deti universidade de aveiro
departamento de eletrónica,
telecomunicações e informática

Delivery 3 - SIO

Security of Information and Organizations Project
Professor: João Paulo Barraca

João Viegas nº113144, Jorge Domingues nº113278, João Monteiro nº114547

Abstract

This report presents the implementation and evaluation of a secure document repository system. The system leverages advanced cryptographic techniques to ensure the confidentiality, integrity, and authenticity of documents shared within an organizational structure. The primary objective of this project is to develop a robust repository that facilitates the management of users and organizations while maintaining a high level of security.

The repository enables subjects (users) to interact with the system by creating secure sessions, which establish a trusted context for communication between the client and the repository. These sessions are a cornerstone of the system's security, as they allow users to authenticate and operate within predefined boundaries. Roles are assigned to subjects, defining their levels of authorization and determining their specific permissions within the repository. This role-based access control (RBAC) mechanism ensures that sensitive actions and data access are restricted to authorized users only.

Additionally, the system supports a wide range of functionalities, including document creation, retrieval, sharing, and deletion. Access control lists (ACLs) are utilized to enforce permissions at a granular level, ensuring that only authorized roles can perform operations on specific resources.

To assess the security of the implementation, this report also includes an analysis of the V3: Session Management guidelines from the OWASP Application Security Verification Standard (ASVS). This evaluation focuses on the session handling mechanisms employed in the system, examining their alignment with best practices to prevent common vulnerabilities such as session fixation, hijacking, and replay attacks. The analysis provides insights into the robustness of the session management strategies implemented in this project, highlighting areas of strength and opportunities for improvement.

Keywords: *repository; documents; cryptography; organizations; subjects; sessions; roles; permissions.*

1. Entities Overview

This section provides a formal description of the main entities that comprise the secure document repository system. Each entity plays a critical role in ensuring the system's functionality, security, and integrity.

1.1. Organizations

Organizations serve as the primary structure within the system, acting as containers for managing users, permissions, and documents. They are integral to establishing boundaries and policies that govern interactions within the repository.

Key Attributes:

- **name:** A unique identifier for the organization.
- **creator:** The user who created the organization.
- **subjects:** A mapping of users (subjects) to their roles, statuses (e.g., active or suspended), and public keys.

- **documents:** A collection of documents associated with the organization.
- **roles:** A definition of roles available within the organization and their states (active or suspended).
- **acl:** An access control list defining the permissions assigned to each role.

Associated Functions:

- Creating and listing organizations.
- Managing roles and permissions within the organization.
- Associating users and documents with the organization.

1.2. Subjects (Users)

Subjects represent the users of the system who interact with organizations and documents. Each subject is uniquely identified and may have various roles and permissions within the repository.

Key Attributes:

- **username:** A unique identifier for the user.
- **name:** The user's full name.
- **email:** The user's email address.
- **public_key:** The user's public key for authentication and encryption.
- **status:** The user's status (*up* for active, *down* for suspended).
- **roles:** A list of roles assigned to the user within an organization.

Associated Functions:

- Adding, activating, or suspending users.
- Listing users and their roles.
- Assigning specific permissions to users.

1.3. Sessions

Sessions define the secure context for interactions between a user and the system. They are essential for authentication and authorization, ensuring that all actions occur within a verified and trusted framework.

Key Attributes:

- **session_id:** A unique identifier for the session.
- **organization:** The organization associated with the session.
- **username:** The user associated with the session.
- **created_at:** The timestamp for when the session was created.
- **valid_until:** The expiration timestamp for the session.
- **public_key:** The public key used in the session.
- **number:** A counter to protect against replay attacks.
- **roles:** The list of roles assumed by the user during the session.

Associated Functions:

- **Session Creation:** Validates user credentials, assigns a session ID, and establishes an expiration time.
- **Session Validation:** Ensures that the session is active, linked to a valid user, and within its validity period before any operation.
- **Role Management:** Dynamically associates or removes roles during a session, adjusting the user's permissions as needed.
- **Replay Protection:** Utilizes the *number* attribute to verify request order and block duplicate or out-of-sequence operations.
- **Expiration Management:** Automatically removes sessions once they exceed their validity period to prevent unauthorized access.

Security Considerations:

- **Encryption:** Communication between the client and server is encrypted using the user's public key, ensuring data confidentiality.
- **Account Status Verification:** Sessions are invalidated if the associated user is suspended or removed from the organization.
- **Role-based Access Control (RBAC):** All session actions are validated against the user's roles and the permissions defined in the organization's ACLs.

1.4. Session File

The **Session File** is a critical component that stores specific session-related information. Unlike the broader concept of a session, this file only contains essential attributes required for cryptographic operations and replay protection. It is designed to be lightweight, secure, and easily accessible for validating active sessions.

Importantly, the **Session File is stored on the client side**, ensuring that session-specific data is localized to the user's environment.

1.4.1. Stored Attributes

- **session_id:** A unique identifier for the session. It is used to associate the file with a specific session context.
- **private_key:** A cryptographic private key stored in its encoded form. This key is utilized for encrypting and decrypting or signing operations within the session.
- **number:** A counter used to prevent replay attacks. It ensures that operations are executed in the correct sequence by incrementing with each authorized request.

1.4.2. Characteristics

- **Simplicity:** The file structure is deliberately minimal, focusing on the essential elements needed for security.
- **Security:** The *private_key* is sensitive and must be protected from unauthorized access. The file is stored on the client side in a secure, encrypted location to prevent tampering or exposure.
- **Replay Protection:** The *number* attribute ensures that requests using this session file are unique and sequential, mitigating replay attacks.

1.4.3. Example Content

Below is an example of the content of a **Session File** in JSON format:

```
{
    "session_id": "b0f6fc4df29d86b719d66458cf8e811c",
    "private_key": "2d2d2d2d2d424547494e205052495641544520b45592d2d2d2d2d0a4d49...",
    "number": 4
}
```

1.4.4. Purpose

The **Session File** ensures that cryptographic operations and replay protection are managed effectively for each session. By avoiding unnecessary information, it focuses solely on attributes critical for session validation and security.

1.4.5. Security Considerations

- **Encryption:** The file must be encrypted during storage to prevent unauthorized access to the private key.
- **Access Control:** Only authorized components of the client system should have permission to read or update the file.
- **Integrity:** A hashing mechanism should be used to detect unauthorized modifications to the file.
- **Lifecycle Management:** The file must be deleted or invalidated when the session ends or the key is rotated.

The **Session File**, stored on the client side, is a foundational element of the secure document repository system, balancing simplicity with robust security to support safe and efficient session management.

1.5. Documents

Documents represent the core data stored within the repository. They are divided into public metadata and private, encrypted content to ensure security and controlled access.

Key Attributes:

- *Public Metadata:*
  - **name:** The document's name.
  - **document_handle:** A unique identifier for the document.
  - **create_date:** The creation timestamp of the document.
  - **creator:** The user who created the document.
  - **acl:** Access control lists specifying permissions for the document.
  - **deleter:** If deleted, indicates who removed the document.
- *Private Data:*
  - **key:** The encryption key for the document.
  - **iv:** The initialization vector used in encryption.
  - **tag:** The integrity tag generated during encryption.
  - **alg:** The encryption algorithm (e.g., AES-GCM).

  Encrypted documents are stored in a secure directory structure on the server, ensuring unauthorized access is prevented.

Associated Functions:

- Adding, listing, retrieving, and removing documents.
- Managing ACLs to ensure secure access to documents.
- Encrypting and decrypting document content using SHA-256 for integrity verification and AES-GCM for confidentiality.

Roles

Roles define levels of authorization assigned to users within an organization. They determine what actions a user can perform by associating permissions with each role.

Key Attributes:

- **Role Name:** Identifies the role (e.g., *Manager*).
- **State:** Indicates whether the role is active (*up*) or suspended (*down*).

Default Role: Manager

- The **Manager** role was intentionally implemented as the default role in our system design to provide a consistent mechanism for identifying the subject responsible for managing the organization. This decision ensures that every organization has at least one user with high-level administrative capabilities.
- Managers have the highest level of authorization and are permitted to perform critical operations such as managing users, roles, and documents.
- The Manager role cannot be suspended if it is the last active Manager in the organization, ensuring organizational integrity.

Associated Functions:

- Creating, suspending, reactivating, or listing roles.
- Managing permissions associated with roles.
- Assigning or removing roles from users.

1.6. Permissions

Permissions define specific actions that can be performed within the system. They are linked to roles to control access to resources and operations.

Examples of Permissions:

- **DOC_READ:** Permission to read documents.
- **DOC_NEW:** Permission to create new documents.
- **SUBJECT_NEW:** Permission to add new users.
- **ROLE_ACL:** Permission to modify ACLs.

Associated Functions:

- Adding or removing permissions from roles.
- Verifying permissions before executing sensitive operations.

1.7. Access Control Lists (ACLs)

Access Control Lists (ACLs) are used to define fine-grained permissions for roles within an organization. They play a critical role in ensuring that only authorized users can perform specific operations on documents or other resources.

Key Features:

- ACLs are associated with both the organization and individual documents.
- Each ACL entry specifies a permission (e.g., *DOC_READ*) and a list of roles allowed to perform the associated action.

- ACLs ensure that security policies are enforced consistently across the system.

Default Permissions for Managers:

- Managers are granted all permissions by default, including those related to organizational and document management.
- Managers can modify ACLs to delegate permissions to other roles as necessary.

1.8. Role Management

Role management defines and controls the access levels and user permissions within an organization. It ensures that users have appropriate permissions to perform actions while preventing unauthorized access. In this system, roles define the scope of what users can and cannot do, making it critical to manage them thoughtfully.

The implemented **Role Management** module allows for the creation, modification, suspension, reactivation, and assignment of roles to users. It also facilitates precise access control through permissions, ensuring that roles align with the organizational hierarchy and the principle of least privilege.

This module integrates seamlessly with the session management and encryption systems to maintain security and traceability.

Roles are stored in the **organizations** collection, in the **roles** field, where each role has an associated **status** (UP ou DOWN). **ACLs** are maintained on the **acl** field, where an association is made between permissions and roles.

The following functionalities are available within the Role Management Module:

1.8.1. Role Creation (/role/add):

This endpoint, which is attainable by making a **POST** request, allows for the creation of a new role within an organization. This is crucial for expanding the scope of access control and defining specific responsibilities for users.

The system verifies that, for the current session, the **ROLE_NEW** permission is included in its assigned roles.

Users with the **ROLE_NEW** permission can execute this command and, therefore, add new roles to the system. A validation is made to verify that the role that is being added does not already exist. Every new role is initialized with the status **"UP"** as default.

1.8.2. Role Drop (/role/drop):

This endpoint allows a user to drop (remove) a specific role from their current active session. This is useful when a user no longer needs the permissions associated with a role during a session.

The system ensures that the user exists and that the role that we are looking to remove is assigned to the current session. If both conditions are verified, the specified role is removed from the user's session by using the **$pull** operation, which removes the role from the **roles** array in the **sessions_collection**.

1.8.3. Role Suspension (/role/suspend):

This endpoint is equally attainable by making a **POST** request. It is used to temporarily disable a specific role within an organization. This is essential for maintaining proper access control, especially when a role is no longer required or under review.

An verification is made, beforehand, to make sure that the user that made the request is authenticated and active. On the other hand, the current session must have the **ROLE_DOWN** permission to suspend a role.

The role that is being suspended must obviously exist in the system and the **MANAGER** role can never be suspended. If the current status of the targeted role is already **DOWN**, the role cannot be suspended, and an error is thrown. Otherwise, the status of the role will be updated to **DOWN**.

1.8.4. Role Reactivation (/role/reactivate):

This endpoint is used to restore a previously **suspended** role ("down") to an **active** status ("up"). This ensures that roles essential to organizational functions can be re-enabled when needed.

An verification is made, beforehand, to make sure that the user that made the request is authenticated and active. On the other hand, the current session must have the **ROLE_UP** permission to reactivate a previously suspended role.

The role that is being reactivated must exist and if the status of the targeted role is already **UP**, then an error is thrown, since the role is already active. Otherwise, the status of the role will be updated to **UP**.

1.8.5. Role Assume (/role/assume):

This endpoint, attainable with a **POST** request, allows a user to assume a specific role within their active session. This functionality is essential for dynamically granting access to certain actions or resources during a session based on the user's role.

An verification is made, beforehand, to make sure that the role **exists** and is **active**. Also, that role cannot already be assigned to the session. The role must be explicitly **assigned to the subject**. A user cannot assume roles that are not directly linked to them.

If all this verifications are checked, the requested role is added to the user's session.

1.8.6. Adding a Permission (/permission/add):

In the system it is possible to grant a permission to a specific role or assign a role to a user. It ensures that roles or users are provided the required permissions to perform their tasks while maintaining access control.

The current session must have the **ROLE_MOD** permission to be able to modify roles or permissions. The system also confirms that the specified role exists.

In case a permission is being added to a role, the session must have the **ROLE_ACL** permission to do so. Afterwards, a confirmation that the role is **active** and that the role does not already have the permission is made, to ensure that everything works as expected.

On the other hand, if a role is being assigned to the user, the user must exist, must be active and it should not have that role already assigned. All this confirmations are made, while assigning a role to an user.

1.8.7. Remove Permission (/permission/remove):

This endpoint facilitates the removal of a permission from a role or the unassignment of a role from a user. This allows administrators to dynamically adjust user or role access, ensuring security and adherence to the principle of least privilege.

The current session must have the **ROLE_MOD** permission to be able to modify roles or permissions. The system also confirms that the specified role exists.

If removing a permission from a role, the session must confirm that the permission is currently associated with the role.

On the other hand, if removing a role from a user, the user must exist, and the specified role must be assigned to him. If the role being removed is **MANAGER**, a confirmation must be made that this is not the last existent **MANAGER** in the system, since there always needs to exist one, to ensure the correct functioning of the system.

1.8.8. Change ACL (/document/acl):

This endpoint endpoint allows for the modification of a document's **Access Control List (ACL)**. It provides the functionality to add or remove permissions for specific roles on a document, thereby controlling which roles can perform specific actions on the document.

The system must have the **DOC_ACL** permission to change the ACL. Based on the signal provided permissions can be added or removed for a specified role.

Also, a verification is made to ensure that the document exists and that the permission being added is part of the document's ACL.

1.8.9. List Roles (/role/list):

The purpose of this endpoint is providing a list of roles currently associated with a user's active session. This is useful for users to review their active permissions and access rights.

It extracts all roles, from the **roles** array stored in the current **session**.

1.8.10. List Role Subjects (/role/list_subjects):

This endpoint retrieves a list of subjects (users) associated with a specific role within an organization. This helps administrators view and manage which users are assigned to a given role.

No particular permissions are necessary to make this operation. For the given role, the system searches for users with that role, retrieving their **username** and **status**. If no subjects are returned, an error is thrown.

1.8.11. List Subject Roles (/subject/list_roles):

This endpoint retrieves all the roles assigned to a specific subject (user) within an organization. This helps in understanding a subject's permissions and access levels in the organizational hierarchy.

A verification is made to ensure that the subject exists within the organization and all roles assigned to the subject are retrieved.

1.8.12. List Role Permissions (/role/list_permissions):

This method provides a comprehensive list of all permissions assigned to a specific role within an organization. This helps administrators understand the scope and access rights granted to a role.

No particular permissions are necessary to make this operation. All permissions related to the given role are fetched, whether they are on the **Organizational Access Control Lists (ACLs)** or on **Document-specific ACLs**.

1.8.13. List Permission Roles (/permission/list_roles):

This method retrieves all roles associated with a specific permission within an organization. It enables administrators to view which roles have been granted access to a particular action or resource.

No particular permissions are necessary to make this operation. A verification is made to ensure that the provided permission exists in the organization's ACL. Afterwards, all roles assigned to the specified permission are gathered.

2. Overall Implementation Decisions

2.1. Encryption

- **Asymmetric Cryptography**
  - To securely exchange keys, we used the **Elliptic Curve Cryptography (ECC)** approach. More specifically, we used the **ECDH (Elliptic Curve Diffie-Hellman)** protocol to derive symmetric keys to encrypt communication.
  - In this case, each involved party generates a unique pair of keys, securely storing the private key and forwarding the public one to the other party.
  - Using the **ECDH** protocol, the private key of one party combined with the shared public key of the other party derive a shared secret, ensuring that only the parties involved can compute it.
  - This shared secret is passed through **HKDF (HMAC-based Extract-and-Expand Key Derivation Function)**. By combining the secret with a random **salt**, **HKDF** generates a robust and unique symmetric encryption key for each session, even when the same key pair is reused.

- **Symmetric Cryptography**
  - To ensure confidentiality and integrity during communication, we employed **AES (Advanced Encryption Standard)** in **GCM (Galois/Counter Mode)**.
  - **AES-GCM** relies on a (unique) nonce for each encryption operation, which ensures different results for multiple encryptions with the same data. This makes communication more secure, since it prevents attackers from identifying patterns in the encrypted data.
  - Additionally, an **authentication tag** is produced that is used to verify data integrity. This ensures that any unauthorized modifications are straight away identified.
  - Furthermore, a 256-bit key derived from the shared secret is used in the encryption process, which, as mentioned before, ensures that the encryption key is unique for each session.

- **Digital Signatures**
  - To guarantee authenticity and integrity of exchanged messages, we utilized digital signatures based on the **ECDSA (Elliptic Curve Digital Signature Algorithm)**. This allows a verification of the sender identity while ensuring that the message has not been altered.
  - When a message is sent, the sender first computes a **SHA-256 hash of the message**. Using their private key, the sender signs the hash with the **ECDSA algorithm**. The resulting signature is then appended to the message before transmission.
  - Upon receiving the message, the receiver extracts the signature and computes the **SHA-256 hash** of the received message. Using the sender's public key, that was shared with him, the receiver verifies the signature against the computed hash. If the verification succeeds, the recipient can be confident that the message was indeed sent by the claimed sender and that it has not been tampered.
  - This mechanism effectively prevents **MiTM (Man-in-the-Middle)** attacks.
- **Subject Key Usage for Encryption and Decryption**
  - The public keys provided during the creation of subjects are used for encrypting and decrypting messages within the repository for commands that require session-based operations. These keys work in conjunction with the repository's private key.
  - On the client side, messages pertaining to such commands are encrypted and decrypted using the repository's public key and the subject's private key.

2.2. Attack Prevention

One of the requirements that was necessary to achieve a good implementation was to make sessions robust to the following 4 attacks:

- **Eavesdropping**
- **Impersonation**
- **Manipulation**
- **Replay**

In addition, we also defined a session lifetime of 1 hour and an inactivity timeout of 5 minutes, which means that if a session is not used it will be deleted, mitigating potential security vulnerabilities.

2.2.1. Eavesdropping

With the usage of **AES-GCM** for securely encrypting information and **ECC** via the **ECDH protocol**, the system can efficiently encrypt communication and allow secure key exchange, which deny attackers from accessing the content that is exchanged. This allows the system to defend itself from **eavesdropping** attacks. (See 2.1 to understand how these protocols function).

2.2.2. Impersonation

To prevent impersonation, the system employs robust authentication which is enforced using private-public key pairs. When an user attempts to perform any operation, they sign the request with their private key, which acts as proof of identity. Following this, the repository validates that signature by comparing it with the user's public key, which is known, ensuring the request comes from the claimed user.

2.2.3. Manipulation

To ensure data integrity, the system employs **digital signatures**, using the **ECDSA algorithm**. Furthermore, the use of **authentication tags** generated by **AES-GCM** also helps prevent this type of attacks. With these methods, the system ensures that any unauthorized modification to the data is immediately detectable, protecting the system from manipulation attempts.

2.2.4. Replay

To prevent replay attacks, each session tracks a **unique counter** (`number`, which corresponds to the counter used in messages sent in our project) that increments with every message sent. When a message is received, the counter is compared with the last-known counter. This allows the repository to verify if it just received a repeated or out-of-order message, rejecting it in either case. The **unique counter** is only used to defend against attacks targeting commands that require a session. For commands that do not require a session, the counter is not utilized, as some commands will automatically fail if the same command is sent again. For commands using an anonymous API, the defense against replay attacks differs. The creation time of the message is included along with the command information. The repository then verifies if the sum of the message creation time and one minute is greater than the current time. If it is, this means the message was sent within one minute and is considered valid. However, if the repository finds that the sum of the creation time and one minute is less than the current time, it indicates the message was sent significantly later than its creation time. In this case, the message is identified as a replay attack and the command is canceled.

2.3. Data Persistence

To collect and store our data, we opted for a non-relational database. More specifically, the choice was **MongoDB**, a document-oriented database. Each entity in our repository (organizations, documents, subjects, and sessions) is represented with a separate collection.

For documents, we used **two distinct collections**: one to store the public information of each document and another to handle the restricted metadata of each one.

2.4. Decision on ROLE_ACL

Due to the uncertainty regarding the interpretation of the `ROLE_ACL`, the group decided that, for the `rep_add_permission` command to add permissions to a role, it would also be necessary to have the `ROLE_ACL` permission.

2.5. Encryption and Decryption Process in the Client-Server Communication

Most of the encryption and decryption logic has been abstracted into the `KeysGenerator.py` file. The repository generates an elliptic key pair, and all clients have access to the repository's public key.

The `encryptMessageWithSession` function is used to encrypt the information (message) that the client intends to send to the server. First, the `session_file` is read. This file contains a dictionary with the session identifier (`session_id`), the subject's private key, and the number of messages exchanged in this session (`number`). When sending the message to the repository, the `session_id` is included along with the information to be transmitted. The subject's private key is converted to bytes, and the message number (`number`) is added to the message. The subject's private key is then combined with the repository's public key to generate a `shared_key`, which is used to derive a `derived_key` using AES encryption (with `salt` and the `hashes.SHA256()` algorithm). This `derived_key` is subsequently used to encrypt the message. Before encryption, the message is augmented with its creation date.

The encryption process uses AES in `GCM` mode (`Cipher(algorithms.AES(derived_key), modes.GCM(nonce))`). Once the message is encrypted, it is signed, generating a digest using `hashes.Hash(hashes.SHA256())` and a signature with `ec.ECDSA(Prehashed(hashes.SHA256()))`. The signature ensures that the message was sent by the client.

The encrypted message produces a `ciphertext`, which is combined with the following elements: `nonce`, `tag` (generated by `encryptor.tag`), `salt`, `digest`, `signature`, the subject's public key, and the `session_id`. All these elements are converted to hexadecimal strings and organized in a dictionary, which is sent to the repository.

```python
# client.py line 87
def encryptMessageWithSession(message, session):
    logger.debug(f"Encrypting message with session: {message}")
    state = load_state()
    session_id=session["session_id"]
    private_key = bytes.fromhex(session["private_key"])
    number=session["number"]
    message["number"] = number
    server_public_key = serialization.load_pem_public_key(state["REP_PUB_KEY"].encode())
    subject_private_key = load_private_key_from_bytes(private_key)
    ciphertext, nonce, tag, salt = encrypt_data(json.dumps(message).encode(),
    subject_private_key, server_public_key)
    signature, digest = sign_data(json.dumps(message).encode(), subject_private_key)
    cipherMessage = {
        "ciphertext": ciphertext.hex(),
        "nonce": nonce.hex(),
        "tag": tag.hex(),
        "salt": salt.hex(),
        "signature": signature.hex(),
        "digest": digest.hex(),
        "session_id": session_id,
    }
    logger.debug(f"Message encrypted successfully: {cipherMessage}")
    return cipherMessage
```

Listing 1: Function to encrypt a session-based message and prepare it for secure transmission.

```python
# KeysGenerator.py line 87
def encrypt_data(data, private_key, peer_public_key):
    shared_key = private_key.exchange(ec.ECDH(), peer_public_key)

    salt = os.urandom(16)
    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        info=b"handshake data",
    ).derive(shared_key)

    nonce = os.urandom(12)
    cipher = Cipher(algorithms.AES(derived_key), modes.GCM(nonce))
    encryptor = cipher.encryptor()
    ciphertext = encryptor.update(data) + encryptor.finalize()

    return ciphertext, nonce, encryptor.tag, salt
```

Listing 2: Decrypting and validating a session-based message on the repository side

```python
# client.py line 1109
def rep_add_role(session_file, role):
    logger.debug("rep_add_role: Adding role")
    try:
        logger.debug(f"Opening session file '{session_file}'")
        with open(session_file, "r") as file:
            session = json.loads(file.read())
            session_id=session["session_id"]

        data = {
            "session": session_id,
            "role": role
        }
        state = load_state()
        repo_address = state.get("REP_ADDRESS", "localhost:5000")
        logger.debug(f"Repository address: {repo_address}")

        request_data = encryptMessageWithSession(data, session)
        logger.debug("Sending add role request to repository")
        response = requests.post(f"http://{repo_address}/role/add", json=request_data)
```

Listing 3: Example of sending a message with a session file; The information is encrypted before being transmitted.

On the repository side, the `decryptDataToDataWithSession` function is responsible for decoding the received message. First, all dictionary values are converted back to bytes (using `bytes.fromhex(data[])`) except for the `session_id`, which remains a string. The `session_id` is then used to locate the corresponding session in the MongoDB database, which contains the subject's public key. This public key, combined with the repository's private key, generates the same `shared_key` and derives the `derived_key`. Using this AES key along with the `nonce` and `tag`, the encrypted message (`ciphertext`) is decrypted to obtain the plaintext.

After decryption, the message's authenticity is verified using the subject's public key, the `signature`, and the `digest`. If the signature is invalid, the repository returns an error to the client. If valid, the decoded message is converted back to its original format (using `json.loads(plaintext.decode())`).

As a security measure, the message number (`number`) is checked against the number stored in the repository session. If they do not match, it is considered a replay attack, and an error message is returned to the client. If the number is valid, it is removed from the message.

```python
# repository.py line 107
def decryptDataToDataWithSession(data):
    ciphertext = bytes.fromhex(data["ciphertext"])
    nonce = bytes.fromhex(data["nonce"])
    tag = bytes.fromhex(data["tag"])
    salt = bytes.fromhex(data["salt"])
    signature = bytes.fromhex(data["signature"])
    digest = bytes.fromhex(data["digest"])
    session_id = data.get("session_id")
    session = sessions_collection.find_one({"_id": session_id})
    subject_public_key=serialization.load_pem_public_key(session["public_key"].encode())

    try:
        plaintext = decrypt_data(ciphertext, nonce, tag, salt, repository_private_key,
    subject_public_key)
        if verify_signature(signature, digest, subject_public_key):
            data = json.loads(plaintext.decode())
    except InvalidSignature:
        return None, 1
    except Exception as e:
        return None, 2
    if session["number"] == data["number"]:
        data.pop("number")
    else:
        return None, 5
    return data, subject_public_key
```

Listing 4: Function to decrypt a session-based message; verify its signature; validate the session counter.

```python
# KeysGenerator.py line 106
def decrypt_data(ciphertext, nonce, tag, salt, private_key, peer_public_key):
    shared_key = private_key.exchange(ec.ECDH(), peer_public_key)

    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        info=b"handshake data",
    ).derive(shared_key)

    cipher = Cipher(algorithms.AES(derived_key), modes.GCM(nonce, tag))
    decryptor = cipher.decryptor()
    plaintext = decryptor.update(ciphertext) + decryptor.finalize()

    return plaintext
```

Listing 5: Function to decrypt data using AES with a derived key; the key is generated from an elliptic curve shared secret; the decryption validates the integrity of the message.

```python
# KeysGenerator.py line 149
def verify_signature(signature, digest, public_key):
    try:
        public_key.verify(
            signature,
            digest,
            ec.ECDSA(Prehashed(hashes.SHA256()))
        )
        return True
    except InvalidSignature:
        print("Error in Signature")
        return False
```

Listing 6: unction to verify a digital signature; ensures the integrity and authenticity of the message digest using the subejct public key.

```
# repository.py line 820
@app.route("/role/add", methods=["POST"])
def add_role():
    try:
        data = request.get_json()
        if not data:
            return json.dumps({"error": "No JSON data received"}), 400

        data, subject_public_key = decryptDataToDataWithSession(data)
        session = getSession(data.get("session"))
        if session is None:
            return json.dumps({"error": "Session not stored"}), 400
```

Listing 7: Example of a route that decrypts incoming data; validates the session; handles missing JSON data or invalid sessions.

The `getSession` function then checks if the session has expired. If expired, the session is removed from the database, and an error message is sent to the client. Otherwise, the repository executes the logic corresponding to the message's intended operation.

```
# repository.py line 261
def getSession(session_id):
    session_data = sessions_collection.find_one({"_id": session_id})
    if session_data:
        valid_until = datetime.datetime.fromisoformat(session_data["valid_until"])
        organization = organizations_collection.find_one({"name": session_data["
    organization"]})
        if valid_until < datetime.datetime.now():
            sessions_collection.delete_one({"_id": session_id})
            return None
        if organization['subjects'][session_data["username"]]["status"] == "down":
            return None
        return session_data
    else:
        return None
```

Listing 8: Function to retrieve a session by ID; checks if the session is expired or the user's status is inactive before returning the session data.

If any error occurs during execution, the repository returns a specific error message. Otherwise, a response message is generated, indicating the success of the operation or containing the requested information (e.g., confirmation of session creation).

In the response, the message number is incremented by 1 (`number + 1`) before being included. This message is encrypted using the same process described earlier, but with the repository's private key and the subject's public key used to derive the AES key. After encryption, the number stored in the repository session is incremented by 1.

```
# repository.py line 107
def encryptMessageWithSession(message, subject_public_key, session_id):
    session = sessions_collection.find_one({"session_id": session_id})
    number = session["number"]
    message["number"] = number + 1
    ciphertext, nonce, tag, salt = encrypt_data(json.dumps(message).encode(),
    repository_private_key, subject_public_key)
    signature, digest = sign_data(json.dumps(message).encode(), repository_private_key)
    cipherMessage = {
        "ciphertext": ciphertext.hex(),
        "nonce": nonce.hex(),
        "tag": tag.hex(),
        "salt": salt.hex(),
        "signature": signature.hex(),
        "digest": digest.hex()
    }
    sessions_collection.update_one(
        {"session_id": session_id},
        {"$inc": {"number": 1}}
    )
    return cipherMessage
```

Listing 9: Example of encrypting a message with a session; increments the session counter; prepares the encrypted message for secure transmission.

```
# repository.py line 820
@app.route("/role/add", methods=["POST"])
def add_role():
        ...
        organizations_collection.update_one(
            {"name": organization_name},
            {"$set": {"roles." + role: "up"}}
        )

        message = {"message": f"Role {role} add successfully"}
        cipherMessage = encryptMessageWithSession(message, subject_public_key,session["
    session_id"])
        return json.dumps(cipherMessage), 200

    except Exception as e:
        return json.dumps({"error": "An exception occurred", "details": str(e)}), 500
```

Listing 10: Example of encrypting a success message before sending it as a response; uses the session and recipient's public key to ensure secure transmission.

Finally, on the client side, the repository's response is decrypted using the subject's private key and the repository's public key. The message signature is verified. If valid, the client's session number is updated in the `session_file`, and the repository's message content is displayed to the client.

```
# client.py line 209
def decryptDataWithSession(data, session, session_file):
    logger.debug(f"Decrypting data with session: {data}")
    state = load_state()
    number=session["number"]
    private_key = bytes.fromhex(session["private_key"])
    ciphertext = bytes.fromhex(data["ciphertext"])
    nonce = bytes.fromhex(data["nonce"])
    tag = bytes.fromhex(data["tag"])
    salt = bytes.fromhex(data["salt"])
    signature = bytes.fromhex(data["signature"])
    digest = bytes.fromhex(data["digest"])

    state = load_state()
    server_public_key = serialization.load_pem_public_key(state["REP_PUB_KEY"].encode())
    subject_private_key = load_private_key_from_bytes(private_key)

    plaintext = decrypt_data(ciphertext, nonce, tag, salt, subject_private_key,
    server_public_key)
    logger.debug("Data decrypted successfully")

    if verify_signature(signature, digest, server_public_key):
        decrypt = json.loads(plaintext.decode())
        session["number"] = number + 1
        with open(session_file, 'w') as file:
            file.write(json.dumps(session, indent=4))
        logger.debug("Signature verification successful")
        if number + 1 == decrypt["number"]:
            decrypt.pop("number")
        else:
            logger.error("Error: 5.")
            return None, 5
    return decrypt
```

Listing 11: Function to decrypt and verify a message using session data; updates the session counter and ensures the integrity of the decrypted content.

```python
# client.py line 1109
def rep_add_role(session_file, role):
    logger.debug("rep_add_role: Adding role")
    try:
        ...
        if response.status_code == 200:
            logger.info(f"Role '{role}' added successfully")
            data = response.json()
            data = decryptDataWithSession(data, session,session_file)
            logger.info(json.dumps(data, indent=4))
            return 0
        else:
            logger.error(json.dumps({
                "error": "Failed to acitvate subject",
                "status_code": response.status_code,
                "details": response.text
            }, indent=4))
            return -1
```

Listing 12: Example of decrypting and reading a received message; ensures the data is securely decrypted using the session before processing its contents.

2.6. Simplified Encryption and Decryption Process

For commands like `org_create`, `session_create`, and `get_file`, the encryption process is similar to the previously described method. The client generates a new elliptic key pair for every request and derives a `shared_key` with the repository's public key. This is transformed into a `derived_key` for encrypting the message using AES in `GCM` mode. Before encryption, a timestamp is added to the message with the message creation time. The encrypted message is signed to ensure its authenticity, and the necessary components (e.g., `ciphertext`, `nonce`, `tag`, `salt`, `digest`, and `signature` **client's public key**) are sent to the repository in hexadecimal format.

On the repository side, the decryption process mirrors the earlier explanation using the repository's private key and client's public key. It verifies the message's authenticity using the signature and ensures it was sent within the last minute to prevent replay attacks. The timestamp is removed before processing the plaintext.

Response messages are encrypted and signed similarly, using the repository's private key and the client's public key to derive the encryption key. The client then decrypts the response, verifies its signature, and processes the returned data.

This streamlined approach ensures secure and authenticated communication between clients and the repository, leveraging shared key derivation and signature verification as described earlier.

```python
# client.py line 242
def rep_create_org(organization, username, name, email, public_key_file):
    ...
    message = {
        "organization": organization,
        "username": username,
        "name": name,
        "email": email,
        "public_key_file": public_key
    }
    logger.debug(f"Message: {message}")

    cipherMessage,private_key = encryptMessage(message)
    logger.debug("Message encrypted")

    try:
        state = load_state()
        repo_address = state.get("REP_ADDRESS", "localhost:5000")
        logger.debug(f"Repository address: {repo_address}")

        response = requests.post(f"http://{repo_address}/organization/create",json=
    cipherMessage)
        if response.status_code == 200:
            data = response.json()
            data = decryptData(data,private_key)
            logger.info("Organization created successfully")
            logger.debug(json.dumps(data, indent=4))
            return 0
```

Listing 13: Example of using encryption for sending a message and decryption for processing the received response.

```
# client.py line 134
def encryptMessage(message):
    logger.debug(f"Encrypting message: {message}")
    state = load_state()
    message["created1"] = datetime.datetime.now().isoformat()
    server_public_key = serialization.load_pem_public_key(state["REP_PUB_KEY"].encode())
    client_private_key,client_public_key = key_gen_without_file()
    ciphertext, nonce, tag, salt = encrypt_data(json.dumps(message).encode(),
    client_private_key, server_public_key)
    signature, digest = sign_data(json.dumps(message).encode(), client_private_key)
    cipherMessage = {
        "ciphertext": ciphertext.hex(),
        "nonce": nonce.hex(),
        "tag": tag.hex(),
        "salt": salt.hex(),
        "signature": signature.hex(),
        "digest": digest.hex(),
        "public_key":client_public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo).hex()
    }
    logger.debug(f"Message encrypted successfully: {cipherMessage}")
    return cipherMessage,client_private_key


def decryptData(data,client_private_key):
    logger.debug(f"Decrypting data: {data}")
    ciphertext = bytes.fromhex(data["ciphertext"])
    nonce = bytes.fromhex(data["nonce"])
    tag = bytes.fromhex(data["tag"])
    salt = bytes.fromhex(data["salt"])
    signature = bytes.fromhex(data["signature"])
    digest = bytes.fromhex(data["digest"])

    state = load_state()
    server_public_key = serialization.load_pem_public_key(state["REP_PUB_KEY"].encode())

    try:
        plaintext = decrypt_data(ciphertext, nonce, tag, salt, client_private_key,
    server_public_key)
        logger.debug("Data decrypted successfully")

        if verify_signature(signature, digest, server_public_key):
            decrypt = json.loads(plaintext.decode())
            logger.debug("Signature verification successful")
            return decrypt
        else:
            logger.error("Signature verification failed")
            return None
    except Exception as e:
        logger.error(f"Error decrypting data: {e}")
        return None
```

Listing 14: Example of encrypting a message with a client-generated key pair and decrypting the response; ensures secure communication with signature verification.

```
# repository.py line 153

@app.route("/organization/create", methods=["POST"])
def org_create():
    try:
        data = request.get_json()
        if not data:
            return json.dumps({"error": "No JSON data received"}), 400

        data, public_key = decryptDataToData(data)
        if data is None:
            if public_key == 3:
                return json.dumps({"error": "State id_client error"}), 402
```

```python
        ...
        message={"message": "Organization created successfully"}
        cipherMessage=encryptMessage(message, public_key)
        return json.dumps(cipherMessage), 200
```

Listing 15: Example of decrypting a received message

```python
# repository.py line 69
def decryptDataToData(data):
    ciphertext = bytes.fromhex(data["ciphertext"])
    nonce = bytes.fromhex(data["nonce"])
    tag = bytes.fromhex(data["tag"])
    salt = bytes.fromhex(data["salt"])
    signature = bytes.fromhex(data["signature"])
    digest = bytes.fromhex(data["digest"])
    public_key = bytes.fromhex(data["public_key"])
    client_public_key=serialization.load_pem_public_key(public_key)
    try:
        plaintext = decrypt_data(ciphertext, nonce, tag, salt, repository_private_key,
    client_public_key)
        if verify_signature(signature, digest, client_public_key):
            data = json.loads(plaintext.decode())
    except InvalidSignature:
        return None, 1
    except Exception as e:
        return None, 2
    created1 = datetime.datetime.fromisoformat(data["created1"])
    if created1 + datetime.timedelta(minutes=1) > datetime.datetime.now():
        data.pop("created1")
    else:
        return None, 5
    return data, client_public_key

def encryptMessage(message, client_public_key):
    ciphertext, nonce, tag, salt = encrypt_data(json.dumps(message).encode(),
    repository_private_key, client_public_key)
    signature, digest = sign_data(json.dumps(message).encode(), repository_private_key)
    cipherMessage = {
        "ciphertext": ciphertext.hex(),
        "nonce": nonce.hex(),
        "tag": tag.hex(),
        "salt": salt.hex(),
        "signature": signature.hex(),
        "digest": digest.hex()
    }
    return cipherMessage
```

Listing 16: Example of decrypting a received message to validate its integrity and freshness; encrypting a response message for secure communication with the client.

2.7. Authentication and Session Creation

Generating an elliptic key pair requires a file and a password. The password acts as an additional security measure to protect the private key. The private key is stored in a file using the following process:

```python
# KeysGenerator.py line 22
f.write(private_key.private_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PrivateFormat.PKCS8,
    encryption_algorithm=serialization.BestAvailableEncryption(password)
))
```

Listing 17: Saving a private key to a file; the key is serialized using PKCS8 format and encrypted with a password for security.

In this process, the password is integrated into the encryption algorithm, ensuring that only users with the correct password can access the private key. When a subject is created in an organization, it is assigned a public key.

To start a session, the following authentication steps must be completed, and the following arguments provided:

- Organization name;
- Subject name (the user wishing to start the session);
- File containing the elliptic key pair;
- Password used to generate the key pair;
- File to store the session information on the client-side.

Steps:

1. **Load the Private Key:** The `load_private_key` function is used to validate the password. Successful operation confirms that the password matches the one used during key pair generation.

```python
    # client.py line 297
        try:
            logger.debug(f"Loading private key from credentials file '{
    credentials_file}'.")
            client_private_key = load_private_key(password.encode(), credentials_file)
        except Exception as e:
            logger.error("Error loading private key: ", e)
            return 1

```

Listing 18: Loading a private key from a credentials file if passwor is correct and handles errors during key loading.

2. **Generate a Session Identifier (`session_id`):** A unique identifier is generated using `secrets.token_hex(16)`, ensuring uniqueness and security.

```python
    # client.py line 308
    session_id = secrets.token_hex(16)

```

Listing 19: Generating a unique session ID using a secure random hex token.

3. **Build the Authentication Message:** A message is created with the following encoded data:
   - Organization name (`organization`);
   - Username (`username`);
   - Subject's public key (`encryption_key`);
   - Creation date (`created_at`);
   - Session validity (`valid_until`);
   - Session identifier (`session_id`).

```python
    # client.py line 311
    message = {
        "organization": organization_name,
        "username": username,
        "encryption_key": load_public_key_text(credentials_file),
        "created_at": datetime.datetime.now().isoformat(),
        "valid_until": (datetime.datetime.now() + datetime.timedelta(minutes=60)).
    isoformat(),
        "session_id": session_id
    }

    logger.debug(f"Message: {message}")
    message, private_key = encryptMessage(message)

```

Listing 20: Building an authentication message with encoded session details and encrypting it for secure transmission.

4. **Repository Validations:** The repository decodes the received message and performs the following checks:
   - Verify if the organization exists;

- Check if the user is active and linked to the organization;
- Confirm that the submitted public key matches the stored public key.
  The key verification is performed by comparing the hash of the keys using sha256.

```
# repository.py line 113
if organizations_collection.find_one({"name": organization}) is None:
    return json.dumps({"error": "An exception occurred", "details": f"Organization with name '{organization}' no exists."}), 400

organizationSession = organizations_collection.find_one({"name": organization})

subject = subjects_collection.find_one({"username": username})
if not subject or username not in organizationSession["subjects"].keys():
    return json.dumps({"error": "User not found"}), 400

# if suspended
if organizationSession["subjects"][username]["status"] == "down":
    return json.dumps({"error": "Subject is suspended"}), 400

if not(sha256(organizationSession["subjects"][username]["public_key"].encode()).hexdigest() == sha256(encryption_key.encode()).hexdigest()):
    return json.dumps({"error": "Public key was different to in organization with this subject"}), 400

```

Listing 21: Validating organization and user details; ensuring the user is active and their public key matches the organization's record.

5. **Session Creation:** If all validations succeed, a session is created and stored in MongoDB with the following data:
   - `session_id`;
   - Organization;
   - Username;
   - Creation and expiration dates;
   - Public key;
   - Initial number (0);
   - Empty roles list.

```
# repository.py line 229
session_data = {
    "_id": session_id,
    "session_id": session_id,
    "organization": organization,
    "username": username,
    "created_at": created_at,
    "valid_until": valid_until,
    "public_key": encryption_key,
    "number": 0,
    "roles": [],
}
...
sessions_collection.insert_one(session_data)

```

Listing 22: Creating a session entry with all necessary details and storing it in the database.

6. **Client Confirmation:** The repository sends a confirmation message to the client. The client then stores the `session_id`, private key, and initial number in the `session_file`. The private key is used to derive encryption and decryption keys (AES), ensuring secure and unique communications between the client and the repository.

```
# client.py line 332
if response.status_code == 200:
    logger.info("Session created successfully")
    data = response.json()
    data = decryptData(data, private_key)
```

```
        logger.debug(json.dumps(data, indent=4))

        logger.debug(f"Saving session ID to file '{session_file}'")

        session = {
            "session_id": session_id,
            "private_key": client_private_key.private_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PrivateFormat.PKCS8,
                encryption_algorithm=serialization.NoEncryption()
            ).hex(),
            "number": 0
        }

        with open(session_file, "w") as file:
            file.write(json.dumps(session, indent=4))

        logger.debug("Session created successfully and saved to state file")
        return 0

```

Listing 23: Handling a successful session creation response; decrypting the response data and saving session details to a file.

3. Commands

3.1. Features

3.1.1. Organization Management

Allows the creation and listing of organizations.

- rep_create_org
- rep_list_orgs

3.1.2. Session Management

Manages user sessions to provide authenticated access to the system.

- rep_create_session
- All other endpoints depend on session validation (session_id) to ensure security

3.1.3. Role Management

Define and manage roles within an organization. See 1.8

3.1.4. Subject Management

Allows managing users (subjects) and their roles within an organization.

- rep_add_subject
- rep_suspend_subject
- rep_activate_subject
- rep_list_subjects
- rep_add_permission
- rep_remove_permission

3.1.5. Document Management

Manage documents and their associated metadata and permissions.

- rep_add_doc
- rep_get_doc_metadata
- rep_get_doc_file
- rep_acl_doc
- rep_delete_doc
- rep_list_docs

3.2. Local commands

- **rep_subject_credentials**
  - This command is used to generate cryptographic credentials (private and public keys) for a subject, ensuring secure communication with the repository. A new key pair is generated using the **key_gen** function. This command ensures that every subject has an **unique** and **securely stored** cryptographic key pair.
- **rep_decrypt_file**
  - This command is used to decrypt a previously encrypted file using its associated encryption metadata. The file is decrypted using the **AES-GCM** algorithm. The encryption key, IV and authentication tag are extracted from the metadata.

3.3. Anonymous API

- **rep_create_org**
  - This command is used to create a new organization in the repository, defining its initial structure, users, and permissions. The **load_public_key_text** loads the creator's public key from the specified file. A message is constructed containing the organization details and the creator's information, including the loaded public key, sent to the repository. The repository has the responsibility to add a new organization by adding it to the respective collection.
- **rep_list_orgs**
  - This command retrieves and displays a list of all organizations stored in the repository, allowing users to view available organizations and their names. When this function is called, the client sends a request to the repository. On the other hand, the repository lists all organizations, using the database collection.
- **rep_create_session**
  - This command is used to establish a secure session between a user and the repository. Sessions are critical for authenticated interactions, ensuring that all subsequent operations are secure and tied to the authenticated user. When creating a session, a **unique session ID** is generated. If the session creation is successful, the session ID is saved in the **session_file**, **password** is encrypted and the **credentials_file** name are stored in the state for subsequent use.
  - The credentials file used in this command includes the public-private key pair, and **password** is the password that originated the key pair. Furthermore, the public key provided must match the public key associated with the subject in the organization where the session is being logged.
- **rep_get_file**
  - This command retrieves an encrypted file from the repository using its unique handle and optionally saves it to a specified location. The request with the **file_handle** is sent to the repository, which returns the corresponding file.

3.4. Authenticated API

- **rep_assume_role**
  - This command allows a user to assume a specific role within a session. Roles determine the permissions a user has to perform actions in the repository, such as accessing documents, managing users, or modifying roles. The request is made by the client to assume the role, and then the repository checks if the specified role exists and is assigned to the user.
- **rep_drop_role**
  - This command allows a user to remove a previously assumed role from their active session. This ensures users can manage their active roles dynamically within the same session. After the request is made the repository needs to check if the role is active in the session and if so, it is removed from the active session roles.
- **rep_list_roles**
  - This command retrieves a list of all active roles currently assigned to the user's session. This helps users manage and verify their active roles.
- **rep_list_subjects**
  - This command retrieves a list of subjects (users) associated with an organization. This enables users to view information about other members of their organization, such as usernames, names, email addresses, and their statuses. It is possible to pass an **username** as an argument which if passed, makes this command return only one user in the organization.
- **rep_list_role_subjects**
  - This command retrieves all subjects who are assigned a specific role. The repository receives the role and for the given organization, searches and returns the information of the subjects who have that role assigned.
- **rep_list_subject_roles**
  - This command retrieves all roles from a specified subject in an organization.
- **rep_list_role_permissions**
  - This command retrieves all permissions assigned to a specific role within the organization. The repository, receiving the **session_file** and the specified **role** returns all permissions associated with the specified role, including document-specific ACLs.
- **rep_list_permission_roles**
  - This command retrieves a list of roles that are granted a specific permission. As in the last command, the repository receives the **session_file** and the specified **permission** and returns the list of roles associated with the specified permission in the organization's ACL.
- **rep_list_docs**
  - This command retrieves a list of documents stored in the repository, optionally filtered by criteria such as creator or creation date (e.g., newer than, older than, or equal to a specific date). The repository fetches the documents through the **documents collection** of the database checking the conditions if necessary.

3.5. Authorized API

- **rep_add_subject**
  - This command allows an user with the appropriate permissions to add a new subject (user) to an organization. The repository has the responsibility to check validity (see if the user does not already exist), and if it is valid it adds a new subject to the **subject collection**.

- **rep_suspend_subject**
  - This command allows an user with sufficient permissions to suspend another user (subject) within the organization. Suspending a subject prevents them from interacting with the repository until reactivated. The repository checks if the specified subject exists and is part of the organization. If valid, the subject's status is updated to **"down"** in the organization's records. Despite this, if the subject's status is already "down" an **error is thrown**.
- **rep_activate_subject**
  - This command allows an user with sufficient permissions to reactivate a previously suspended subject (user) within the organization. Reactivation restores the subject's ability to interact with the repository. The repository verifies that the specified subject exists and is part of the organization. If valid, the subject's status is updated to **"up"** in the organization's records. As said before, if a subject is already active, he can't be activated again.
- **rep_add_role**
  - This command allows an user with sufficient permissions to add a new role to the organization. The repository checks if the role already exists in the organization's records. If it does not exist, the new role is added to the organization's roles list with the default status **"up"** (active).
- **rep_suspend_role**
  - This command allows an user with sufficient permissions to suspend an existing role in the organization. Suspending a role disables it, preventing any user from assuming it or performing actions associated with it. The repository verifies whether the role exists in the organization's records and whether it is active. If valid, the status of the role is updated to **"down"** in the organization's roles list, but only if the specified role is not **already suspended**.
- **rep_reactivate_role**
  - This command allows an user with sufficient permissions to reactivate a previously suspended role within the organization. Reactivating a role restores its functionality, allowing users to assume it and perform associated actions. The repository checks if the specified role exists and is currently suspended. If valid, the status of the role is updated to **"up"** in the organization's roles list, but only if the specified role is not **already active**.
- **rep_add_permission**
  - This command allows an user with sufficient permissions to not only manage role permissions but also to add a role from a specific user when a username is passed as the argument. In the case a permission is passed as argument, the repository ensures the role exists in the organization's records. If valid, the permission is added to the role's list of permissions. On the other hand, if an user is passed as an argument, the specified role is added to the user's roles list in the subjects field of the organization's data.
- **rep_remove_permission**
  - This command allows an user with sufficient permissions to not only manage role permissions but also to remove a role from a specific user when a username is passed as the argument . In the case a permission is passed as argument, the repository ensures the role and permission exist in the organization's ACL. If valid, the permission is removed from the role. On the other hand, if an user is passed as an argument, the specified role is removed from the user's roles list in the subjects field of the organization's data. For the **MANAGER** role, an additional check is performed to ensure that **at least one** active manager remains in the organization. If the user is the last active manager, the operation is denied to maintain organizational integrity.
- **rep_add_doc**
  - This command is used to securely upload a new document to the repository. It handles encryption of the document, storage of metadata, and configuration of access controls for the uploaded file. The document is encrypted using the **AES-GCM** algorithm, producing **ciphertext**, an **initialization vector (iv)**, and an **authentication tag (tag)**. An unique file handle is generated using the **SHA-256 hash** of the document name and its content. Afterwards, the metadata is created and the **public** and **private metadata** are stored in the respective collections.

- **rep_get_doc_metadata**
  - This command retrieves comprehensive metadata for a specific document. This metadata includes public information (e.g., creation date, creator) and private details (e.g., encryption parameters) if the session has the necessary permissions. The repository consolidates metadata into a unified object, joining both **public** and **private** metadata and encrypts the response. The private metadata is stored to enable subsequent decryption of the file using the **rep_decrypt_file**.
- **rep_get_doc_file**
  - This command retrieves a document file securely from the repository. It combines metadata fetching, file content decryption, and optional file storage to provide a user-friendly way to access documents. The client sends a request to the **/document/get_metadata** endpoint to retrieve document metadata, checking beforehand if the document has **not been deleted**. The metadata includes **file encryption** details (key, IV, tag) and other document-specific information. The file handle is retrieved from the metadata which is then sent to the **/document/get_file** endpoint to retrieve the encrypted file content. Finally the file content is decrypted, and sent to a file (if a file is specified) or printed in the terminal.
- **rep_delete_doc**
  - This command securely removes a document from the repository. It ensures that only authorized users with the appropriate permissions can perform the deletion, while maintaining a record of the deletion action. The server verifies if the document has already been deleted and if not the **deleter** field is set to the username of the session, and the **file_handle** is cleared.
- **rep_acl_doc**
  - This command allows authorized users to manage access control lists (ACLs) for a specific document in the repository. It lets users grant or revoke permissions for a role on a document. The repository verifies the document exists and the permission is valid and based on the signal provided **adds/removes** the role to the permission. Afterwards the **document collection** is updated.

4. Security Analysis Based on OWASP ASVS

4.1. Introduction

To conclude our work, we will analyze it based on Chapter V3 of the Application Security Verification Standard (ASVS). Developed by OWASP, the ASVS is an open-source standard that provides a comprehensive framework for verifying the security of applications. Structured into three levels, it ensures that security controls are consistently applied throughout the development lifecycle.

Session management, addressed in Chapter V3, plays a vital role in maintaining the security and integrity of our document repository. It ensures proper user identification and authorization while safeguarding sensitive information. Effective session management is critical to controlling access, preventing misuse, and supporting confidentiality within the repository.

To comply with best practices, sessions must meet the following requirements:

- Be unique to each user and unpredictable.
- Not be shareable between users or devices.
- Be invalidated when no longer needed or after periods of inactivity.

4.2. Analysis of Session Management Compliance with OWASP ASVS V3

To conduct this analysis, we evaluated the session management mechanisms implemented in the system against the security requirements outlined in OWASP ASVS V3. Particular focus was given to the methods used for session identification, validation, expiration, and overall compliance with best practices. This approach ensures a thorough assessment of the system's adherence to recommended security standards.

- **Session File: A Brief Recap**

- The session file serves as a secure token that uniquely identifies each session through a **session_id**.
- It is stored on the client side, enabling a secure and persistent method for associating users with their active sessions.
- One of its most important features is the **valid_until** mechanism, which enforces a session expiration time. This ensures that sessions are automatically invalidated after their lifetime, reducing the risk of unauthorized access and session hijacking.

V3.1 Fundamental Session Management Security

- **3.1.1 - Verify the application never reveals session tokens in URL parameters.**
  - **Description:** This requirement ensures that session tokens are never exposed as URL parameters to prevent sensitive information from being logged in server logs, browser histories, or intermediaries. Instead, session tokens should be sent in headers, cookies, or message bodies.
  - **Applicability to the Project:** Yes, this is applicable because the project uses a `session_id` to manage user sessions. It's critical to ensure that this identifier is not exposed in the URL for security reasons.
  - **Implementation in the Project:** The project complies with this requirement. The `session_id` is always included in the **message body** of the request rather than as a URL parameter. The `request.get_json()` method is used to retrieve the session information from the incoming JSON payload. Additionally, all messages, including the `session_id`, are encrypted before transmission. This ensures that even if intercepted, the session identifier is not exposed in plain text.
  - **Relevant Code Examples:**
    * **Client-Side Implementation:** The following snippet demonstrates how the encrypted message containing the `session_id` is sent to the repository. This ensures the `session_id` is protected and not exposed in the URL.

```python
# client.py line 515
request_data = {
    "session": session_id
}

if username:
    logger.debug(f"Adding username to request data: {username}")
    request_data["username"] = username

request_data = encryptMessageWithSession(request_data, session)
logger.debug("Sending list subjects request to repository")
response = requests.get(f"http://{repo_address}/subject/list", json=
    request_data)

```

Listing 24: Sending the encrypted message to the repository

    * **Message Encryption Algorithm:** The following snippet illustrates how the message, including the `session_id`, is encrypted before being sent. This guarantees that all sensitive data remains secure during transmission.

```python
# client.py line 185
def encryptMessageWithSession(message, session):
    logger.debug(f"Encrypting message with session: {message}")
    state = load_state()
    session_id=session["session_id"]
    private_key = bytes.fromhex(session["private_key"])
    number=session["number"]
    message["number"] = number
    server_public_key = serialization.load_pem_public_key(state["REP_PUB_KEY"
    ].encode())
    subject_private_key = load_private_key_from_bytes(private_key)
    ciphertext, nonce, tag, salt = encrypt_data(json.dumps(message).encode(),
    subject_private_key, server_public_key)
    signature, digest = sign_data(json.dumps(message).encode(),
    subject_private_key)
    cipherMessage = {
        "ciphertext": ciphertext.hex(),
        "nonce": nonce.hex(),
        "tag": tag.hex(),
        "salt": salt.hex(),
        "signature": signature.hex(),
        "digest": digest.hex(),
        "session_id": session_id,
    }
    logger.debug(f"Message encrypted successfully: {cipherMessage}")
    return cipherMessage

```

Listing 25: Encrypting the message with the session

- **Conclusion:** The project effectively prevents session tokens from being exposed in URL parameters by:
  - Including the `session_id` in the request body.
  - Encrypting all messages before transmission, ensuring additional protection for sensitive session information.

V3.2 Session Binding

- **3.2.1 - Verify the application generates a new session token on user authentication.**
  - **Description:** This requirement ensures that each user session is uniquely identified with a session token that is newly generated upon successful authentication. This helps prevent session fixation attacks and enhances overall security.
  - **Applicability to the Project:** Yes, this is applicable as the project involves managing user sessions securely.
  - **Implementation in the Project:** The project ensures compliance with this requirement. A new `session_id` is generated for each user session using `secrets.token_hex`, which utilizes cryptographically secure random number generation. This ensures uniqueness and unpredictability of session tokens, protecting against brute force or guessing attacks.
  - **Relevant Code Example:** The following snippet illustrates how the `session_id` is generated and securely stored in a *session file*.

```
# client.py line 294
def rep_create_session(organization_name, username, password, credentials_file,
    session_file):
    ...

    # Generate a new session ID using a cryptographically secure random function
    session_id = secrets.token_hex(16)
    logger.debug(f"Session ID: {session_id}")

    # Construct the message containing session metadata
    message = {
        "organization": organization_name,
        "username": username,
        "encryption_key": load_public_key_text(credentials_file),
        "created_at": datetime.datetime.now().isoformat(),
        "valid_until": (datetime.datetime.now() + datetime.timedelta(minutes=60)).
    isoformat(),
        "session_id": session_id
    }

    logger.debug(f"Message: {message}")
    message, private_key = encryptMessage(message)
    ...

    try:
        ...
        if response.status_code == 200:
            logger.info("Session created successfully")
            data = response.json()
            data = decryptData(data, private_key)
            logger.debug(json.dumps(data, indent=4))

            # Save the session data to a dedicated session file
            logger.debug(f"Saving session ID to file '{session_file}'")
```

```python

            session = {
                "session_id": session_id,  # Unique session identifier
                "private_key": client_private_key.private_bytes(
                    encoding=serialization.Encoding.PEM,
                    format=serialization.PrivateFormat.PKCS8,
                    encryption_algorithm=serialization.NoEncryption()
                ).hex(),
                "number": 0  # Counter for session-related operations
            }

            # Write the session data to the specified session file
            with open(session_file, "w") as file:
                file.write(json.dumps(session, indent=4))

            logger.debug("Session created successfully and saved to state file")
            return 0
        ...
```

Listing 26: How the session ID is created and stored in the session file.

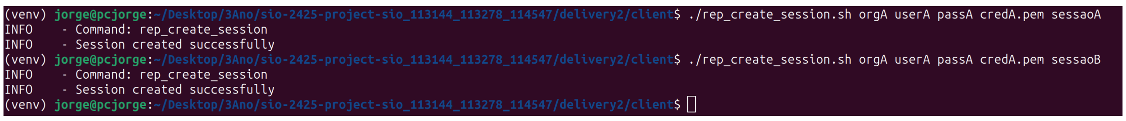
Detailed Analysis:
1. **Session ID Generation:**
- The session identifier (`session_id`) is generated using the Python library `secrets.token_hex(16)`. This method ensures cryptographic security, making the session ID both unpredictable and unique.

2. **Session File Storage:**
- After successfully creating the session on the server, the client application saves the session data to a local file specified by the `session_file` parameter.
- The saved session data includes:
- `session_id`: The unique identifier for the session.
- `private_key`: The client's private key, stored in a secure PEM format and serialized as a hexadecimal string.
- `number`: A counter initialized to zero, used for session-related operations.

3. **Implementation Details:**
- The session file is saved in JSON format, making it easy to read and write session details programmatically.
- The process of saving the session includes proper error handling to ensure that issues (e.g., file permissions) are logged and can be debugged effectively.
- By separating the session information into a dedicated file, the project ensures clear encapsulation of session data, making it easy to manage and secure.

This implementation aligns with the requirements for secure session management by:
- Generating a strong and unique session identifier.
- Securely storing session metadata in an easily accessible and structured format.

– **Visual Examples:** Below are visual examples of the session creation process and session ID generation.

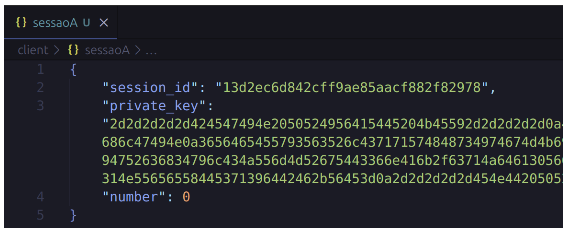


```
(venv) jorge@pcjorge:~/Desktop/3Ano/sio-2425-project-sio_113144_113278_114547/delivery2/client$ ./rep_create_session.sh orgA userA passA credA.pem sessaoA
INFO    - Command: rep_create_session
INFO    - Session created successfully
(venv) jorge@pcjorge:~/Desktop/3Ano/sio-2425-project-sio_113144_113278_114547/delivery2/client$ ./rep_create_session.sh orgA userA passA credA.pem sessaoB
INFO    - Command: rep_create_session
INFO    - Session created successfully
(venv) jorge@pcjorge:~/Desktop/3Ano/sio-2425-project-sio_113144_113278_114547/delivery2/client$ 
```

Figure 1. Example of session creation.

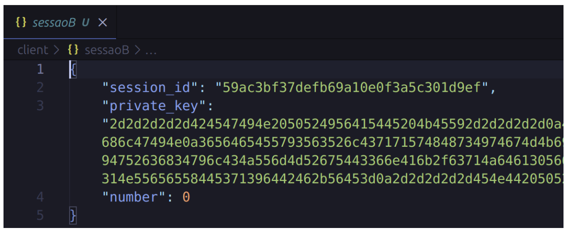
– **Conclusion:** The project effectively generates a new session token upon user authentication, ensuring that:

* Each session is uniquely identified.
* The session token is cryptographically secure and unpredictable.
* The session token is securely stored in a dedicated session file for later use.



```
{
    "session_id": "13d2ec6d842cff9ae85aacf882f82978",
    "private_key":
    "2d2d2d2d2d424547494e20505249564154452d4b45592d2d2d2d2d0a
    686c47494e0a365646545579356352 6c43717157484873497467 4d4b6
    94752636834796c434a556d4d52675443366e416b2f63714a6461305 6
    314e5565655844537139644246 2b56453d0a2d2d2d2d2d454e4420505
    "number": 0
}
```

(a) Example of session ID (1).



```
{
    "session_id": "59ac3bf37defb69a10e0f3a5c301d9ef",
    "private_key":
    "2d2d2d2d2d424547494e20505249564154452d4b45592d2d2d2d2d0a
    686c47494e0a365646545579356352 6c43717157484873497467 4d4b6
    94752636834796c434a556d4d52675443366e416b2f63714a6461305 6
    314e5565655844537139644246 2b56453d0a2d2d2d2d2d454e4420505
    "number": 0
}
```

(b) Example of session ID (2).

Figure 2. Examples of generated session IDs.

- **3.2.2 - Verify that session tokens possess at least 64 bits of entropy.**
  - **Description:** This verification point ensures that session tokens have sufficient randomness (entropy) to make them unique and unpredictable. Tokens with at least 64 bits of entropy are highly resistant to brute force attacks and token prediction, providing a secure foundation for session management.
  - **Applicability to the Project:** This verification point is highly relevant to the project, as it ensures that session tokens are cryptographically secure and resistant to brute force attacks, which is critical for maintaining secure user sessions in any system handling sensitive data.
  - **Implementation in the Project:** Session tokens are generated using `secrets.token_hex(16)`, which provides a token with 16 bytes (128 bits) of entropy. This ensures that each session ID is unique, unpredictable, and resistant to guessing or brute force attacks.
  - **Analysis of the Implementation:**
    - **Security:** The use of `secrets.token_hex` leverages system-level randomness, ensuring compliance with modern cryptographic security standards.
    - **Sufficient Entropy:** Each session token contains 128 bits of entropy, which is twice the required minimum of 64 bits, ensuring robustness against security threats.
    - **Resilience:** The implementation effectively mitigates the risk of session hijacking or prediction, safeguarding user sessions.
  - **Relevant Code Example:** The following snippet demonstrates the creation of a secure `session_id` using `secrets.token_hex(16)`:

```python
        # client.py line 308
        session_id = secrets.token_hex(16)
        logger.debug(f"Generated secure session ID: {session_id}")

```

Listing 27: New session_id generation.

  - **Conclusion:** The project successfully implements session tokens with sufficient entropy, exceeding the requirements outlined in this verification point. By utilizing `secrets.token_hex`, the project ensures secure, unpredictable, and unique session identifiers, aligning with best practices for session management security.

- **3.2.3 - Verify the application only stores session tokens in the browser using secure methods such as appropriately secured cookies (see section 3.4) or HTML5 session storage.**
  - **Description:** This verification point ensures that session tokens, if stored in the browser, are handled securely using mechanisms like HTTP-only, Secure, and SameSite cookies, or through controlled use of HTML5 session storage. Proper storage protects tokens from being accessed by malicious scripts or transmitted insecurely, mitigating risks such as Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF).
  - **Applicability to the Project:** This point is not applicable to the project, as the application does not store session tokens in the browser or interact directly with browser environments. Instead, the session management strategy is designed around secure local storage on the client system.
  - **Implementation in the Project:** The `session_id` is managed entirely outside the browser context and is stored locally in a secure session file on the client system. This approach minimizes the risk of exposure to browser-related security vulnerabilities such as XSS or CSRF.

- **Analysis of the Implementation:**
    * **Storage Location:** The session token is written to a session file stored on the local file system of the client, ensuring that it is isolated from browser-based threats.
    * **Security Measures:** Local storage of the `session_id` leverages controlled file access, limiting exposure to unauthorized entities. This approach avoids the complexities of browser security configurations.
    * **Compliance with Requirements:** While the application does not use browser storage, the chosen storage mechanism adheres to security best practices for protecting sensitive data.
- **Conclusion:** This verification point is not applicable to the project, as no browser-based storage mechanisms are utilized.

- **3.2.4 - Verify that session tokens are generated using approved cryptographic algorithms.**
    - **Description:** This verification point ensures that session tokens are generated using cryptographically secure and approved algorithms, making them resistant to prediction or brute force attacks. Using weak or non-approved algorithms could expose the application to security vulnerabilities, compromising session integrity.
    - **Applicability to the Project:** This point is applicable to the project, as session tokens (`session_id`) play a critical role in maintaining secure session management. Ensuring that these tokens are generated using secure algorithms is essential to prevent token compromise.
    - **Implementation in the Project:**
        * **Algorithm Used:** The `session_id` is generated using the `secrets.token_hex` function from Python's `secrets` module. This module was introduced via PEP 506 to provide a high-level interface for generating cryptographically secure random numbers.
        * **Security Standards:** The `secrets` module relies on Cryptographically Strong Pseudo-Random Number Generators (CSPRNG) to produce random tokens. It uses secure sources such as `os.urandom`, which are suitable for high-security applications.
        * **Compliance Considerations:** While the `secrets.token_hex` function generates secure and unpredictable tokens, it is essential to verify that the underlying CSPRNG implementation adheres to approved cryptographic standards relevant to the deployment environment.
    - **Analysis of the Implementation:**
        * The choice of `secrets.token_hex` ensures a high level of security, as it is specifically designed for generating random values that are suitable for security-related tasks, such as session tokens.
        * The reliance on `os.urandom` or equivalent secure random number generators ensures compliance with modern security best practices.
        * While the method is secure, additional verification could be performed to ensure that the algorithm aligns with specific cryptographic standards required by the organization or regulatory body.
    - **Conclusion:** The implementation of `secrets.token_hex` in the project effectively ensures the generation of cryptographically secure and unpredictable session tokens. This approach is appropriate for secure session management, adhering to best practices for randomness and security. However, for absolute assurance, the organization should validate that the underlying algorithm complies with any applicable regulatory or cryptographic standards.

V3.3 Session Termination

- **3.3.1 - Verify that logout and expiration invalidate the session token, such that the back button or a downstream relying party does not resume an authenticated session, including across relying parties.**
    - **Description:** This verification ensures that session tokens are invalidated immediately upon logout or expiration. This prevents unauthorized access through mechanisms such as the browser's back button or by relying on tokens across downstream services. It is an essential control to protect against session fixation and replay attacks.

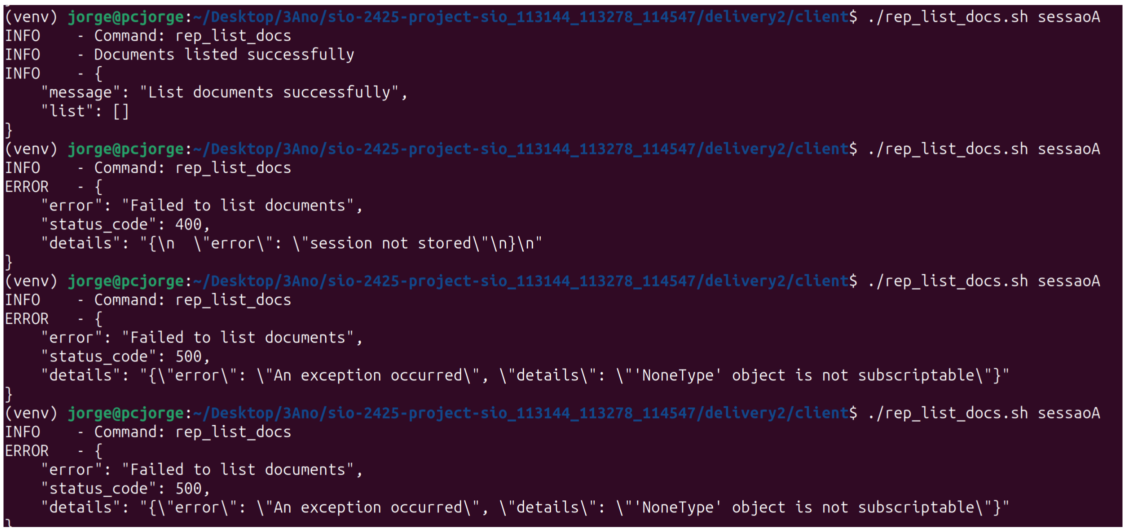
- **Applicability to the Project:** This point partially applies to the project. While the application does not involve browser-based sessions or downstream relying parties, it manages session validation locally. Sessions are validated each time an action requiring authentication is performed, ensuring expired or invalid sessions cannot be reused.
- **Implementation in the Project:** The project enforces session expiration by validating the `session_id` against the `valid_until` timestamp and the user's status. If the session has expired or is invalid for other reasons, it is immediately removed from the database, preventing further use.
- **Relevant Code Example:** The following function ensures that sessions are validated, expired sessions are deleted, and invalid sessions return a `None` response, blocking unauthorized actions.

```python
# repository.py line 261
def getSession(session_id):
    session_data = sessions_collection.find_one({"_id": session_id})
    if session_data:
        valid_until = datetime.datetime.fromisoformat(session_data["valid_until"])
        organization = organizations_collection.find_one({"name": session_data["organization"]})
        if valid_until < datetime.datetime.now():
            sessions_collection.delete_one({"_id": session_id})
            return None
        if organization['subjects'][session_data["username"]]["status"] == "down":
            return None
        return session_data
    else:
        return None

```

Listing 28: Code ensuring session expiration and invalidation.

- **Supporting Evidence:** The following image demonstrates the behavior of the application when attempting to execute commands with expired sessions. Initially, the session is valid, allowing the user to list documents. Once the session expires, subsequent actions return errors indicating that the session is invalid.



```
(venv) jorge@pcjorge:~/Desktop/3Ano/sio-2425-project-sio_113144_113278_114547/delivery2/client$ ./rep_list_docs.sh sessaoA
INFO    - Command: rep_list_docs
INFO    - Documents listed successfully
INFO    - {
    "message": "List documents successfully",
    "list": []
}
(venv) jorge@pcjorge:~/Desktop/3Ano/sio-2425-project-sio_113144_113278_114547/delivery2/client$ ./rep_list_docs.sh sessaoA
INFO    - Command: rep_list_docs
ERROR   - {
    "error": "Failed to list documents",
    "status_code": 400,
    "details": "{\n  \"error\": \"session not stored\"\n}\n"
}
(venv) jorge@pcjorge:~/Desktop/3Ano/sio-2425-project-sio_113144_113278_114547/delivery2/client$ ./rep_list_docs.sh sessaoA
INFO    - Command: rep_list_docs
ERROR   - {
    "error": "Failed to list documents",
    "status_code": 500,
    "details": "{\"error\": \"An exception occurred\", \"details\": \"'NoneType' object is not subscriptable\"}"
}
(venv) jorge@pcjorge:~/Desktop/3Ano/sio-2425-project-sio_113144_113278_114547/delivery2/client$ ./rep_list_docs.sh sessaoA
INFO    - Command: rep_list_docs
ERROR   - {
    "error": "Failed to list documents",
    "status_code": 500,
    "details": "{\"error\": \"An exception occurred\", \"details\": \"'NoneType' object is not subscriptable\"}"
```

Figure 3. Demonstrating session expiration: Initial actions succeed with a valid session, but subsequent attempts fail after the session has expired.

- **Analysis:**
  * **Strengths:** The mechanism ensures that expired sessions are promptly invalidated by deleting them from the database. This prevents unauthorized access to resources or actions that require authentication.
  * **Limitations:** The project does not implement a logout functionality to actively invalidate sessions, as it relies solely on session expiration to handle invalidation. This may limit flexibility in session management.

    * **Observations:** The current implementation effectively protects against the reuse of expired sessions by deleting them immediately upon validation failure. However, it could be further improved by adding explicit session termination logic for scenarios where users may want to log out manually.
  - **Conclusion:** The project adequately handles session expiration by validating the `session_id` and removing it from the database if it has expired. Although there is no explicit logout functionality, the existing implementation provides robust security for invalidating expired sessions, aligning with the requirements of this point.

- **3.3.2 - If authenticators permit users to remain logged in, verify that re-authentication occurs periodically both when actively used or after an idle period.**
  - **Description:** This requirement ensures that sessions are periodically re-authenticated to maintain security, either after a defined period of active use or following an idle period. For higher security levels (L3), sessions should terminate after 15 minutes of inactivity or extend up to 12 hours if actively used.
  - **Applicability to the Project:** This point is applicable to the project, as session expiration is already enforced after a fixed period of 60 minutes. However, the current implementation does not differentiate between active and idle sessions, which limits compliance with advanced security standards.
  - **Implementation in the Project:** The project enforces a fixed 60-minute timeout for session expiration, regardless of user activity. This approach ensures regular re-authentication but does not align with the requirements for idle and active session handling specified for L3 compliance.
  - **Proposed Enhancements:** To fully comply with the requirements: - Introduce a parameter to store the last interaction time (`last_interaction`). - Update this value with every user interaction requiring a valid session. - Terminate sessions if idle for more than 15 minutes or extend them up to a maximum of 12 hours if actively used. - This dynamic mechanism would align the project with advanced session management practices.
  - **Conclusion:** The project enforces a robust fixed expiration mechanism, enhancing security by requiring regular re-authentication. However, it does not fully address advanced security requirements, such as dynamic session handling based on activity and idle periods. Implementing these features would improve security and bring the project into alignment with best practices for session management.

- **3.3.3 - Verify that the application gives the option to terminate all other active sessions after a successful password change (including change via password reset/recovery), and that this is effective across the application, federated login (if present), and any relying parties.**
  - **Description:** This requirement ensures that when a user changes their password (either via a manual update or recovery process), all other active sessions are terminated. This reduces the risk of unauthorized access in the event of compromised session tokens. The termination must be enforced throughout the application, across federated login systems (if used), and any third-party systems relying on the session.
  - **Applicability to the Project:** This requirement is not applicable to the current project, as it does not include functionalities for password changes, recovery, or federated login. All authentication and session management occur independently within the system.
  - **Conclusion:** While the requirement does not apply to the current implementation, future enhancements could consider adding support for password change and recovery mechanisms. If these features are introduced, implementing session invalidation upon password updates would be essential to align with OWASP best practices for session management.

- **3.3.4 - Verify that users are able to view and (having re-entered login credentials) log out of any or all currently active sessions and devices.**
  - **Description:** This requirement ensures that users have visibility over all their active sessions and can manage them by logging out of specific sessions or all sessions across devices. This capability enhances security by allowing users to terminate unauthorized or unnecessary sessions, particularly in cases of suspicious activity.

- **Applicability to the Project:** This requirement is not applicable to the current project, as the system does not provide features for users to view or manage active sessions. The project manages sessions on a per-instance basis, without user-level visibility or control over session activity.
- **Conclusion:** Although this functionality is not currently implemented, it could be a valuable feature to consider for future iterations of the project. Adding session management capabilities would give users greater control and visibility over their sessions, enhancing security and aligning the project with advanced session security practices outlined in the OWASP guidelines. Providing such functionality, combined with authentication for critical actions, would improve the overall user experience and system robustness.

V3.4 Cookie-based Session Management

- **Description:** This requirement ensures that session tokens managed via browser cookies include essential security attributes such as 'Secure', 'HttpOnly', and 'SameSite'. These attributes aim to protect session cookies from interception, unauthorized access, or misuse. For instance, the 'Secure' attribute ensures that cookies are transmitted only over HTTPS, 'HttpOnly' prevents client-side scripts from accessing cookies, and 'SameSite' mitigates Cross-Site Request Forgery (CSRF) attacks by restricting cookie sharing across sites.
- **Applicability to the Project:** This requirement is not applicable to the current project because session management is handled using tokens transmitted directly through REST API requests and stored in the `sessions_collection` in MongoDB. These tokens are explicitly included in the headers or body of API calls and are not stored or managed via browser cookies. Consequently, security measures associated with cookie attributes are irrelevant in the context of this project.
- **Conclusion:** While cookie-based session management is not applicable, the current implementation of token-based session management offers strong security advantages, including centralized session tracking and reduced exposure to client-side vulnerabilities. However, it is critical to ensure proper handling of tokens, including secure storage on the client-side (e.g., `localStorage` or `sessionStorage`), transmission over HTTPS, and prompt invalidation upon logout or suspicious activity. Future iterations of the project could enhance security by incorporating features such as token revocation and short-lived tokens with refresh capabilities to further bolster protection against session-related threats.

V3.5 Token-based Session Management

- **3.5.1 - Verify the application allows users to revoke OAuth tokens that form trust relationships with linked applications.**
  - **Description:** This requirement focuses on ensuring that applications leveraging OAuth allow users to revoke issued tokens that establish trust relationships with third-party or linked applications. This functionality is essential to provide users control over their data and linked services, allowing them to terminate access granted to external applications when no longer needed or in the event of unauthorized activity.
  - **Applicability to the Project:** This requirement is not applicable to the current project, as the system does not utilize OAuth or include integrations with external applications that require trusted tokens. Session management is implemented entirely within the system using internally generated tokens, and there are no external trust relationships that need to be managed.
  - **Conclusion:** While OAuth-related token management is not relevant to this project, it highlights the importance of user control over session and access management. If future iterations of the project involve integrations with external services or adopt OAuth for authentication or authorization purposes, implementing token revocation capabilities would be a critical security measure. For now, the internal session management approach provides sufficient isolation and control within the application's scope.
- **3.5.2 - Verify the application uses session tokens rather than static API secrets and keys, except with legacy implementations.**
  - **Description:** This requirement ensures that applications use dynamically generated session tokens for managing user sessions instead of static API secrets or keys. Dynamic tokens enhance security by being unique to each session and reducing the risk of credential reuse attacks. The use of static secrets or hardcoded keys is discouraged as it can lead to vulnerabilities such as unauthorized access if the keys are exposed.

- **Applicability to the Project:** This requirement is applicable to the project, as it is directly related to secure session management. The system relies on session tokens to authenticate and manage user interactions, making this aspect critical to its design.
- **Implementation in the Project:** The project implements session tokens in the form of `session_id`, which is dynamically generated at the start of each user session. These tokens are unique to each session and are stored in the `sessions_collection` in MongoDB and locally in the `session_file` on the client side. Tokens are used to identify active sessions securely and are included in API requests for authentication purposes.

  The use of `session_id` ensures that:
  * Each session is uniquely identifiable.
  * Tokens are not static or hardcoded in the system, eliminating the risk of exposure through code repositories or other vectors.
  * Session management aligns with security best practices by using ephemeral identifiers instead of long-lived static keys.
- **Security Advantages:** By adopting dynamically generated tokens, the system mitigates common threats such as credential reuse, token theft, and unauthorized access. The tokens are short-lived and bound to specific sessions, enhancing overall security.
- **Conclusion:** The implementation of session tokens (`session_id`) demonstrates the project's adherence to secure session management practices. This approach effectively addresses the requirements of the OWASP guidelines for avoiding static secrets or API keys. Moving forward, maintaining this standard and implementing further enhancements such as token expiration policies and secure transmission methods (e.g., HTTPS) will ensure continued compliance and robustness against evolving security threats.

• **3.5.3 - Verify that stateless session tokens use digital signatures, encryption, and other countermeasures to protect against tampering, enveloping, replay, null cipher, and key substitution attacks.**

- **Description:** This requirement ensures that stateless session tokens are secured using techniques such as digital signatures and encryption to safeguard against various attacks, including tampering, enveloping, replay, null cipher, and key substitution. These protections are critical for maintaining the integrity and confidentiality of tokens in a stateless architecture.
- **Applicability to the Project:** The project employs a stateless session management system, making this requirement directly applicable. The implementation of cryptographic measures for session tokens is essential to ensuring the security of the system.
- **Implementation in the Project:** The project implements stateless session tokens with the following security measures:
  * **Digital Signatures:** The system uses ECDSA (Elliptic Curve Digital Signature Algorithm) to sign session tokens. This ensures the authenticity of tokens and protects against tampering and key substitution attacks.
  * **Encryption:** Messages exchanged with the repository are encrypted using AES-GCM (Advanced Encryption Standard in Galois/Counter Mode). This provides confidentiality and integrity, preventing null cipher and enveloping attacks.
  * **Replay Attack Protection:** For messages that do not require a session (e.g., create organization, list organizations, create session), replay attack protection is achieved by validating the creation time (‘created1‘) of the message. Messages are accepted only if they are created within a one-minute window. For commands requiring a session (e.g., list subjects), a counter is used as an additional layer of protection against replay attacks. The counter ensures that each session message is unique and sequentially processed on both the client and server sides.

```python
# client.py line 134
def encryptMessage(message):
    logger.debug(f"Encrypting message: {message}")
    state = load_state()
    message["created1"] = datetime.datetime.now().isoformat()
    server_public_key = serialization.load_pem_public_key(state["REP_PUB_KEY"].
    encode())
    client_private_key, client_public_key = key_gen_without_file()
    ciphertext, nonce, tag, salt = encrypt_data(json.dumps(message).encode(),
    client_private_key, server_public_key)
    signature, digest = sign_data(json.dumps(message).encode(), client_private_key
    )
    cipherMessage = {
        "ciphertext": ciphertext.hex(),
        "nonce": nonce.hex(),
        "tag": tag.hex(),
        "salt": salt.hex(),
        "signature": signature.hex(),
        "digest": digest.hex(),
        "public_key": client_public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo).hex()
    }
    logger.debug(f"Message encrypted successfully: {cipherMessage}")
    return cipherMessage, client_private_key
```

Listing 29: Encrypting a message from client to repository for non-session-based actions

Analysis: This code implements the `encryptMessage` function, which encrypts a message to be sent from the client to the server without using a session. The message includes a `created1` field that records the message creation timestamp. This timestamp is later verified by the server to ensure the message is recent, thereby mitigating replay attacks.

```
# repository.py line 69
def decryptDataToData(data):
    ciphertext = bytes.fromhex(data["ciphertext"])
    nonce = bytes.fromhex(data["nonce"])
    tag = bytes.fromhex(data["tag"])
    salt = bytes.fromhex(data["salt"])
    signature = bytes.fromhex(data["signature"])
    digest = bytes.fromhex(data["digest"])
    public_key = bytes.fromhex(data["public_key"])
    client_public_key = serialization.load_pem_public_key(public_key)
    try:
        plaintext = decrypt_data(ciphertext, nonce, tag, salt,
    repository_private_key, client_public_key)
        if verify_signature(signature, digest, client_public_key):
            data = json.loads(plaintext.decode())
    except InvalidSignature:
        return None, 1
    except Exception as e:
        return None, 2
    created1 = datetime.datetime.fromisoformat(data["created1"])
    if created1 + datetime.timedelta(minutes=1) > datetime.datetime.now():
        data.pop("created1")
    else:
        return None, 5
    return data, client_public_key
```

Listing 30: Decrypting a message on the repository side and validating the creation time

Analysis: The `decryptDataToData` function decrypts messages on the server side. After decrypting and verifying the message's signature, it checks the `created1` field to ensure the message is less than one minute old. Messages older than this threshold are rejected to prevent replay attacks in non-session-based interactions.

```
# client.py line 185
def encryptMessageWithSession(message, session):
    logger.debug(f"Encrypting message with session: {message}")
    state = load_state()
    session_id = session["session_id"]
    private_key = bytes.fromhex(session["private_key"])
    number = session["number"]
    message["number"] = number
    server_public_key = serialization.load_pem_public_key(state["REP_PUB_KEY"].
    encode())
    subject_private_key = load_private_key_from_bytes(private_key)
    ciphertext, nonce, tag, salt = encrypt_data(json.dumps(message).encode(),
    subject_private_key, server_public_key)
    signature, digest = sign_data(json.dumps(message).encode(),
    subject_private_key)
    cipherMessage = {
        "ciphertext": ciphertext.hex(),
        "nonce": nonce.hex(),
        "tag": tag.hex(),
        "salt": salt.hex(),
        "signature": signature.hex(),
        "digest": digest.hex(),
        "session_id": session_id,
    }
    logger.debug(f"Message encrypted successfully: {cipherMessage}")
    return cipherMessage
```

Listing 31: Encrypting a message from client to repository for session-based actions

Analysis: The `encryptMessageWithSession` function encrypts messages that require a session. The message includes a session counter (`number`) that ensures message integrity and helps protect against replay attacks. The counter is incremented with each message to maintain order and uniqueness.

```
# repository.py line 107
def decryptDataToDataWithSession(data):
    ciphertext = bytes.fromhex(data["ciphertext"])
    nonce = bytes.fromhex(data["nonce"])
    tag = bytes.fromhex(data["tag"])
    salt = bytes.fromhex(data["salt"])
    signature = bytes.fromhex(data["signature"])
    digest = bytes.fromhex(data["digest"])
    session_id = data.get("session_id")
    session = sessions_collection.find_one({"_id": session_id})
    global_public_key_file = serialization.load_pem_public_key(session["public_key
    "].encode())

    try:
        plaintext = decrypt_data(ciphertext, nonce, tag, salt,
    repository_private_key, global_public_key_file)
        if verify_signature(signature, digest, global_public_key_file):
            data = json.loads(plaintext.decode())
    except InvalidSignature:
        return None, 1
    except Exception as e:
        return None, 2
    if session["number"] == data["number"]:
        data.pop("number")
    else:
        return None, 5
    return data, global_public_key_file
```

Listing 32: Decrypting and validating a session-based message on the repository side

Analysis: The `decryptDataToDataWithSession` function decrypts and validates session-based messages on the server side. It compares the session number (`number`) from the message with the expected number stored in the repository. This ensures messages are processed in order and prevents replay attacks.

```
# repository.py line 132
def encryptMessageWithSession(message, subject_public_key, session_id):
    session = sessions_collection.find_one({"session_id": session_id})
    number = session["number"]
    message["number"] = number + 1
    ciphertext, nonce, tag, salt = encrypt_data(json.dumps(message).encode(),
    repository_private_key, subject_public_key)
    signature, digest = sign_data(json.dumps(message).encode(),
    repository_private_key)
    cipherMessage = {
        "ciphertext": ciphertext.hex(),
        "nonce": nonce.hex(),
        "tag": tag.hex(),
        "salt": salt.hex(),
        "signature": signature.hex(),
        "digest": digest.hex()
    }
    sessions_collection.update_one(
        {"session_id": session_id},
        {"$inc": {"number": 1}}
    )
    return cipherMessage
```

Listing 33: Encrypting a response from repository to client with session

Analysis: This block, in the `encryptMessageWithSession` function, handles message encryption when the server sends a response to the client in a session context. The server increments the session counter (`number`) before sending the message, ensuring message sequence integrity. The updated counter is stored in the repository and included in the outgoing message.

```
# client.py line 209
def decryptDataWithSession(data, session, session_file):
    logger.debug(f"Decrypting data with session: {data}")
    state = load_state()
    number = session["number"]
    private_key = bytes.fromhex(session["private_key"])
    ciphertext = bytes.fromhex(data["ciphertext"])
    nonce = bytes.fromhex(data["nonce"])
    tag = bytes.fromhex(data["tag"])
    salt = bytes.fromhex(data["salt"])
    signature = bytes.fromhex(data["signature"])
    digest = bytes.fromhex(data["digest"])

    state = load_state()
    server_public_key = serialization.load_pem_public_key(state["REP_PUB_KEY"].
    encode())
    subject_private_key = load_private_key_from_bytes(private_key)

    plaintext = decrypt_data(ciphertext, nonce, tag, salt, subject_private_key,
    server_public_key)
    logger.debug("Data decrypted successfully")

    if verify_signature(signature, digest, server_public_key):
        decrypt = json.loads(plaintext.decode())
        session["number"] = number + 1
        with open(session_file, 'w') as file:
            file.write(json.dumps(session, indent=4))
        logger.debug("Signature verification successful")
        if number + 1 == decrypt["number"]:
            decrypt.pop("number")
        else:
            logger.error("Error: 5.")
            return None, 5
    return decrypt
```

Listing 34: Decrypting a response from repository to client with session

Analysis: The `decryptDataWithSession` function decrypts and validates messages received by the client in a session context. It verifies the digital signature and compares the session counter (`number`) in the message with the client's locally stored counter. If the counters do not match, the message is rejected, protecting against replay attacks or out-of-sequence messages.

- **Conclusion:** The project effectively implements countermeasures such as encryption and digital signatures to secure stateless session tokens against tampering, enveloping, null cipher, and key substitution attacks. Replay attack protection is ensured through the use of time-based validation for non-session messages and counters for session-based messages. These implementations align with best practices and OWASP guidelines, providing a robust defense against common security threats.

V3.6 Federated Re-authentication

- **Description:** Federated re-authentication refers to the process by which Relying Parties (RPs) and Credential Service Providers (CSPs) collaborate to manage user authentication. In such a system, the RP delegates authentication to a CSP, which issues tokens or credentials to verify the user's identity. This approach allows for centralized authentication, enabling users to access multiple services with a single sign-on (SSO). Key aspects include:

    - **Session Freshness:** Ensuring that RPs specify the maximum allowed time for user authentication, prompting the CSP to re-authenticate users when necessary.
    - **Authentication Event Transparency:** CSPs must inform RPs about the last authentication event, allowing RPs to decide whether to re-authenticate users.
    - These mechanisms enhance security by preventing session hijacking or the misuse of stale credentials.
  - **Applicability to the Project:** This requirement is not directly applicable to the current system. The project does not utilize a federated authentication model involving RPs and CSPs. Instead, authentication and session management are handled centrally within the application, using internally generated and managed tokens.
  - **Implementation in the Project:** The project uses a standalone authentication mechanism:
    - Tokens (`session_id`) are dynamically generated and managed within the application.
    - Session management does not rely on external Credential Service Providers or federated protocols, eliminating the need for RPs and CSPs.
    - Authentication is handled directly by the system, ensuring simplicity and full control over user sessions without delegating responsibilities to external entities.
  - **Limitations and Recommendations:** Although the current system does not implement federated re-authentication, it could benefit from adopting certain practices associated with federated systems, such as:
    - **Session Freshness Checks:** Regularly validating token expiration or user activity to ensure sessions remain secure and valid.
    - **Event Transparency:** Logging authentication events to provide administrators with visibility over user activity, akin to how RPs track authentication via CSPs.

    Implementing such practices could improve security without introducing the complexity of federated authentication.
  - **Conclusion:** Federated re-authentication focuses on collaboration between Relying Parties and Credential Service Providers to maintain session security. While this approach is not relevant to the current project, the principles of session management, such as enforcing session expiration and maintaining authentication event logs, can still be adopted to enhance the system's overall security. These measures would ensure a robust and secure authentication process while preserving the simplicity of the current architecture.

V3.7 Defenses Against Session Management Exploits

- **3.7.1 - Verify the application ensures a full, valid login session or requires reauthentication or secondary verification before allowing any sensitive transactions or account modifications.**
  - **Description:** This requirement ensures that sensitive operations, such as modifying account settings, resetting passwords, or updating permissions, are performed only within the context of a valid login session. It also recommends additional layers of security, such as reauthentication or secondary verification, to safeguard against unauthorized access.
  - **Applicability to the Project:** The project directly addresses session validation as part of its security framework, requiring users to authenticate with secure credentials before initiating any session. This aligns with the requirement to ensure that sensitive actions are restricted to authenticated users. However, additional security mechanisms like reauthentication or secondary verification are not yet implemented.
  - **Implementation in the Project:** In the current implementation:
    - A valid login session is mandatory to interact with the system. The login process involves:
      - Submitting a credentials file containing a public-private key pair.
      - Entering the password that corresponds to the private key.
      - Verifying that the provided public key matches the one registered for the user in the organization.
    - The system enforces strong validation during session initiation, ensuring that only authorized users can access the system.

    * Sensitive operations, such as account modifications or permission changes, currently rely on the validity of the active session without requiring additional verification steps.

- **Limitations and Recommendations:** While the system provides robust session validation, there are opportunities to enhance its defenses against unauthorized access:
    * **Reauthentication for Sensitive Operations:** Introduce a mechanism to require reauthentication before executing sensitive actions. This could involve re-submitting the credentials file or re-entering the password associated with the private key.
    * **Multi-Factor Authentication (MFA):** Implement MFA as an added layer of protection for high-risk operations, such as profile updates or permission modifications.
    * **Session Expiry and Timeout:** Enforce shorter session expiration periods to minimize the risk of session hijacking and require users to reauthenticate more frequently.
- **Conclusion:** The project meets the requirement of ensuring a full and valid login session before granting access. However, incorporating additional safeguards, such as reauthentication for sensitive transactions and multi-factor authentication, would further align the system with advanced security practices. These enhancements would strengthen protection against unauthorized access and enhance user trust in the application.

4.3. Compliance Analysis of Session Management Requirements

Table 1 summarizes the compliance analysis of session management requirements based on OWASP ASVS V3. Each requirement is listed along with a description and whether it is applicable to the current project.

Requirement	**Description**	**Applicable to Project**	**Implemented in Project**
V3.1.1	Verify the application never reveals session tokens in URL parameters.	Yes	Yes
V3.2.1	Verify the application generates a new session token on user authentication.	Yes	Yes
V3.2.2	Verify that session tokens possess at least 64 bits of entropy.	Yes	Yes
V3.2.3	Verify the application only stores session tokens in the browser using secure methods such as appropriately secured cookies (see section 3.4) or HTML 5 session storage.	No	No
V3.2.4	Verify that session tokens are generated using approved cryptographic algorithms.	Yes	Yes
V3.3.1	Verify that logout and expiration invalidate the session token, such that the back button or a downstream relying party does not resume an authenticated session, including across relying parties.	Yes	Partial - Session expiration is handled, but explicit logout functionality is not implemented.
V3.3.2	If authenticators permit users to remain logged in, verify that re-authentication occurs periodically both when actively used or after an idle period.	Yes	No
V3.3.3	Verify that the application gives the option to terminate all other active sessions after a successful password change (including change via password reset/recovery), and that this is effective across the application, federated login (if present), and any relying parties.	No	No
V3.3.4	Verify that users are able to view and (having re-entered login credentials) log out of any or all currently active sessions and devices.	No	No
V3.4	Cookie-based Session Management.	No	No
V3.5.1	Verify the application allows users to revoke OAuth tokens that form trust relationships with linked applications.	No	No
V3.5.2	Verify the application uses session tokens rather than static API secrets and keys, except with legacy implementations.	Yes	Yes
V3.5.3	Verify that stateless session tokens use digital signatures, encryption, and other countermeasures to protect against tampering, enveloping, replay, null cipher, and key substitution attacks.	Yes	Yes
V3.6	Federated Re-authentication.	No	No
V3.7.1	Verify the application ensures a full, valid login session or requires reauthentication or secondary verification before allowing any sensitive transactions or account modifications.	Yes	Partial - Not fully compliant but partially implemented in the project.

Table 1. Compliance Analysis of Session Management Requirements Based on OWASP ASVS V3

4.4. Project Improvements

4.4.1. Delivery1

- **Session system fixed:** The session management system was corrected and is now working as intended.
- **Commands corrected:** All previously non-operational commands, such as `rep_get_file`, `rep_get_doc_file`, and others, were fixed and are now fully functional.

4.4.2. Delivery1 and Delivery2

- **Improved variable naming:** The names of public and private key variables were changed to enhance clarity and understanding. For instance, the subject key previously referred to as a "global key" was renamed more appropriately.
- **Removal of the global public client key:** The global public client key was removed from the system and is no longer sent, resolving the issue of having to remove the state.
- **Encryption and decryption changes without session:** Instead of using the client's global key, a new key pair is now generated for each operation. The public key is sent from the client to the server, ensuring better security.
- **State storage elimination for sessions:** With the migration of all session-related information to the **session file**, there is no longer a need to store session data in the **state**. This change significantly reduces the risks associated with inconsistencies and vulnerabilities previously introduced by state storage.
- **Enhanced session security:** All data required for session communication with the repository, whether through the *Authenticated API* or the *Authorized API*, is now encapsulated in the **session file**. This file includes:
  * The `session_id`, uniquely identifying the session.
  * A nonce (number used once) to protect against replay attacks.
  * The private key of the session's subject, enabling secure communication with the server.
- **Restricted document retrieval:** An improvement was implemented in the file retrieval functionality to prevent unauthorized access to files outside the designated shared documents folder. Previously, a vulnerability in the `get_file` method allowed users to request any file on the system, such as `repository.py`, posing significant security risks.

  The update ensures that only files within the `files` folder can be accessed by introducing the following validation mechanism:
  * The absolute path of the allowed folder (`files`) is retrieved.
  * The absolute path of the requested file is computed.
  * It is verified whether the file's path starts with the allowed folder's path.

  If the validation fails, the system returns an error indicating the file is not found, effectively preventing access to unauthorized files and mitigating the previous vulnerability.

4.4.3. Delivery2

- **Modification to the `drop_role` command:** Changes were implemented to ensure that when a role is removed, the user immediately loses all associated privileges.
- **Restrictions on the `assume_role` command:** It is no longer possible to assume a role that is suspended, strengthening permission control.

4.4.4. Delivery3

- **Improved documentation:** The report was updated to include:
  * A detailed explanation of how messages are exchanged between the client and the repository.
  * A description of the authentication process for a subject.
  * A new section dedicated to *Role Management*, highlighting the changes and improvements implemented.
  * The analysis based on *Chapter V3 of the Application Security Verification Standard (ASVS)* was restructured and conducted with greater detail, ensuring a more explicit and comprehensive explanation of the implemented security measures.

5. Conclusion

This report presented the development of a secure document repository for organizations, focusing on encryption, session management, and authentication. While functional, there is still room for improvement in session management and authentication to make them more robust.

The project was a valuable experience, helping us understand how to securely encrypt data and effectively manage keys. Despite the challenges, it was essential for strengthening our knowledge of information security.