Use of SHA-256:**

   - The application employs SHA-256 for hashing, offering strong resistance to collision and preimage attacks.
   - Example:

     ```
     hashlib.sha256(data.encode()).hexdigest()
     ```

3. **Elliptic Curve Cryptography (ECC):**
    - ECC is used for secure key exchange and digital signatures. Example:

      ```
      ec.ECDSA(hashes.SHA256())
      ec.ECDH()
      ```

4. **Avoidance of Deprecated Algorithms:**
    - No deprecated algorithms (e.g., MD5, SHA-1, DES, RC4) are used in the implementation.

Addressing CWE-327 - Use of a Broken or Risky Cryptographic Algorithm: By avoiding deprecated algorithms and employing secure standards, the application mitigates risks associated with CWE-327.

Conclusion: The application complies with V6.2.2 by using only secure and approved cryptographic algorithms.

Compliance with V6.2.3

V6.2.3 mandates the secure configuration of encryption initialization vectors (IVs), cipher configurations, and block modes.

Evidence of Compliance:

- **Initialization Vectors:**
    - AES-GCM uses a unique 12-byte IV (nonce) for every encryption process, ensuring randomness and preventing vulnerabilities like replay attacks.

      ```
      nonce = os.urandom(12)
      ciphertext = aesgcm.encrypt(nonce, plaintext, associated_data)
      ```

- **Cipher Configuration:**
    - AES-GCM ensures confidentiality and integrity without padding, avoiding vulnerabilities in other modes like CBC or ECB.
- **Encryption Key Strength:**
    - The application uses 256-bit AES keys, providing strong security and resilience against brute-force attacks.

Addressing CWE-326 - Inadequate Encryption Strength: By using AES-256 and securely generating IVs, the application adheres to cryptographic best practices.

Conclusion: The application complies with V6.2.3 through secure configuration of IVs, ciphers, and encryption keys.

Compliance with V6.2.4

V6.2.4 requires that random number generators, encryption or hashing algorithms, key lengths, rounds, ciphers, or modes can be reconfigured, upgraded, or swapped at any time to protect against cryptographic breaks.

Applicability of V6.2.4: Our application complies with this requirement through the following measures:

- **Random Number Generator:** The application uses `os.urandom` for generating random values, such as nonces for AES-GCM encryption. This can be easily swapped or upgraded by modifying the codebase.
- **Encryption and Hashing Algorithms:**

  - **Encryption:** AES-GCM is used for encryption processes. While runtime reconfiguration of ciphers or modes is not supported, the cryptographic library (`cryptography`) provides built-in support for multiple algorithms and modes. Future changes can be programmatically implemented.
  - **Hashing:** SHA-256 is used for integrity verification and signing. Like encryption algorithms, hashing algorithms can also be replaced programmatically as needed.

- **Key Lengths and Rounds:** The application employs 256-bit keys for AES-GCM encryption, ensuring robust security. These parameters are defined in the source code and can be updated if required.

Addressing CWE-326 - Inadequate Encryption Strength: By utilizing secure algorithms like AES-GCM and SHA-256 and adhering to configurable cryptographic practices, the application mitigates the risks of using inadequate encryption schemes.

Compliance with V6.2.5

V6.2.5 mandates avoiding known insecure block modes, padding modes, ciphers with small block sizes, and weak hashing algorithms unless required for backward compatibility.

Evidence of Compliance:

- **Insecure Block Modes:** The application uses AES-GCM, a secure and modern mode of operation that avoids the vulnerabilities of insecure modes like ECB.
- **Padding Modes:** AES-GCM does not rely on padding, mitigating vulnerabilities such as padding oracle attacks associated with modes like PKCS#1 v1.5.
- **Small Block Sizes:** The application uses AES with a modern block size, avoiding older ciphers like Triple-DES or Blowfish, which are vulnerable due to their smaller block sizes.
- **Weak Hashing Algorithms:** The application employs SHA-256 for hashing, ensuring strong resistance to collision and preimage attacks, and avoids deprecated algorithms like MD5 and SHA-1.

Addressing CWE-327 - Use of a Broken or Risky Cryptographic Algorithm: CWE-327 refers to the use of cryptographic algorithms that are considered broken or insecure due to vulnerabilities. By exclusively using secure algorithms like AES-GCM and SHA-256, the application mitigates risks associated with outdated or insecure cryptographic methods.

Conclusion: The application complies with V6.2.5 by using secure block modes, avoiding padding-related vulnerabilities, and implementing modern cryptographic algorithms with sufficient block sizes and hashing strength.

Compliance with V6.2.6

V6.2.6 requires that nonces, initialization vectors (IVs), and other single-use numbers are not reused with the same encryption key to prevent cryptographic vulnerabilities.

Evidence of Compliance:

- **Unique Nonce/IV Generation:** The application uses AES-GCM, which enforces the use of a unique IV for each encryption operation. These IVs are generated randomly using `os.urandom()`, ensuring both uniqueness and unpredictability.

Addressing CWE-324 - Use of a Key Reuse Issue: CWE-324 refers to cryptographic vulnerabilities that arise from reusing keys or associated values (e.g., IVs or nonces) during encryption. By ensuring that nonces and IVs are unique for every encryption operation, the application mitigates risks of key reuse and related vulnerabilities, such as the exposure of plaintext or encryption keys.

Conclusion: The application complies with V6.2.6 by generating unique and unpredictable nonces/IVs for each encryption operation, preventing vulnerabilities related to key reuse.

Compliance with V6.2.7

V6.2.7 requires that encrypted data be authenticated to ensure it is not altered by unauthorized parties.

Evidence of Compliance:

1. **Cipher Authentication with AES-GCM:** AES-GCM ensures both confidentiality and integrity. During decryption, AES-GCM verifies the authenticity of ciphertext and associated data, rejecting tampered or modified content.

```
aesgcm = AESGCM(key)
ciphertext = aesgcm.encrypt(nonce, plaintext, associated_data)
plaintext = aesgcm.decrypt(nonce, ciphertext, associated_data)
```

2. **Digital Signatures with ECC:** The application uses ECDSA (Elliptic Curve Digital Signature Algorithm) to sign data, ensuring that any unauthorized modification is detected.

```
server_pub_key.verify(
    bytes.fromhex(signature_hex),
    json.dumps(associated_data).encode("utf-8"),
    ec.ECDSA(hashes.SHA256())
)
```

3. **Key Derivation with HKDF:** After a secure key exchange using ECDH (Elliptic Curve Diffie-Hellman), the shared key is derived using HKDF (HMAC-based Key Derivation Function) to ensure secure and authenticated key derivation.

```
derived_key = HKDF(
    algorithm=hashes.SHA256(),
    length=32,
    salt=None,
    info=b'session key',
    backend=default_backend()
).derive(shared_key)
```

Addressing CWE-345 - Insufficient Verification of Data Authenticity: CWE-345 refers to the lack of sufficient mechanisms to verify the authenticity of data. By using AES-GCM for authenticated encryption, digital signatures for data integrity, and HKDF for authenticated key derivation, the application ensures that encrypted data remains authentic and tamper-proof.

Conclusion: The application complies with V6.2.7 by implementing authenticated encryption, digital signatures, and secure key derivation processes to ensure the integrity and authenticity of encrypted data.

Compliance with V6.2.8

V6.2.8 requires that all cryptographic operations be performed in constant time, without any "short-circuit" behavior in comparisons, calculations, or return paths. This ensures that sensitive information is not leaked through timing-based side-channel attacks.

Evidence of Compliance:

- **Use of AES-GCM for Encryption:** The application uses the AES-GCM (Galois/Counter Mode) algorithm, implemented via the Python `cryptography` library. This library ensures constant-time behavior for AES-GCM encryption and decryption operations. Example:

```
ciphertext = aesgcm.encrypt(nonce, plaintext_bytes, associated_data_bytes)
```

- **ECDSA for Signing:** Digital signatures are performed using ECDSA with SHA-256. The signature generation process is handled securely by the `cryptography` library, which ensures constant-time operations. Example:

```
response_signature = secret_key.sign(
    associated_data_bytes,
    ec.ECDSA(hashes.SHA256())
)
```

- **Secure Comparisons for Sensitive Data:** Sensitive data such as cryptographic keys, nonces, and derived keys are handled using secure library functions.
- **No Custom Cryptographic Implementations:** The application avoids custom cryptographic algorithms or implementations, which might introduce timing vulnerabilities, relying instead on industry-proven libraries.

Addressing CWE-385 - Covert Timing Channel: CWE-385 refers to vulnerabilities where sensitive information is inferred by analyzing variations in the time it takes for operations to complete. This is particularly relevant in cryptographic contexts, where even small timing discrepancies can reveal information about keys or plaintext.

The application adheres to cryptographic best practices by using established libraries for all cryptographic operations.

Conclusion: The cryptographic operations in the application comply with V6.2.8.

3.3. Random Values

Compliance with V6.3.1

V6.3.1 requires that all random numbers, random file names, random GUIDs, and random strings are generated using a cryptographically secure random number generator (CSPRNG) when these values must be non-guessable by an attacker.

Applicability of V6.3.1: While our application requires random values in limited instances, all cases adhere to best practices by leveraging cryptographically secure modules for generation.

Evidence of Compliance:

1. **Use of `os.urandom()`:**
    - For generating random values like cryptographic keys and salts, the application utilizes `os.urandom()`, a cryptographically secure random number generator (CSPRNG).
    - Example implementation:

      ```
      key = os.urandom(32)
      ```

Addressing CWE-338 - Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG): By exclusively using `os.urandom()`, the application avoids weak pseudo-random number generators. This ensures random values are derived from a cryptographic module with a strong entropy source, providing secure, non-guessable values for cryptographic operations.

Compliance with V6.3.2

V6.3.2 mandates that random GUIDs are created using the GUID v4 algorithm with a Cryptographically Secure Pseudo-Random Number Generator (CSPRNG).

Applicability of V6.3.2: Our application does not generate random GUIDs and does not rely on GUIDs for identifier generation. Therefore, this requirement is not applicable.

Compliance with V6.3.3

V6.3.3 requires that random numbers maintain proper entropy levels even under high system load, or the application degrades gracefully in such circumstances.

Applicability of V6.3.3: The application uses `os.urandom()` for generating random values, ensuring that they are sourced from a cryptographically secure and entropy-rich pool. While our application does not currently implement explicit mechanisms for graceful degradation under entropy exhaustion, the use of `os.urandom()` mitigates most entropy-related risks.

Evidence of Compliance:

1. **Secure Entropy Source:**
   - `os.urandom()` generates random values by drawing from `/dev/urandom`, a secure and dynamically replenished entropy pool on Unix-like systems.
   - Example implementation:

     ```
     nonce = os.urandom(12)
     ```

2. **Non-Blocking Behavior:**
   - Unlike `/dev/random`, `os.urandom()` does not block during entropy depletion. This ensures that the application continues functioning without performance degradation under high load.
   - While `os.urandom()` minimizes blocking risks, extreme load could reduce entropy availability, potentially affecting randomness quality over time.

Addressing CWE-334 - Insufficient Entropy in PRNG: The application ensures sufficient entropy by relying on `os.urandom()`, which maintains a secure entropy pool. However, measures can further enhance system robustness under heavy load.

Recommendations for Graceful Degradation: To address potential entropy depletion risks during extreme load, we propose the following enhancements:

- **Monitoring and Alerts:** Implement real-time monitoring of entropy levels, with alerts triggered when they fall below a safe threshold.
- **Fallback Mechanisms:** Introduce fallback procedures, such as temporarily using pre-generated random values or limiting cryptographic operations during low-entropy scenarios.
- **System Load Management:** Apply load balancing, rate limiting, or queueing strategies to distribute load evenly and prevent entropy source exhaustion.

Conclusion: The application complies with V6.3.3 by leveraging `os.urandom()` to ensure cryptographically secure random value generation with adequate entropy under normal conditions. While current measures do not explicitly address entropy exhaustion under extreme load, implementing the recommended enhancements will further strengthen compliance and ensure continued system reliability and security.

3.4. Secret Management

Compliance with V6.4.1

V6.4.1 mandates the use of a secrets management solution, such as a key vault, to securely create, store, control access to, and destroy secrets.

Applicability of V6.4.1: Currently, our application does not utilize a dedicated secrets management solution, such as a key vault. Instead, it relies on the operating system's security layer to store keys and session information in memory. This approach leverages the OS's built-in security mechanisms, such as memory isolation and access control, to safeguard sensitive data. While this design decision provides a certain level of protection, it does not fully meet the requirements outlined in V6.4.1 for a dedicated secrets management solution.

Evidence of Compliance:

1. **Use of Operating System Security:**

- Keys and session information are stored in memory, and access is controlled by the operating system's security mechanisms. These include memory isolation and enforced access control policies to ensure that only authorized processes can access sensitive data.

2. **Adherence to CWE-798 (Use of Hard-coded Credentials):**

   - The application adheres to best practices by avoiding hardcoding of secrets in source code. This minimizes the risk of exposing sensitive information, maintaining confidentiality and reducing attack surfaces.

Conclusion: Although the current implementation relies on operating system-level protections and avoids hardcoded secrets, it does not fully align with V6.4.1's requirement for a dedicated secrets management solution. To achieve full compliance, the application should integrate a dedicated key vault or secrets management solution to securely handle the lifecycle of sensitive information.

Compliance with V6.4.2

V6.4.2 requires that key material is not exposed to the application but instead uses an isolated security module like a vault for cryptographic operations.

Applicability of V6.4.2: Our application does not use a separate isolated security module (e.g., a hardware security module or vault) for cryptographic operations. Instead, keys and session information are stored and processed in memory by the application, relying on the operating system to isolate and protect the data. While this setup does not involve a dedicated security module, we believe the operating system's inherent security features—such as memory isolation and access control—adequately protect sensitive data.

Evidence of Compliance:

1. **Key Material Handling:**

   - Cryptographic keys are handled in memory, where access is restricted by the operating system's memory protection and access control mechanisms. This ensures that unauthorized processes cannot easily access sensitive data.
   - The application does not expose key material directly to any external modules or services, avoiding risks associated with exposing keys to potentially less secure environments.

2. **No Use of Hardcoded Secrets:**

   - As with V6.4.1, the application ensures that no secrets are hardcoded in the source code, further protecting against inadvertent exposure of key material or sensitive data.

4. Conclusion

This project delivers a secure and robust repository for managing organizational documents, with a strong emphasis on encryption, authentication, and secure workflows. Its modular architecture facilitates seamless integration of additional features and adaptability to evolving security needs. By employing modern cryptographic standards such as `ECDSA` and `AES-GCM`, the implementation adheres to industry best practices and compliance requirements. While the current focus is on core functionalities, future enhancements could include multi-factor authentication and integration with external identity providers. Overall, this project underscores the critical role of secure design in modern software systems, ensuring both practical utility and robust protection against emerging threats.