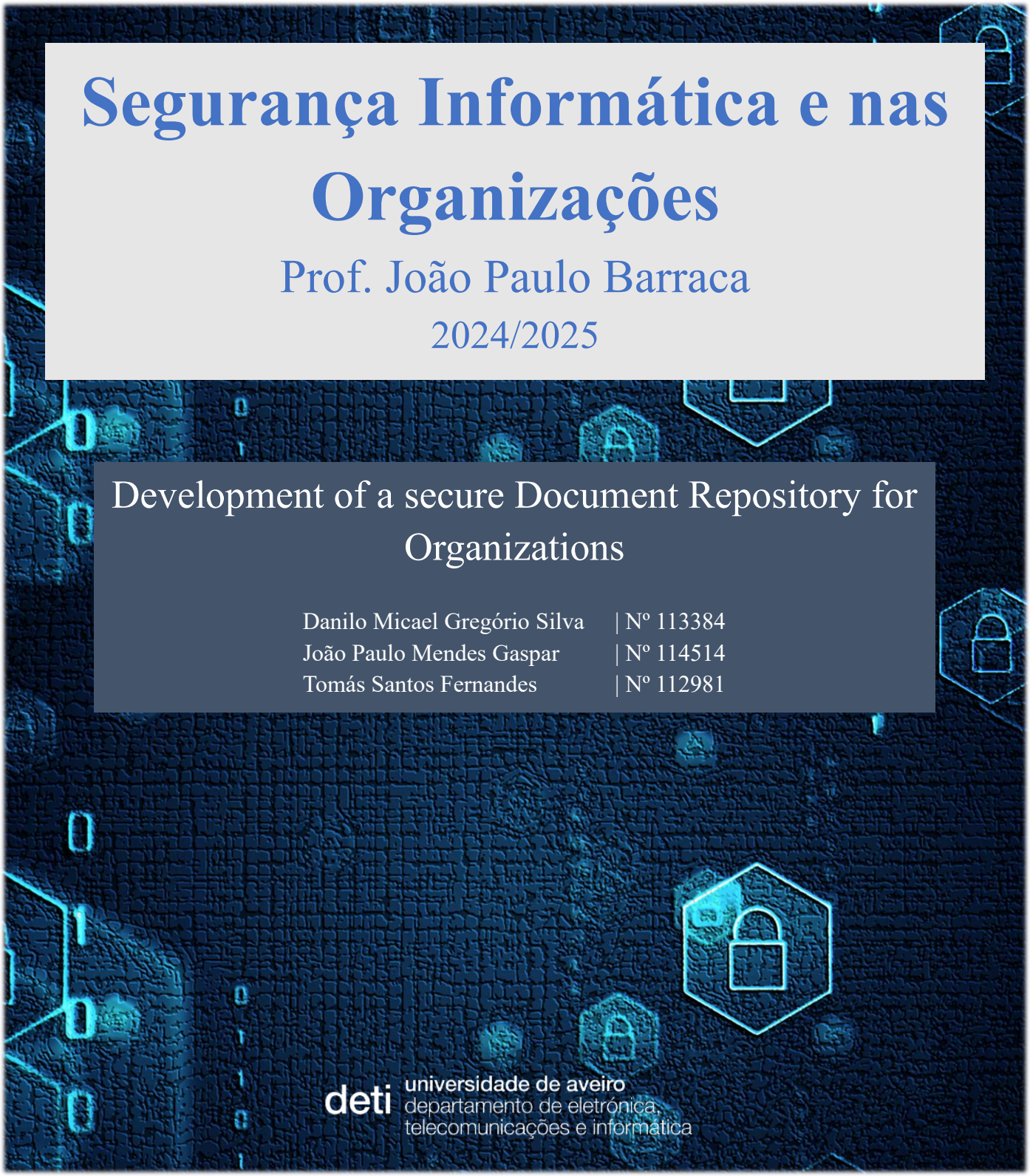
Segurança Informática e nas Organizações

Prof. João Paulo Barraca

2024/2025

Development of a secure Document Repository for Organizations

Danilo Micael Gregório Silva | Nº 113384
João Paulo Mendes Gaspar | Nº 114514
Tomás Santos Fernandes | Nº 112981



deti universidade de aveiro
departamento de eletrónica, telecomunicações e informática

Table of Contents

Introduction

Within the scope of the course "*Segurança Informática e nas Organizações*", this project focuses on the development of a secure document repository designed for organizational use. The repository aims to facilitate the safe sharing of documents among multiple users while ensuring the confidentiality, integrity, and availability of sensitive data.

The repository implements a robust system for managing documents and their associated metadata. While metadata is primarily public, sensitive components, such as encryption information, are securely protected. Documents are encrypted prior to storage, ensuring that their contents remain confidential, even in the event of unauthorized access.

To enforce access control, the repository leverages a role-based access control (*RBAC*) system, enabling fine-grained permissions for actions such as uploading, reading and deleting documents or modifying access control lists (*ACLs*). Additionally, the project ensures compliance with key security principles, providing resilience against common threats such as eavesdropping, impersonation, data manipulation, and replay attacks.

The implementation of the repository includes a well-defined API, enabling seamless interaction between clients and the system. It supports operations such as document management, session creation, and role assignment while adhering to strict security guidelines. By integrating advanced cryptographic techniques and a layered security approach, the repository aims to address the challenges of secure document management in organizational environments.

Implemented Features

In this section, we will **describe the features implemented** in our project, detailing the design and implementation of commands, the approach used for data encryption, and the decisions made regarding security. For each type of command, we'll explain its purpose, how it was implemented, and the security measures integrated into its design.

Encryption played a key role in our system, ensuring data confidentiality and integrity. We will discuss the reasons behind our choice of algorithms, the impact of these decisions on performance, and how we ensured data protection in different usage contexts.

This section is divided into four main parts: **Local Commands**, **Anonymous API Commands** (where we also detail how we made this communication secure, namely on "**Securing Anonymous Communication – General Approach**" and "**Securing Anonymous Communication – Creating a Session**"), **Authenticated API Commands** and **Authorized API Commands** (after which we specify how we secure these last two types of operations, on "**Securing Authenticated and Authorized Communication**").

Local Commands

This subsection explores the commands that can be executed locally without requiring interaction with the repository. These commands are designed to perform critical operations securely and efficiently. In addition to describing each command's functionality, we explain the underlying technical decisions and the rationale behind choosing specific algorithms and encryption modes.

Implemented Commands

1. **rep_subject_credentials <password> <credentials file>**
   - **Functionality**: This command generates a key pair for a subject and stores it in a credentials file.
   - **How it works**: The key pair is generated using **Elliptic Curve Cryptography (ECC)**, and the private component is encrypted with a password provided by the user.
2. **rep_decrypt_file <encrypted file> <encryption metadata>**
   - **Functionality**: This command decrypts a file and outputs its contents to *stdout*, ensuring integrity control based on the encryption metadata.
   - **How it works**: The file is decrypted using the encryption data, including key, iv, algorithm and mode, specified in the metadata. Currently, **AES-256 in CBC mode** is used.

Technical Decisions

Use of ECC for both Subject and Repository credentials

Elliptic Curve Cryptography (*ECC*) was selected for this project as the foundation for both **digital signatures** and **key exchange** operations due to its combination of high security, efficiency, and versatility.

High Security with Small Key Sizes

ECC provides robust security with significantly smaller key sizes compared to traditional algorithms like RSA. For instance:

- A 256-bit ECC key offers equivalent security to a 3072-bit RSA key.

- This compact key size reduces computational overhead while maintaining strong cryptographic guarantees, making ECC particularly suitable for environments with performance constraints or limited resources.

Efficient Key Exchange

ECC is highly efficient for key exchange protocols, such as **Elliptic Curve Diffie-Hellman** (ECDH). In this project, ECC is used to establish shared secrets securely between parties for either anonymous, authenticated or authorized interactions.

While the ECDH approach is robust, it may be vulnerable to certain attacks, such as *Man-in-the-Middle* (*MitM*) attacks, when used in isolation. To mitigate this risk, **we complement the key exchange process by incorporating digital signatures**. By signing the ephemeral public keys exchanged during the ECDH process, we ensure the authenticity of the communicating parties, effectively eliminating the possibility of such attacks.

Key exchange using ECC minimizes the data transmitted over the network and reduces the computational load on both the client and server, enabling fast and secure session establishment.

Secure Digital Signatures

ECC is also used for generating digital signatures through **ECDSA** (*Elliptic Curve Digital Signature Algorithm*). ECC-based signatures offer:

- **Authentication:** Ensuring that data originates from a verified source.
- **Compact signatures:** Smaller signature sizes compared to RSA or DSA, reducing storage and transmission costs.
- **Resistance to forgery:** Due to the mathematical properties of elliptic curves, ECC signatures are highly resistant to cryptographic attacks.

Flexibility in Implementation

The use of ECC in this project provides flexibility in handling both key exchange and authentication through signatures. This dual-purpose capability allows the system to achieve verification of data integrity and authenticity through digital signatures (ECDSA) and secure session establishment via shared secrets (ECDH).

Alignment with Modern Cryptographic Standards

ECC is widely regarded as a modern cryptographic standard, **endorsed by organizations like NIST** and recommended for use in securing sensitive

communications. Its adoption ensures the project aligns with best practices and maintains compatibility with contemporary cryptographic ecosystems.

AES-256-CBC Mode

The decision to use *AES-256-CBC* in this project is rooted in its strong security guarantees when implemented with best practices and its alignment with the project's overall design philosophy. Below, we state the main reasons for this decision:

Security of AES-256

AES-256 is widely recognized as one of the most secure symmetric encryption algorithms available today. It has undergone extensive scrutiny by the cryptographic community and is **approved by standards organizations such as NIST** for securing sensitive data. With a 256-bit key, AES provides a high level of resistance against brute-force attacks, making it an ideal choice for encrypting sensitive information in this project.

Security of CBC Mode

While the *Cipher Block Chaining* (*CBC*) mode can be vulnerable to certain attacks (e.g., padding oracle attacks) if improperly implemented, it remains secure when paired with best practices. In this project, we ensure that each encryption operation follows these best practices by, for example, using a unique, random and cryptographically secure Initialization Vector (IV). This prevents the reuse of IVs, which is critical for maintaining the confidentiality of encrypted data and mitigating risks associated with patterns in the ciphertext.

Why Not Use Authenticated Modes Like GCM?

Authenticated encryption modes like *Galois/Counter Mode* (*GCM*) provide both encryption and integrity/authentication in a single operation, which, for some cases, may be more secure and efficient. However, in this project, we have opted to separate these concerns. Authentication and integrity are ensured through other mechanisms, such as the use of signatures or authenticated shared secrets providing integrity mechanisms, such as *HMACs* (*Hash-based Message Authentication Codes*). This separation allows for more flexibility in managing authentication and encryption independently, which aligns with the project's specific requirements and design goals.

Anonymous API Commands

Here, we delve into the commands accessible through the API that do not require authentication/authorization. This section provides an overview of the implemented commands, and the security measures in place to prevent abuse or unauthorized access. Decisions regarding data handling and minimal encryption for anonymous interactions are also discussed.

Implemented Commands

1. **rep_create_org <organization> <username> <name> <email> <public key file>**
   - **Functionality**: This command creates an organization in a Repository and defines its first subject.
2. **rep_list_orgs**
   - **Functionality**: This command lists all organizations defined in a Repository.
3. **rep_create_session <organization> <username> <password> <credentials file> <session file>**
   - **Functionality**: This command creates a session for a username belonging to an organization and stores the session context in a file.
4. **rep_get_file <file handle> [file]**
   - **Functionality**: This command downloads a file given its file handle. The file contents are written to stdout or to the file referred to in the optional last argument.

Securing Anonymous Communication – General Approach

This subsection explains the cryptographic process used to secure communication in anonymous interactions and the mechanism of creating a secure session.

The following points explain the secure communication in anonymous interactions:

1. **Key Exchange with ECDH**
   i. The client generates a random public key and sends it in plain text to the server.
   ii. The server creates its own random public key and generates the shared secret using the client's public key.
   iii. The server signs its generated public key with the repository's private key.
   iv. Then, sends it to the client.

v. The client verifies the server's signature, ensuring that the communication is being established with a trustworthy source.
vi. After verification, the client generates the shared secret.

2. **Encrypted Message Exchange**
   i. Using the shared secret, the client encrypts messages to the server, ensuring confidentiality.
   ii. The server decrypts the message and encrypts its response using the shared secret.
   iii. The client decrypts the server's response for further processing.
3. **Security Assurance**
   i. This process ensures the confidentiality of communication and protects against impersonation attacks (on the client side).
   ii. Since the client remains anonymous, server-side authentication of the client is unnecessary.

The following diagram takes a closer look at how this is made in the code, showing the general approach to encrypt (and, on the server side, authenticate) anonymous communication.

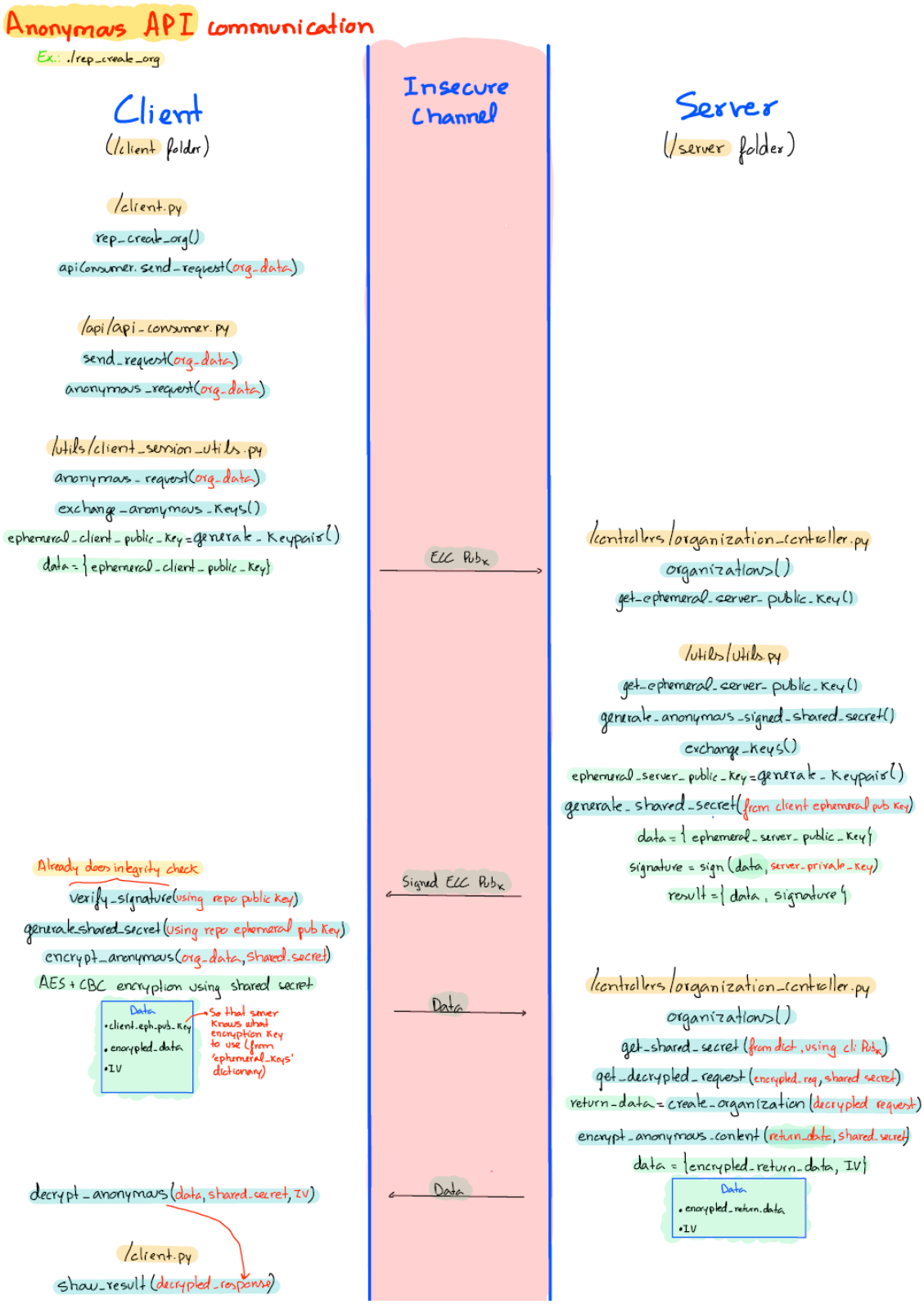


Diagram 1 – Anonymous API Communication

Securing Anonymous Communication – Creating a Session

Now, we'll explain all the process of securely creating a session. We decided to include this detailed explanation because this command has a particular flow, which we considered that would be better understood followed by a visual representation, also in diagram form. This diagram can be found after the following explanation of all the steps upon the execution of this command.

1. The subject generates a random ECC public key, used to obtain the **session shared key**. We refer to the ECC public keys generated on both sides as **session public keys**.
2. Having already shared his credentials with the repository, when registering to an organization, the subject **signs** the session info (which includes his generated session public key) he wants to send to the server (namely the organization and username).
3. Then, as this **still is an anonymous communication**, the same process described on *Securing Anonymous Communication – General Approach* is used to generate an ephemeral shared secret, only for this command (this is, just to take the session - signed - information confidentially from the client to the server). Here, we refer to the ECC public keys generated on both sides, which are only used to agree on the shared secret to secure this anonymous interaction, as **ephemeral**.
4. Once the **ephemeral** secret is agreed, the client sends the signed session info to the server, securely encrypted.
5. The server then decrypts the message.
6. With the session info decrypted, on ***create_session()*** function, the server **verifies the signature** using the public key of the client on that organization.
7. After that, he generates his own **session public key**.
8. Then, the server generates the **session shared key/secret** using the **session public key** of the client.
9. The server then creates a response message with all the session information for the client's session file. This information also includes the **session public key** of the server, so that the client can generate the **session shared key/secret** on his side.
10. The server **signs** the response message with the **repository private key**.
11. Then, all this information is encrypted with the **ephemeral shared key/secret** agreed for this command execution.
12. The server sends the encrypted message to the client.
13. The client decrypts the message.
14. Then, he **verifies the signature** using the **repository public key**.
15. The client generates the **session shared key/secret** using the **session public key** of the server.
16. Finally, the client stores the session information in the session file.

As this command is the only anonymous one that requires the server to verify that the subject requesting to create a session is actually the subject with those credentials on the organization, and then, the corresponding response with the session info requires the client to verify that it was sent by the actual server, two more authentication processes using signatures are implemented. So, for readability, a "*Ax.x*" mark on the left side of each signature-related line on the diagram was used to identify the signature ("*Ax.1*") and the corresponding verification ("*Ax.2*"). Here is the visual representation:

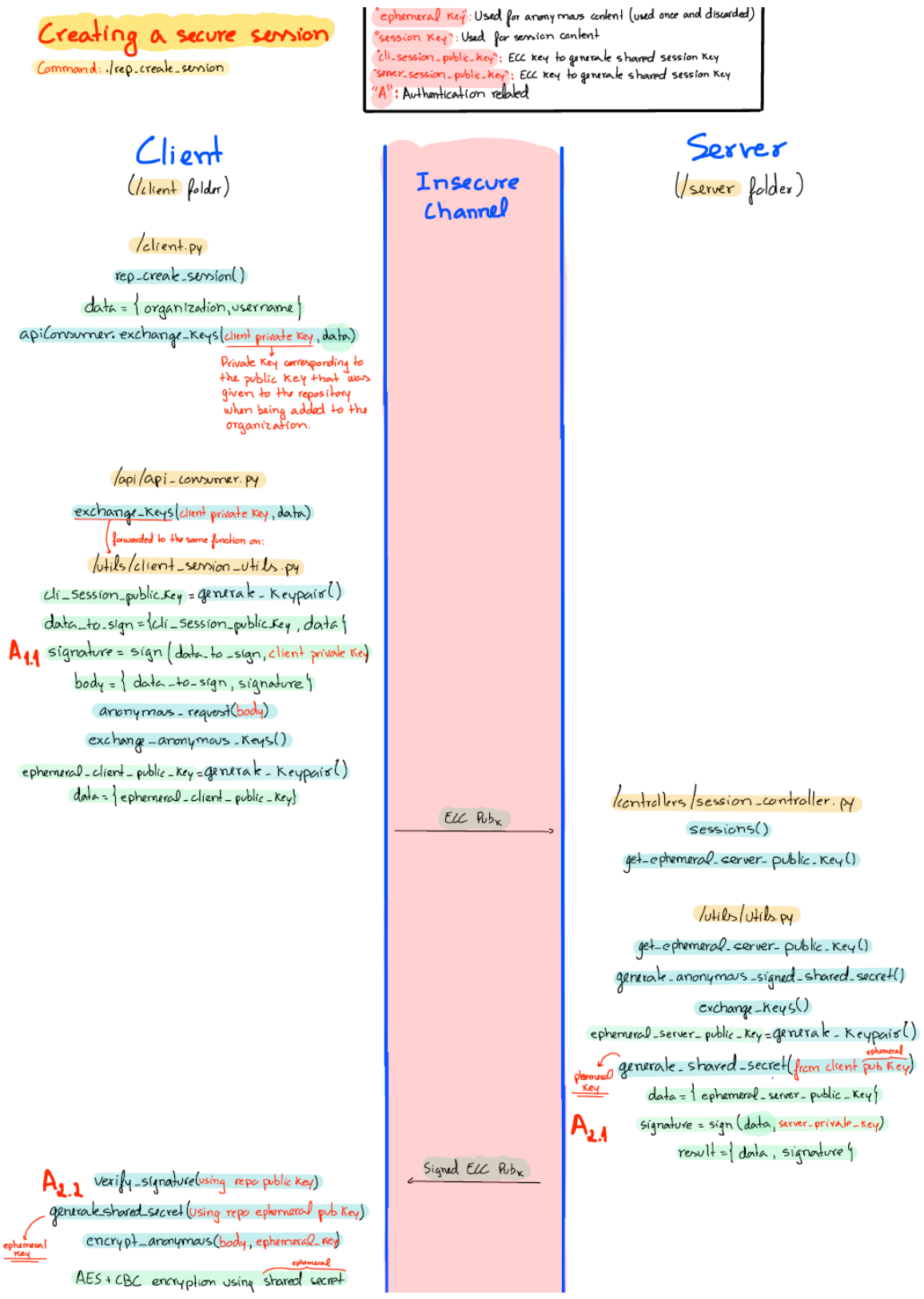

Creating a secure session
Command: ./rep_create_session
"ephemeral key": Used for anonymous content (used once and discarded)
"session key": Used for session content
"cli_session_public_key": ECC key to generate shared session key
"server_session_public_key": ECC key to generate shared session key
"A": Authentication related
Client
(/client folder)
Insecure Channel
Server
(/server folder)
/client.py
rep_create_session()
data = {organization, username}
apiConsumer.exchange_Keys(client private key, data)
Private key corresponding to the public key that was given to the repository when being added to the organization.
/api/api_consumer.py
exchange_Keys(client private key, data)
forwarded to the same function on:
/utils/client_session_utils.py
cli_session_public_key = generate_Keypair()
data_to_sign = {cli_session_public_key, data}
A1.1
signature = sign(data_to_sign, client private key)
body = {data_to_sign, signature}
anonymous_request(body)
exchange_anonymous_Keys()
ephemeral_client_public_key = generate_Keypair()
data = {ephemeral_client_public_key}
ECC Pubx
/controllers/session_controller.py
sessions()
get_ephemeral_server_public_key()
/utils/utils.py
get_ephemeral_server_public_key()
generate_anonymous_signed_shared_secret()
exchange_Keys()
ephemeral_server_public_key = generate_Keypair()
generate_shared_secret(from client pub Key)
ephemeral
ephemeral Key
data = {ephemeral_server_public_key}
A2.1
signature = sign(data, server_private_key)
result = {data, signature}
Signed ECC Pubx
A2.2
verify_signature(using repo public key)
generate_shared_secret(using repo ephemeral pub key)
ephemeral Key
encrypt_anonymous(body, ephemeral_key)
AES + CBC encryption using shared secret
ephemeral

Diagram 2 – Creating a Session using Anonymous Communication

Authenticated API Commands

Implemented Commands

These commands allow the management of roles, subjects, permissions, and documents within an authenticated session:

i. **Role Management**
   1. **rep_assume_role <session file> <role>**
      - **Functionality**: Requests a role for the current session.
   2. **rep_drop_role <session file> <role>**
      - **Functionality**: Releases a role for the current session.
   3. **rep_list_roles <session file>**
      - **Functionality**: Lists the roles associated with the current session.

ii. **Subject Management**
   1. **rep_list_subjects <session file> [username]**
      - **Functionality**: Lists all subjects in the organization or filters by a specific username.
   2. **rep_list_role_subjects <session file> <role>**
      - **Functionality**: Lists the subjects assigned to a specific role.
   3. **rep_list_subject_roles <session file> <username>**
      - **Functionality**: Lists the roles assigned to a specific subject.

iii. **Permission Management**
   1. **rep_list_role_permissions <session file> <role>**
      - **Functionality**: Lists the permissions associated with a specific role.
   2. **rep_list_permission_roles <session file> <permission>**
      - **Functionality**: Lists roles that have a given permission.

iv. **Document Management**
   1. **rep_list_docs <session file> [-s username] [-d nt/ot/et date]**
      - **Functionality**: Lists documents in the organization, optionally filtered by subject and date. Filters include: nt (Newer Than), ot (Older Than), et (Equal To).

Authorized API Commands

Implemented Commands

These commands enable the management of subjects, roles, permissions, and documents within an organization. They require specific permissions to execute, ensuring that only authorized users can perform these operations:

i. **Subject Management**

1. **rep_add_subject <session file> <username> <name> <email> <credentials file>**
   - **Functionality**: This command adds a new subject to the organization with which I have currently a session. By default, the subject is created in active status.
   - **Required Permission**: **SUBJECT_NEW**
2. **rep_suspend_subject <session file> <username>**
   - **Functionality**: This command changes the status of a subject in the organization with which I have currently a session.
   - **Required Permission**: **SUBJECT_DOWN**
3. **rep_activate_subject <session file> <username>**
   - **Functionality**: This command changes the status of a subject in the organization with which I have currently a session.
   - **Required Permission**: **SUBJECT_UP**

ii. **Role Management**

1. **rep_add_role <session file> <role>**
   - **Functionality**: This command adds a role to the organization with which I have currently a session.
   - **Required Permission**: **ROLE_NEW**
2. **rep_suspend_role <session file> <role>**
   - **Functionality**: This command changes the status of a role in the organization with which I have currently a session.
   - **Required Permission**: **ROLE_DOWN**
3. **rep_reactivate_role <session file> <role>**
   - **Functionality**: This command changes the status of a role in the organization with which I have currently a session.
   - **Required Permission**: **ROLE_UP**
4. **rep_add_permission <session file> <role> <username>**
   - **Functionality**: This command changes the properties of a role in the organization with which I have currently a session, by adding a subject.

- **Required Permission**: **ROLE_MOD**

5. **rep_remove_permission <session file> <role> <username>**
   - **Functionality**: This command changes the properties of a role in the organization with which I have currently a session, by removing a subject.
   - **Required Permission**: **ROLE_MOD**
6. **rep_add_permission <session file> <role> <permission>**
   - **Functionality**: This command changes the properties of a role in the organization with which I have currently a session, by adding a permission.
   - **Required Permission**: **ROLE_MOD**
7. **rep_remove_permission <session file> <role> <permission>**
   - **Functionality**: This command changes the properties of a role in the organization with which I have currently a session, by removing a permission.
   - **Required Permission**: **ROLE_MOD**

iii. Document Management

1. **rep_add_doc <session file> <document name> <file>**
   - **Functionality**: This command adds a document with a given name to the organization with which I have currently a session. The document's contents are provided as parameter with a file name.
   - **Required Permission**: **DOC_NEW**
2. **rep_get_doc_metadata <session file> <document name>**
   - **Functionality**: This command fetches the metadata of a document with a given name to the organization with which I have currently a session. The output of this command is useful for getting the clear text contents of a document's file.
   - **Required Permission**: **DOC_READ**
3. **rep_get_doc_file <session file> <document name> [file]**
   - **Functionality**: This command is a combination of ***rep_get_doc_metadata*** with ***rep_get_file*** and ***rep_decrypt_file***. The file contents are written to stdout or to the file referred in the optional last argument.
   - **Required Permission**: **DOC_READ**
4. **rep_delete_doc <session file> <document name>**
   - **Functionality**: This command clears ***file_handle*** in the metadata of a document with a given name on the organization with which I have currently a session. The output of this command is the ***file_handle*** that ceased to exist in the document's metadata.

- **Required Permission**: **DOC_DELETE**

5. **rep_acl_doc <session file> <document name> [+/-] <role> <permission>**
    - **Functionality**: This command changes the ACL of a document by adding (+) or removing (-) a permission for a given role.
    - **Required Permission**: **DOC_ACL**

Securing Authenticated and Authorized Communication

To ensure secure communication between the client and server during a session, we use **ECDH (Elliptic-Curve Diffie-Hellman)** to generate a **64-bit shared secret**. The security of the session is primarily determined at its creation moment, involving the several steps described previously, on *Securing Anonymous Communication – Creating a Session*.

By following this process, the client and server establish a secure session, with all subsequent communication encrypted using the session's shared secret, successfully providing authenticity and confidentiality. The following diagrams illustrate the key stages of **encryption** and **decryption**:

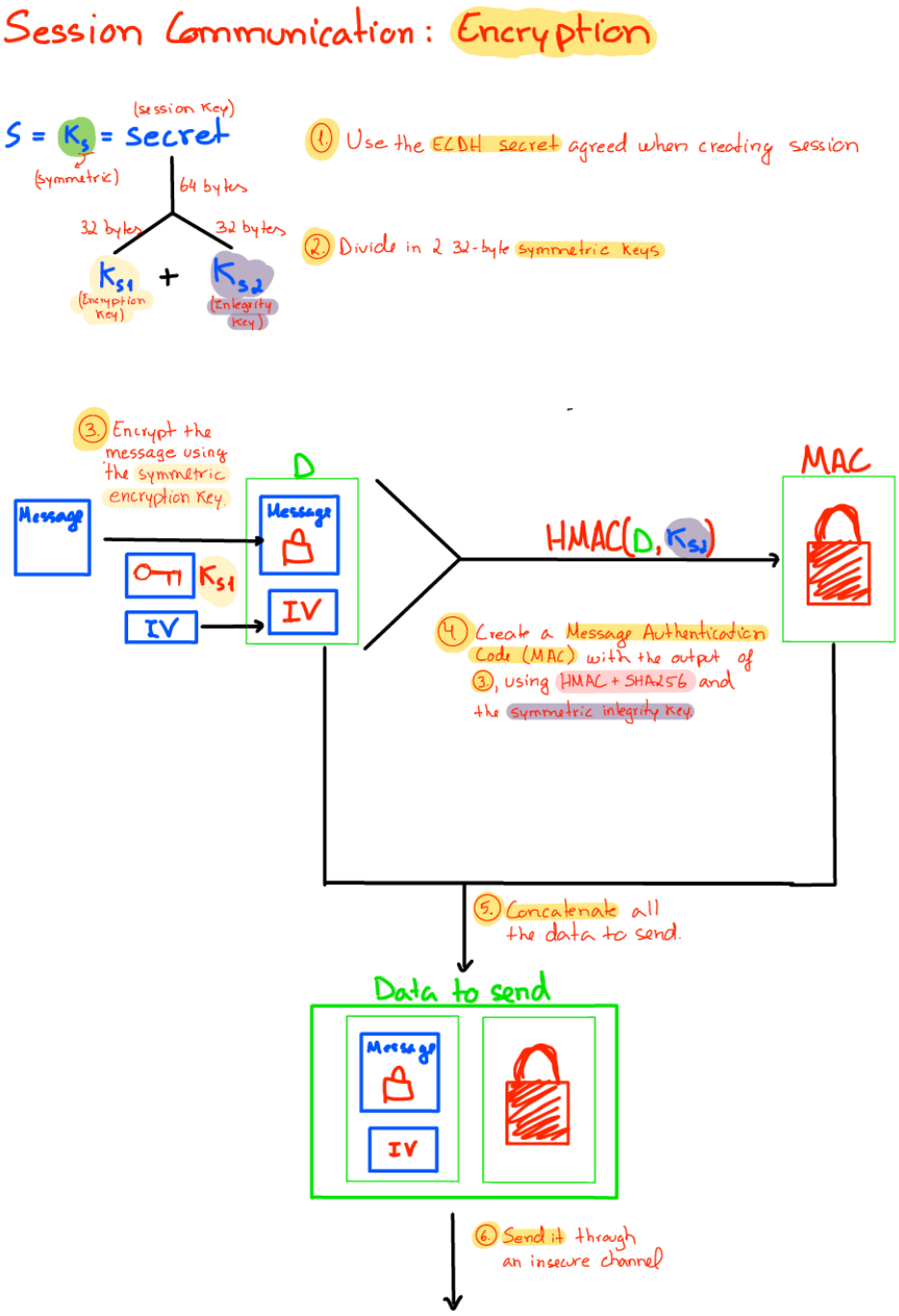


Diagram 3 – Session Communication: Encryption

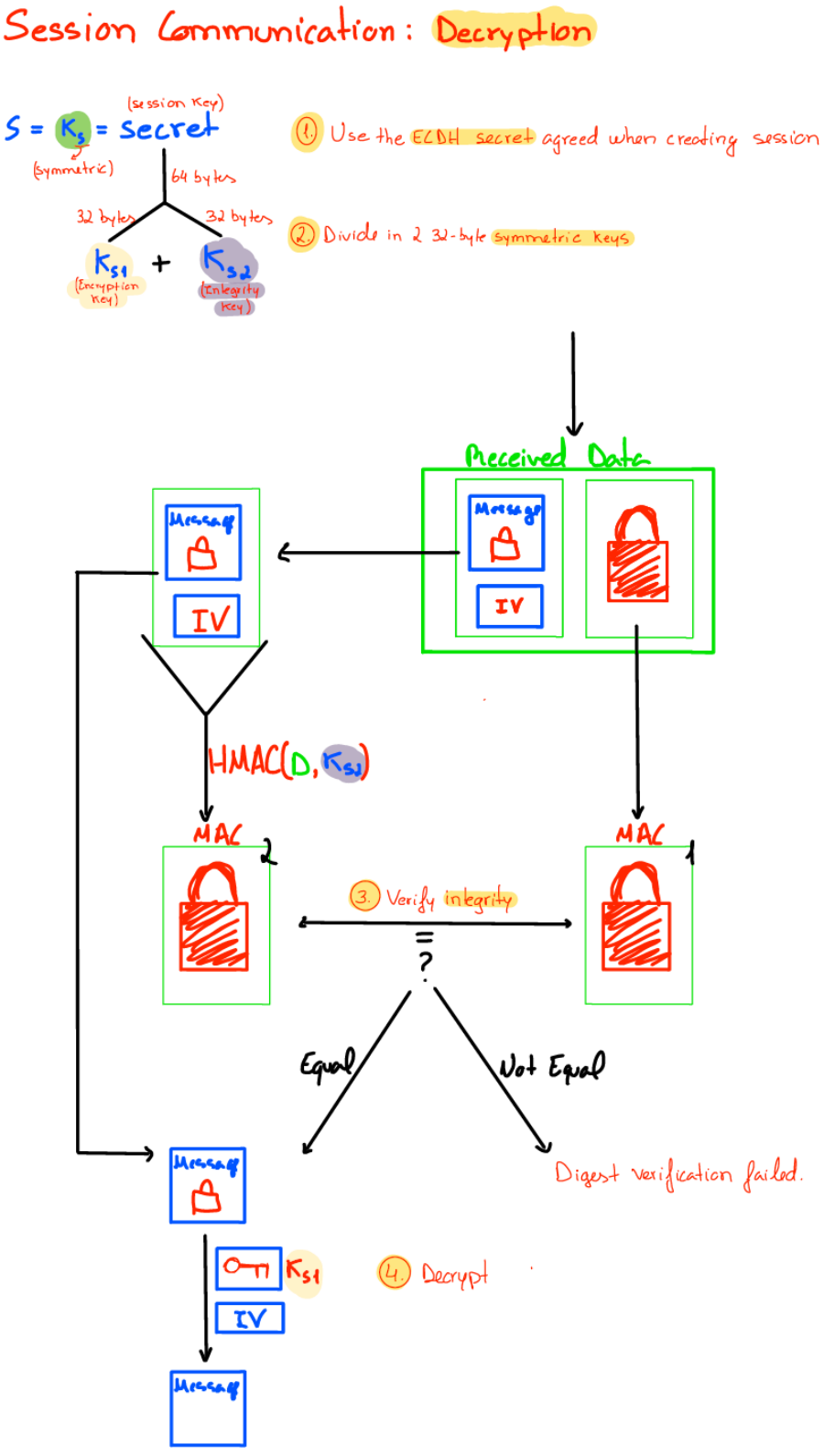


Diagram 4 – Session Communication: Decryption

Key Functional and Security Specifications

This subsection outlines the system's main security and operational features. It discusses requisites for organizational roles and permissions, session management, file encryption, and repository integrity. Each topic is broken down to explain its implementation, including key generation, hashing techniques, and role-based access controls. The rationale behind these security measures is explored, highlighting their role in ensuring the system's robustness and compliance with best practices.

All the commands referred were completely implemented, with all the wanted requirements, such as the following ones (for the operations that need a specific assumed role, it's implicit that the subject needs to be logged in on an active session with the organization):

- **Roles**:
  - Act as groups of subjects.
  - Have a **name**.
  - Are related to one ACL (either document or organization one).
  - Have a **set of permissions**.
  - Have a **list of subjects** associated with it.
  - Have a status, which tells if it is suspended or not.
- **Documents** can be added to the repository.
  - Only subjects assuming a session role with **DOC_NEW** permission are allowed to upload documents.
  - When a document file is uploaded, it is encrypted with a given algorithm and mode.
    - The key to encrypt the file is randomly generated.
  - File integrity is checked after being received by the Repository.
    - When a user uploads a file, he also uploads a ***file_handle*** that consists of the **digest of the original file contents** which is used by the server to carry out an integrity check.
  - The public documents' metadata is stored in plaintext by the Repository, and is publicly available.
  - A document has an ACL that links roles to permissions on that document (the next point states which are the document permissions).
  - When a subject uploads a document to the repository, one of the roles he is assuming at the moment of creation is assigned with all the document permissions for that document:
    - **DOC_READ** to allow subjects to read the file contents.
    - **DOC_DELETE** to allow subjects to delete the file.

        - **DOC_ACL** to allow subjects to modify the document ACL, by adding or removing *Role-Permission* connections on that document.
    - When a document is deleted:
        - The information is not destroyed, it just clears the ***file_handle*** and registers the subject who deleted it in the *deleter* field of the document's metadata.
        - The file is always accessible by subjects who have the file handle of the original file.
    - There is always, at least, one role for each document with the permission **DOC_ACL**.
    - A subject assuming a role with **DOC_ACL** permission for a specific document can add/remove permissions of that document to/from a role.
    - Documents' metadata is stored on the app's SQLite database, while their files are stored in a different physical storage, which is a file system on the server side.
- A subject can download a document from the repository, as long as he is assuming a session role with the permission **DOC_READ** for that document.
    - This is done by having already the document's metadata or fetching a new one (if the document was deleted, in order to successfully get the file contents, the subject must already possess the file handle; if it wasn't deleted, all the necessary info comes in the metadata response).
    - Then, the subject fetches the file using the file handle.
    - Using the *decrypt command* and the encryption data provided by the metadata, he decrypts it.
    - Upon decryption, the *file_handle* provided in the metadata is used to carry out an integrity check.
    - All these steps may be done with only one command: *rep_get_doc_file*.
- To delete a document, a subject must be assuming a role with **DOC_DELETE** permission for that document.
    - After deleting a document, the subject receives, confidentially and for future use, its file handle and the encryption metadata.
- Regarding organizations:
    - The repository has a list of organizations.
    - Each of organization has a list of documents.
    - Organizations can be universally listed.
    - Each organization has an ACL.
    - The ACL defines which roles have which permissions.

    - The set of organization's permissions is: **ROLE_ACL**, **SUBJECT_NEW**, **SUBJECT_DOWN**, **SUBJECT_UP**, **DOC_NEW**, **ROLE_NEW**, **ROLE_DOWN**, **ROLE_UP**, **ROLE_MOD**.
- New roles can be created in the context of each organization with the permission **ROLE_NEW**.
    - When an organization is created, its creator becomes a **Manager** of the organization.
    - Permissions can be added to roles by subjects assuming a role with **ROLE_MOD** permission.
    - A subject can be added to a role by another subject assuming a role also with **ROLE_MOD** permission.
    - A subject of an organization is able to query which users have a role and which roles a subject can assume on his organization.
    - A subject of an organization can also query which roles have a permission and which permissions a role has on his organization.
    - The Manager has all organization permissions and cannot have these modified.
    - The only permissions related to the Manager role that can be modified are document ones.
    - Each organization has always, at least, one subject with the role Manager.
        - When a subject is removed from the Manager role, it's ensured that the operation is only successful if other managers in the organization exist.
    - The role Manager and its subjects can never be suspended.
    - Roles can be suspended or reactivated by subjects who assume a role that have the permission **ROLE_DOWN** or **ROLE_UP**, respectively.
    - If a role is suspended, subjects cannot assume that role, and, if there is any subject assuming it on an active session, it is removed from the set of session roles.
- Subjects:
    - Have a unique **username**.
    - Have a unique **email**.
    - Have a **full name**.
    - May be associated to one or more organizations.
    - Have an **Elliptic Curve key pair**, and the public component is used upon a registration on an organization.
    - Can have multiple key pairs, one per organization.
    - May choose between creating a new key pair or using an existing one when being associated to a new organization.
- Subjects can be added to organizations by other subjects assuming a role that has the **SUBJECT_NEW** permission.

  - Subjects can be suspended or reactivated by subjects in the organization who are assuming a role with the permission **SUBJECT_DOWN** or **SUBJECT_UP**, respectively.
  - When a subject is suspended, his association with the corresponding organization is modified and he is prohibited from performing any operations within the given organization.
  - When the subjects of an organization are listed, the corresponding status within that organization is listed too.
  - Is guaranteed that at least one role on the organization has the **ROLE_ACL** permission.
- Sessions have:
  - One or more **keys**.
    - Keys are used for integrity checks and to ensure confidentiality.
    - Secrets are agreed/generated through ECDH and then passed through a Hash Based Key Derivation Function to get a session key with a length of 64 bytes, of which the first 32 are used to encrypt the exchanged data between the server and the client, and the remaining 32 to generate a Message Authentication Code using a HMAC function for integrity of messages.
    - Since the public keys exchanged to generate the shared secret are signed before sending, authenticity and, consequently, confidentiality are ensured, because it ensures that only the subject and the repository know about the secret.
    - These mechanisms prevent eavesdropping, impersonation and manipulation attacks.
  - An **identifier**.
  - An associated **organization**.
  - An associated **subject**.
    - In order to perform operations within an organization, a subject logs in with the credentials he was registered with in that same organization.
    - Subjects have no default role upon logging in into a session.
    - They need to explicitly ask for a role (assume) **they are bound to on the organization**.
    - Subjects can assume multiple roles in a session.
    - When a subject assumes roles, he is allowed to perform actions based on all the permissions of all the roles he is assuming in the current session.
    - Subjects can release a role during the session, losing the permissions associated to it.

        - A subject can maintain multiple active sessions with different organizations.
    - A predefined **Time-To-Live** (TTL) that is refreshed with each interaction. If the TTL is reached, the session expires and becomes inaccessible.
    - Active **session roles** (assumed roles).
    - A **counter on each exchanged message**, which is a strictly increasing integer, verified at each interaction, to prevent out of order messages.
        - If a message has a counter lower than the last message, an error is thrown.
    - A **nonce**, a random value, that ensures, when combined with the message counter, that replay attack is impossible by making each session unique.
- The repository has a well-known public key that is used by the client to verify integrity and source authentication of the returned data.
    - This key is also used to protect anonymous interactions, as explained on *Securing Anonymous Communication – General Approach*.
    - All asymmetric keys, of both the repository and the client, are generated through **Elliptic Cryptography**.
- Sensitive data, is encrypted on the server by a master key before being stored in the database.
    - Data that needs to be encrypted on the repository, such as (as)symmetric keys, files or documents' restricted metadata, are encrypted with a key generated from a password based key derivation function (PBKDF) using the repository password as argument.
    - To create entropy in the generated key used to encrypt the data, a different random salt is passed to the PBKDF each time a key is needed. All the encryption context (salt and IV, for example) is stored alongside the encrypted data for future decryption use.

ASVS – Software Analysis

The **Application Security Verification Standard** (ASVS) is a globally recognized framework designed to guide organizations in developing and accessing secure applications. Developed by the **Open Web Application Security Project** (OWASP), ASVS provides a comprehensive set of security requirements that can be tailored to different levels of application security needs. The framework is divided into multiple verification levels (L1, L2, and L3), each representing increasing rigor and depth of security measures, with Level 3 (L3) being the most comprehensive and intended for applications with the highest security requirements.

This report focuses on **Level 3 (L3)** verification, which is specifically designed for applications handling highly sensitive data, such as financial systems, healthcare platforms, or applications subject to stringent regulatory requirements. The analysis will be conducted under the scope of **V6: Stored Cryptography**, a critical chapter in the ASVS framework.

Scope of Analysis: V6 - Stored Cryptography

Chapter V6 of ASVS outlines the necessary controls for securely implementing cryptographic mechanisms for data at rest. This includes requirements for:

- The proper use of cryptographic algorithms and libraries;
- Secure storage and management of cryptographic keys;
- Mitigating risks associated with weak or deprecated cryptographic methods;
- Ensuring data integrity and confidentiality through robust encryption practices.

For each V6 chapter control, the analysis will:

1. **Assess Control Relevance**: Determine whether the control is applicable to the project concept, providing justification for the decision.
2. **Check Control Implementation**: Verify if the control is addressed by the software.
3. **Provide Evidence of Implementation and Conclusions**: If the control is implemented, present supporting evidence, such as code snippets or configuration settings. Additionally, other kind of evidences, such as references, may be given to justify some conclusions.

4. **Identify Missing Controls**: If the control is not implemented, describe the issue in detail and discuss the potential risks and impacts of the missing control on the software's security or functionality.
5. **Propose Solutions**: Offer recommendations or solutions to address any missing or poorly implemented controls.

If the control is partially implemented, all the previous points are approached.

Analysis Objective

The objective of this analysis is to evaluate the software's adherence to ASVS L3 requirements for **stored cryptography**, ensuring the implementation of secure practices for protecting sensitive data at rest. This includes verifying the correct use of cryptographic standards, identifying any potential vulnerabilities, and providing actionable recommendations to strengthen the cryptographic framework.

Through this assessment, the team aims to demonstrate a deep understanding of cryptographic principles. This analysis will ensure the application aligns with best practices for safeguarding sensitive information, thereby reinforcing its resilience against potential threats, and, if there is any control that has not been properly applied, if applicable, the team will purpose a solution.

V6.1 - Data Classification

ASVS Requirement: v4.0.3-6.1.1 *(v<version>-<chapter>.<section>.<requirement>)*

Description

Verify that regulated private data is stored encrypted while at rest, such as Personally Identifiable Information (PII), sensitive personal information, or data assessed likely to be subject to EU's GDPR.

Applicable to Project's Concept: Yes

Why is/isn't it applicable to this project?

The project handles several types of sensitive data, including Personally Identifiable Information (PII) such as names and email addresses, as well as highly sensitive data like session keys, symmetric keys, and private keys. Additionally, files' metadata such as encryption algorithms and modes stored in the database are critical for ensuring secure cryptographic operations. Given the nature of this data and its potential exposure to risks, adhering to ASVS Requirement 6.1.1 is essential for protecting the data against unauthorized access, ensuring compliance with regulations like GDPR, and maintaining user trust.

Implemented: Partially

Problem

Our solution is to store encrypted the most important data, **but not all of it**.

The current implementation encrypts several critical pieces of data before storing them, including **private keys**, **symmetric keys**, **document file contents** and **sensitive file metadata**, such as files encryption keys.

However, there are gaps in the encryption strategy, as certain fields that also may be qualified as sensitive remain unencrypted. These include:

- **PII (Subjects' full names and emails):** Currently stored in plaintext.
- **Algorithm and mode in the files restricted metadata:** Although the file encryption key is encrypted, the algorithm and mode fields are stored unencrypted.

Impact

Failing to encrypt sensitive data as required by ASVS 6.1.1 can lead to severe consequences, including non-compliance with GDPR or other privacy laws, resulting in legal penalties and reputational damage. Unencrypted data is vulnerable to breaches, enabling identity theft, phishing, and unauthorized access. Storing encryption details like algorithms and modes in plaintext increases the risk of cryptographic attacks. These exposures can erode

user trust, compromise system security, and lead to cascading failures affecting the entire application.

Solution

We could implement a similar approach for encrypting the subjects' full names and emails as we do for restricted metadata keys. Specifically, when creating a subject, we would store the encryption-related data – such as the initialization vector (IV) and salt used for deriving the encryption key from the repository password – alongside the subject's details. This would allow us to maintain the necessary encryption context.

For the restricted metadata, including the encryption algorithm and mode, we could use the same encryption key that was used to encrypt the corresponding metadata key. This key would also be used to encrypt the details of the encryption algorithm and mode, ensuring consistency and security across all encrypted data.

Evidences

Here is how we store the encrypted files contents:

```python
def create_document(self, name: str, session_id: int, digest: str, encrypted_data: bytes, algorithm: str, mode: str, key: bytes, iv: bytes) -> Document:
    """Create a new document, its ACL, and metadata, and store the encrypted file."""

    try:
        # Step 1: Obtain the session details
        session = self.session_dao.get_by_id(session_id)
        creator = session.subject
        organization = session.organization

        if not organization:
            raise ValueError("Session is not associated with any organization.")

        # Step 2: Generate creation date
        creation_date = datetime.now()

        # Step 3: Generate the document handle and file handle
        document_handle = f"{organization.name}_{name}"
        file_handle = f"{organization.name}_{digest}"
        file_path = os.path.join("data", organization.name, file_handle) + ".enc"

        # Step 4: Store the data of the encrypted document file in a system file
        write_file(file_path, encrypted_data)
```

Figure 1 – Method to create a Document (DocumentDAO.py)

Regarding the encryption of keys, before storing the following distinction is made:

```python
class KeyStoreDAO(BaseDAO):

# ------------------------------

    def create(self, key: bytes, type: str) -> tuple[KeyStore, bytes, bytes] | KeyStore:
        """Create a new KeyStore entry."""
        try:
            if type in ["symmetric", "private"]:
                key, iv, salt = self.encrypt_key(key)

            new_key = KeyStore(key=key, type=type)
            self.session.add(new_key)
            self.session.commit()

            return (new_key, iv, salt) if type in ["symmetric", "private"] else new_key

        except IntegrityError:
            self.session.rollback()
            raise ValueError(f"Key '{key}' already registered.")
```

Figure 2 – Method to create Keys (KeyStoreDAO.py)

```python
def encrypt_key(self, key: bytes) -> tuple[bytes, bytes, bytes]:
    """
    Encrypt the key using AES256 with a derived key from the repository password.
    """

    aes = AES(AESModes.CBC)

    # Derive AES key from the repository password
    repository_password = os.getenv("REPOSITORY_PASSWORD")
    salt = secrets.token_bytes(16)
    aes_key = aes.derive_aes_key(repository_password, salt)

    encrypted_key, iv = aes.encrypt_data(key, aes_key)

    return (encrypted_key, iv, salt)
```

Figure 3 – Method to encrypt a key using AES256 (KeyStoreDAO.py)

Then, we just need to do the following:

- When creating a document (specifically, when storing the restricted metadata, where “key” is the file encryption key):

```
# Step 7: Create the RestrictedMetadata entity
encrypted_metadata_key, iv_encrypted_key, salt = self.key_store_dao.create(key, "symmetric")

metadata = self.restricted_metadata_dao.create(
    document=document,
    algorithm=algorithm,
    mode=mode,
    encrypted_metadata_key_id=encrypted_metadata_key.id,    # Store the key encrypted (used to encrypt the document file)
    iv=iv,                                                  # Store the IV used to encrypt the document file
    salt=salt,                                              # Store the salt used to derive the key used to encrypt the metadata key
    iv_encrypted_key=iv_encrypted_key                       # Store the IV used to encrypt the metadata key
)

# Commit all changes
self.session.commit()
```

Figure 4 – Document metadata (DocumentDAO.py)

- When creating a session, where “key” is the shared secret:

```
encrypted_session_key, iv, salt = self.key_store_dao.create(key, "symmetric")

# Create the session
new_session = Session(
    subject_username=subject_username,
    organization_name=organization_name,
    key_id=encrypted_session_key.id,
    key_salt=salt,
    key_iv=iv,
    nonce=nonce,
    counter=counter
)
```

Figure 5 – Create Session (SessionDAO.py)

ASVS Requirement: v4.0.3-6.1.2

Description

Verify that regulated health data is stored encrypted while at rest, such as medical records, medical device details, or de-anonymized research records

Applicable to Project’s Concept: No

Why is/isn’t it applicable to this project?

This requirement is not applicable because the project does not handle or process regulated health data.

ASVS Requirement: v4.0.3-6.1.3

Description

Verify that regulated financial data is stored encrypted while at rest, such as financial accounts, defaults or credit history, tax records, pay history, beneficiaries, or de-anonymized market or research records.

Applicable to Project's Concept: No

Why is/isn't it applicable to this project?

This requirement is not applicable because the project does not handle or process regulated financial data.

V6.2 – Algorithms

ASVS Requirement: v4.0.3-6.2.1

Description

Verify that all cryptographic modules fail securely, and errors are handled in a way that does not enable Padding Oracle attacks.

Applicable to Project's Concept: Yes

Why is/isn't it applicable to this project?

This requirement is applicable because the project involves decryption of padded encrypted data, as such, it could be vulnerable to padding attacks if the system is not implemented securely.

Implemented: Yes

Evidences

When the server tries to decrypt the payload, in case of any type of failure happening during data decryption, the error code is always 403 (Forbidden) which means that the attacker cannot infer if the error happened due to invalid padding, signature or message order.

```
decrypted_data = decrypt_payload(data, session_key[:32], session_key[32:])
if decrypted_data is None:
    raise ValueError(
        f"Invalid session key",
        HTTP_Code.FORBIDDEN,
        session_key
    )

if (decrypted_data.get("counter") is None) or (decrypted_data.get("nonce") is None):
    raise ValueError(
        f"No counter or nonce provided!",
        HTTP_Code.FORBIDDEN,
        session_key
    )

if not verify_message_order(decrypted_data, counter=session.counter, nonce=session.nonce):
    raise ValueError(
        f"Invalid message order",
        HTTP_Code.FORBIDDEN,
        session_key
    )

if organization_name != session.organization_name:
    raise ValueError(
        f"Cannot access organization {organization_name}",
        HTTP_Code.FORBIDDEN,
        session_key
    )
```

Figure 6 – Method load_session() (server_session_utils.py)

Besides that, for each message sent, a new random IV is generated such that it is not possible to recreate the exact same response and associate it to a specific error, for example, inferring that a specific packet always refers to padding error.

```python
# Get session
try:
    decrypted_data, session, session_key = load_session(data, db_session, organization_name)
except ValueError as e:
    message, code, session_key = e.args
    return return_data("error", message, code, session_key)
```

Figure 7 – Exception handling for loading a session (organization_service.py)

```python
def return_data(key: str, data: str, code: HTTP_Code, session_key: bytes = None):
    if session_key:
        return encrypt_payload({key: data}, session_key[:32], session_key[32:]), code
    return json.dumps({key: data}), code
```

Figure 8 – Method to return encrypted data (utils.py)

```python
def encrypt_payload(data: dict | str, encryption_key: bytes, integrity_key: bytes) -> dict[str, dict]:
    """Encrypts the payload to be sent to the server

    Args:
        data (dict | str): Data to be sent
        encryption_key (bytes): first part of session key, used to encrypt data
        integrity_key (bytes): second part of session key, used to encrypt mac

    Returns:
        dict[str, dict]: Encrypted payload
    """

    # Encrypt data
    if isinstance(data, dict):
        data = json.dumps(data)

    encryptor = AES()
    encrypted_data, data_iv = encryptor.encrypt_data(data.encode(), encryption_key)
```

Figure 9 – Method encrypt_payload() (client_session_utils.py)

```python
def encrypt_data(self, data: bytes, key: bytes) -> tuple[bytes, bytes]:
    """ Encrypts data using AES in the selected mode

    Args:
        data (bytes): Data to be encrypted
        key (bytes): Key to encrypt data

    Returns:
        tuple[bytes, bytes]: (encrypted_data, initialization_vector)
    """

    if self.mode == AESModes.CBC:
        return self.cbc_encrypt(data, key)
```

Figure 10 – Method encrypt_data() (AES.py)

```python
def cbc_encrypt(self, data: bytes, key: bytes) -> tuple[bytes, bytes]:
    """ Encrypts data using AES in CBC mode

    Args:
        data (bytes): data to be encrypted
        key (bytes): key to encrypt data

    Returns:
        tuple[bytes, bytes]: (encrypted_data, initialization_vector)
    """

    iv = os.urandom(16)
    cipher = Cipher(algorithms.AES256(key), self.mode(iv))
```

Figure 11 – Method to encrypt data using AES CBC mode (AES.py)

Finally, the client, when he sends a packet to the server inside a session, also sends the signature of the packet, so before we do any decrypt, the signature is verified, which means that, if any part of the packet is modified then we don't perform a decryption, preventing the attacker from catching any pattern in the time it takes to process the message to know if a decryption happened (and failed due to invalid padding or was successful due to valid padding).

```python
def decrypt_payload(response, encryption_key: bytes, integrity_key: bytes):
    """ Decrypts the payload received from the client

    Args:
        response: Response from the client
        encryption_key: first part of session key, used to encrypt data
        integrity_key: second part of session key, used to encrypt mac

    Returns:
        dict: Decrypted payload
    """

    encryptor = AES()
    received_data = response["data"]
    received_mac = base64.b64decode(response["signature"].encode('utf-8'))

    message_str = received_data["message"]
    data_to_digest = (message_str + received_data["iv"]).encode()

    # Verify digest of received data and check integrity
    if ( not verify_digest(data_to_digest, received_mac, integrity_key) ):
        logging.debug("Digest verification failed")
        return None
```

Figure 12 – Method decrypt_payload() (server_session_utils.py)

```python
def verify_digest(data: bytes, digest: bytes, key: bytes) -> bool:
    """Verify the integrity of the data

    Args:
        data (bytes): Data to be verified
        digest (bytes): Digest to compare with the calculated digest
        key (bytes): Key to hash the data

    Returns:
        bool: True if the data is valid, False otherwise
    """

    new_digest = calculate_digest(data, key)

    return new_digest == digest
```

Figure 13 – Method to verify the integrity of the data (integrity.py)

So, we can conclude that no Padding Oracle Attack is possible at all as decryption only occurs if and only if the message is received untouched from the Client.

ASVS Requirement: v4.0.3-6.2.2

Description

Verify that industry proven or government approved cryptographic algorithms, modes, and libraries are used, instead of custom coded cryptography.

Applicable to Project's Concept: Yes

Why is/isn't it applicable to this project?

This requirement is applicable because the project relies on secure cryptography to ensure secrecy of communications, so industry proven/government approved cryptography secure algorithms are a must to guarantee that the encryption methods used do not compromise system security and are indeed deemed secure.

Implemented: Yes

Evidences

Encryption was made using python's module ***cryptography***, which is a wrapper around ***OpenSSL*** which has a wide range of algorithms approved by ***NIST*** [1].

All symmetric encryption is made using **AES** through **CBC** mode with a **32 bytes key** (256 bits). According to *NIST* [1], AES with CBC is secure when used with keys with 128-bit, 196-bit or 256-bit keys and proper random IVs. (Figure 14, Figure 15, Figure 16).

```python
def send_request(self, endpoint: str, method: str, data=None, sessionKey: bytes = None, sessionId: int = None):
    """Send request to the repository


    Args:
        endpoint (str): Endpoint to send the request
        method (str): Method to send the request
        data (_type_, optional): Data to be sent. Defaults to None.
        sessionKey (bytes, optional): Session key to encrypt the data. Defaults to None.
        sessionId (int, optional): Session ID to be sent. Defaults to None.

    Returns:
        dict: Response from the server
    """

    try:
        received_message = None

        if sessionKey:
            encryption_key, integrity_key = sessionKey[:32], sessionKey[32:]

            logging.debug(f"Sending ({method}) to \'{endpoint}\' in session with sessionKey: {sessionKey},
                          with (decrypted) payload: \"{data}\"")

            # Create and encrypt Payload
            body = self.encrypt_payload(
                data=data,
                encryption_key=encryption_key,
                integrity_key=integrity_key
            )
            body["session_id"] = sessionId

            logging.debug(f"Encrypted payload = {body}")

            # Send Encrypted Payload
            response = requests.request(method, self.rep_address + endpoint, json=body)
            logging.debug(f"Server Response = {response.json()}")
```

Figure 14 – Method to send a request to the server (api_consumer.py)

```python
    def encrypt_payload(self, data, encryption_key, integrity_key):
        return encrypt_payload_utils(data, encryption_key, integrity_key)
```

Figure 15 – encrypt_payload() function caller (api_consumer.py)

```python
def encrypt_payload(data: dict | str, encryption_key: bytes, integrity_key: bytes) -> dict[str, dict]:
    """Encrypts the payload to be sent to the server

    Args:
        data (dict | str): Data to be sent
        encryption_key (bytes): first part of session key, used to encrypt data
        integrity_key (bytes): second part of session key, used to encrypt mac

    Returns:
        dict[str, dict]: Encrypted payload
    """

    # Encrypt data
    if isinstance(data, dict):
        data = json.dumps(data)

    encryptor = AES()
    encrypted_data, data_iv = encryptor.encrypt_data(data.encode(), encryption_key)
```

Figure 16 – Method encrypt_payload() (client_session_utils.py)

Encryption inside a session is also made with AES-CBC (Figure 14), using a 32-bytes key only valid inside the valid session. The key is only known by the server and the client as it was generated by ***ECDH***, where the exchanged public keys were properly authenticated with signatures, and passed through *OpenSSL* ***HKDF*** using ***SHA256*** Algorithm, to derive a 64-bytes shared secret, then split in two 32-bytes keys, the former portion as encryption key and the latter portion as integrity key.

```python
def rep_create_session(org, username, password, credentials_file, session_file):
    """
    rep_create_session <org> <username> <password> <credentials_file> <session_file>
    - This command creates a session for a username belonging to an organization,
    and stores the session context in a file.
    - Calls POST /sessions endpoint
    """

    shared_secret, session_data = apiConsumer.exchange_keys(private_key=private_key, data=data)
    logger.debug(f"SHARED SECRET: {shared_secret}")
```

Figure 17 – Exchange keys in rep_create_session command (client.py)

```python
    def exchange_keys(self, private_key: ec.EllipticCurvePrivateKey, data: dict):
        return exchange_keys_utils(private_key, data, self.rep_address, self.rep_pub_key)
```

Figure 18 – exchange_keys() function caller (api_consumer.py)

```python
def exchange_keys(private_key: ec.EllipticCurvePrivateKey, data: dict, rep_address: str, rep_pub_key):
    ### HANDSHAKE ###
    ecdh = ECC()

    # Generate random Private Key and obtain Public Key
    _, session_public_key = ecdh.generate_keypair()
    session_public_key_str = convert_bytes_to_str(session_public_key)

    # Create packet made of public key and data
    data = {
        "public_key" : session_public_key_str,
        **data
    }

    # Generate Signature
    signature = sign(
        data = data,
        private_key = private_key
    )

    signature_str = convert_bytes_to_str(signature)

    # Build Session creation packet
    body = {
        "data": data,
        "signature": signature_str
    }

    endpoint = "/sessions"
    # Send to the server
    response, received_message = anonymous_request(rep_pub_key, "post", rep_address, endpoint, body)

    if response.status_code not in [201]:
        logging.debug(f"Error: Invalid repository response: {response}")
        print("Error: ", received_message.get("error"))
        sys.exit(ReturnCode.REPOSITORY_ERROR)

    # Verify if signature is valid from repository
    if (not verify_signature(received_message, rep_pub_key)):
        print("Error: Error verifying server authenticity")
        sys.exit(ReturnCode.REPOSITORY_ERROR)

    # If it is valid, finish calculations
    response_data = received_message["data"]
    server_session_public_key = convert_str_to_bytes(response_data["public_key"])
    session_key: bytes = ecdh.generate_shared_secret(server_session_public_key)

    return session_key, response_data
```

Figure 19 - Method exchange_keys() (client_session_utils.py)

```python
def generate_shared_secret(self, peer_public_key: bytes) -> bytes:
    """ Generates a shared secret using the provided peer public key

    Args:
        peer_public_key (bytes): public key of the peer to generate the shared secret with

    Returns:
        bytes: shared secret

    """

    if self.private_key is None:
        raise Exception("No private key has been generated yet!")

    peer_public_key_ecc = serialization.load_pem_public_key(peer_public_key)

    self.shared_key = self.private_key.exchange(ec.ECDH(), peer_public_key_ecc)

    self.derived_key = HKDF(
        algorithm=algorithm,
        length=64, # So then I can use 32 bytes for the encryption key and 32 bytes for the MAC
        salt=None,
        info=b''
    ).derive(self.shared_key)

    return self.derived_key
```

Figure 20 – Method to generate a shared secret (ECC.py)

To assert integrity control, ***HMAC*** with ***SHA2-256*** was used alongside the integrity key, after encrypting the payload (Figure 21).

```python
    encryptor = AES()
    encrypted_data, data_iv = encryptor.encrypt_data(data.encode(), encryption_key)

    data = {
        "message": base64.b64encode(encrypted_data).decode(),     # Data to be sent to the server
        "iv" : base64.b64encode(data_iv).decode(),                # IV used to encrypt the data
    }

    data_to_digest = (data["message"] + data["iv"]).encode()
    digest = calculate_digest(data_to_digest, integrity_key)

    body = {
        "data": data,
        "signature": base64.b64encode(digest).decode('utf-8') # convert_bytes_to_str(digest)
    }

    return body
```

Figure 21 – Method encrypt_payload() (server_session_utils.py)

```python
def calculate_digest(data: bytes, key: bytes) -> bytes:
    """Calculate the digest of the data using SHA256 algorithm

    Args:
        data (bytes): Data to be hashed
        key (bytes): Key to hash the data

    Returns:
        bytes: Hashed data
    """

    digest = hmac.HMAC(key, hashes.SHA256())
    digest.update(data)

    return digest.finalize()
```

Figure 22 – Method to calculate the digest of the data (integrity.py)

The client then verifies the signature (Figure 23) using the integrity key before decrypting the message (Figure 24).

```python
def verify_digest(data: bytes, digest: bytes, key: bytes) -> bool:
    """Verify the integrity of the data

    Args:
        data (bytes): Data to be verified
        digest (bytes): Digest to compare with the calculated digest
        key (bytes): Key to hash the data

    Returns:
        bool: True if the data is valid, False otherwise
    """

    new_digest = calculate_digest(data, key)

    return new_digest == digest
```

Figure 23 – Method to verify integrity of the data (integrity.py)

```python
def decrypt_payload(response, encryption_key: bytes, integrity_key: bytes):
    encryptor = AES()
    received_data = response["data"]
    received_mac = convert_str_to_bytes(response["signature"])

    message_str = received_data["message"]
    data_to_digest = (message_str + received_data["iv"]).encode()

    # Verify digest of received data and check integrity
    if ( not verify_digest(data_to_digest, received_mac, integrity_key) ):
        logging.debug("Digest verification failed")
        return None
```

Figure 24 – Method to decrypt the payload (client_session_utils.py)

All asymmetric signatures were generated and verified using ***ECDSA***, key-pairs are derived through a ***SECP521R1 Curve***, an *ECC curve* (***P-521***), and, although slower than *Curve25519*, displays a higher level of resistance against various attacks. When a user is creating a session, to verify authenticity, he signs the packet with his private key, then, the server, with his public key, verifies the source of the message.

```
    # Verify Signature
    if (not verify_signature(data=data, pub_key=client_pub_key)):
        return return_data("error", f"Invalid signature!", HTTP_Code.BAD_REQUEST)
```

Figure 25 – Verify signature (session_service.py)

```
def verify_signature(response: dict[str, str], public_key: ec.EllipticCurvePublicKey) -> bool:
    """Verifies the signature of a document using the provided public key.

    Args:
        response (dict): Response from the server containing the data and signature
        public_key (ec.EllipticCurvePublicKey): Public key to verify the signature

    Returns:
        bool: True if the signature is valid, False otherwise

    """

    data_str = str(response["data"])
    signature = base64.b64decode(response["signature"])

    try:
        public_key.verify(signature, data_str.encode(), ec.ECDSA(hashing_algorithm))
        logging.debug("Document signature is valid")
        return True
    except InvalidSignature:
        logging.debug("Document signature is not valid")
        return False
```

Figure 26 – Method to verify the signature (auth.py)

All these algorithms – **ECDH**, **HKDF**, **SHA256**, **HMAC with SHA2-256 mode**, **ECDSA with P-521 curve** – are considered secure by ***NIST*** [1]

ASVS Requirement: v4.0.3-6.2.3

Description

Verify that encryption initialization vector, cipher configuration, and block modes are configured securely using the latest advice.

Applicable to Project's Concept: Yes

Why is/isn't it applicable to this project?

This requirement is applicable because the project involves cryptographic operations, and ensuring the correct configuration is essential in maintaining the security of the messages interchanged between client and server.

Implemented: Yes

Evidences

According to *OWASP Cheat Sheet Series* [2], IV must be generated using Cryptographically Secure Pseudo-Random Number Generators (CSPRNG), as they produce a greater amount of entropy, translating to a stronger resistance to being predicted. IVs are generated using python's ***os.urandom*** method that, according to *Python documentation* [3], reads the random data from */dev/urandom*, and also according to the *Python documentation* [3]:

«*The returned data should be unpredictable enough for cryptographic applications*»

```
def cbc_encrypt(self, data: bytes, key: bytes) -> tuple[bytes, bytes]:
    """ Encrypts data using AES in CBC mode

    Args:
        data (bytes): data to be encrypted
        key (bytes): key to encrypt data

    Returns:
        tuple[bytes, bytes]: (encrypted_data, initialization_vector)
    """

    iv = os.urandom(16)
    cipher = Cipher(algorithms.AES256(key), self.mode(iv))
```

Figure 24 – Method to encrypt data using AES CBC mode (AES.py)

```python
def encrypt_payload(data: dict | str, encryption_key: bytes, integrity_key: bytes) -> dict[str, dict]:
    """Encrypts the payload to be sent to the server

    Args:
        data (dict | str): Data to be sent
        encryption_key (bytes): first part of session key, used to encrypt data
        integrity_key (bytes): second part of session key, used to encrypt mac

    Returns:
        dict[str, dict]: Encrypted payload
    """

    # Encrypt data
    if isinstance(data, dict):
        data = json.dumps(data)

    encryptor = AES()
    encrypted_data, data_iv = encryptor.encrypt_data(data.encode(), encryption_key)
```

Figure 25 – Method encrypt_payload() (server_session_utils.py)

As referred before, cipher configurations and block modes (AES-CBC) are configured securely based on *NIST* [1] security policies.

ASVS Requirement: v4.0.3-6.2.4

Description

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

Applicable to Project's Concept: Yes

Why is/isn't it applicable to this project?

This requirement is applicable because the project involves cryptographic operations, and ensuring the use of secure algorithms and modes is critical to maintaining data integrity and security, which leads to, in case a used algorithm is no longer safe, it should be replaceable with a safe algorithm to maintain security.

Implemented: Yes

Evidences

Due to how our code is structured, most of it (at least, the essential parts) can be reconfigured, upgraded or swapped at any time, without significant changes to infrastructure or architecture as the components follow a decoupled structure.

This is evident on the server side, as shown in figures 29, 30 and 31, as all data returned passes through *return_data* method and in figures 32 and 33 as all data received from the client inside a session passes through *load_session* method to be decrypted.

```
# Get session
try:
    decrypted_data, session, session_key = load_session(data, db_session, organization_name)
except ValueError as e:
    message, code, session_key = e.args
    return return_data("error", message, code, session_key)
```

Figure 26 – Exception handling for loading a session (organization_service.py)

```
def return_data(key: str, data: str, code: HTTP_Code, session_key: bytes = None):
    if session_key:
        return encrypt_payload({key: data}, session_key[:32], session_key[32:]), code
    return json.dumps({key: data}), code
```

Figure 27 – Method to return encrypted data (utils.py)

```
def encrypt_payload(data: dict | str, encryption_key: bytes, integrity_key: bytes) -> dict[str, dict]:
    """Encrypts the payload to be sent to the server

    Args:
        data (dict | str): Data to be sent
        encryption_key (bytes): first part of session key, used to encrypt data
        integrity_key (bytes): second part of session key, used to encrypt mac

    Returns:
        dict[str, dict]: Encrypted payload
    """
    encryptor = AES()
    encrypted_data, data_iv = encryptor.encrypt_data(data.encode(), encryption_key)
```

Figure 28 – Method encrypt_payload() (server_session_utils.py)

```
def load_session(data: dict, db_session: SQLAlchemySession, organization_name: str) -> tuple[dict, Session, bytes]:
    """Load the session from the received data and make the necessary verifications:
        - organization
        - message order
        - message integrity
        - message uniqueness

    Args:
        data (dict): Data received from the client
        session_dao (SessionDAO): DAO to access the session
        organization_name (str): Name of the organization

    Returns:
        tuple[dict, Session, bytes]: Decrypted data, Session, Session key
    """

    decrypted_data = decrypt_payload(data, session_key[:32], session_key[32:])
```

Figure 29 – Method load_session() (server_session_utils.py)

```python
def decrypt_payload(response, encryption_key: bytes, integrity_key: bytes):
    encryptor = AES()
    received_data = response["data"]
    received_mac = base64.b64decode(response["signature"].encode('utf-8'))

    message_str = received_data["message"]
    data_to_digest = (message_str + received_data["iv"]).encode()

    # Verify digest of received data and check integrity
    if ( not verify_digest(data_to_digest, received_mac, integrity_key) ):
        logging.debug("Digest verification failed")
        return None

    encrypted_message = base64.b64decode(message_str.encode('utf-8'))
    # Decrypt data
    received_message = encryptor.decrypt_data(
        encrypted_data=encrypted_message,
        key=encryption_key,
        iv=base64.b64decode(received_data["iv"].encode('utf-8'))
    )
    return json.loads(received_message.decode('utf-8'))
```

Figure 30 – Method decrypt_payload() (server_session_utils.py)

On client side this simplicity is also evident, as shown in figures 34, 35 and 36, as all data sent to repository passes through the *send_request* function, where all modifications are made, for both anonymous and session messages. Both are encrypted and decrypted, and every other modification that needs to be done, is made on this function. For example, inside a session, data is encrypted through *api_consumer.encrypt_payload* method and decrypted through *api_consumer.decrypt_payload* method, as shown in figures 34 through 39.

```python
def send_request(self, endpoint: str, method: str, data=None, sessionKey: bytes = None, sessionId: int = None):
    try:
        received_message = None

        if sessionKey:
            encryption_key, integrity_key = sessionKey[:32], sessionKey[32:]

            logging.debug(f"Sending ({method}) to \'{endpoint}\' in session with sessionKey: {sessionKey}, with
            (decrypted) payload: \"{data}\"")

            # Create and encrypt Payload
            body = self.encrypt_payload(
                data=data,
                encryption_key=encryption_key,
                integrity_key=integrity_key
            )
            body["session_id"] = sessionId
```

Figure 31 – Method to send a request to the server (api_consumer.py)

```python
def encrypt_payload(self, data, encryption_key, integrity_key):
    return encrypt_payload_utils(data, encryption_key, integrity_key)
```

Figure 32 – encrypt_payload() function caller (api_consumer.py)

```python
def encrypt_payload(data: dict | str, encryption_key: bytes, integrity_key: bytes) -> dict[str, dict]:
    # Encrypt data
    if isinstance(data, dict):
        data = json.dumps(data)

    encryptor = AES()
    encrypted_data, data_iv = encryptor.encrypt_data(data.encode(), encryption_key)

    data = {
        "message": convert_bytes_to_str(encrypted_data),            # Data to be sent to the server
        "iv" : convert_bytes_to_str(data_iv),                       # IV used to encrypt the data
    }

    data_to_digest = (data["message"] + data["iv"]).encode()
    digest = calculate_digest(data_to_digest, integrity_key)

    body = {
        "data": data,
        "signature": convert_bytes_to_str(digest)
    }

    logger.debug(f"Encrypted payload: {body} with encryption key: {encryption_key} and integrity key: {integrity_key}")
    return body
```

Figure 33 – Method encrypt_payload() (client_session_utils.py)

```python
        # Create and encrypt Payload
        body = self.encrypt_payload(
            data=data,
            encryption_key=encryption_key,
            integrity_key=integrity_key
        )
        body["session_id"] = sessionId

        logging.debug(f"Encrypted payload = {body}")

        # Send Encrypted Payload
        response = requests.request(method, self.rep_address + endpoint, json=body)
        logging.debug(f"Server Response = {response.json()}")

        try:
            received_message = self.decrypt_payload(
                response=response.json(),
                encryption_key=encryption_key,
                integrity_key=integrity_key
            )
            logging.debug(f"Decrypted Server Response = {received_message}")
        except Exception as e:
            received_message = response.json()
            logging.debug(f"Error decrypting server response: {e}")
```

Figure 34 – Exception handling for decrypting payload (api_consumer.py)

```python
    def decrypt_payload(self, response, encryption_key, integrity_key):
        return decrypt_payload_utils(response, encryption_key, integrity_key)
```

Figure 35 – decrypt_payload() function caller (api_consumer.py)

```python
def decrypt_payload(response, encryption_key: bytes, integrity_key: bytes):
    encryptor = AES()
    received_data = response["data"]
    received_mac = convert_str_to_bytes(response["signature"])

    message_str = received_data["message"]
    data_to_digest = (message_str + received_data["iv"]).encode()

    # Verify digest of received data and check integrity
    if ( not verify_digest(data_to_digest, received_mac, integrity_key) ):
        logging.debug("Digest verification failed")
        return None

    encrypted_message = convert_str_to_bytes(message_str)
    # Decrypt data
    received_message = encryptor.decrypt_data(
        encrypted_data=encrypted_message,
        key=encryption_key,
        iv=convert_str_to_bytes(received_data["iv"])
    )
    return json.loads(received_message.decode('utf-8'))
```

Figure 36 – Method decrypt_payload() (client_session_utils.py)

With this, it is possible to conclude that only simple changes would be necessary to modify the implementation at any time. Overall, most changes would be made by only changing the values of variables, such as replacing encryptor AES-CBC mode with encryptor AES-GCM mode (and fundamental requirements of GCM), or values of number of iterations or length of the key on functions like PBKDF.

ASVS Requirement: v4.0.3-6.2.5

Description

Verify that known insecure block modes (i.e. ECB, etc.), padding modes (i.e. PKCS#1 v1.5, etc.), ciphers with small block sizes (i.e. Triple-DES, Blowfish, etc.), and weak hashing algorithms (i.e. MD5, SHA1, etc.) are not used unless required for backwards compatibility.

Applicable to Project's Concept: Yes

Why is/isn't it applicable to this project?

This requirement is applicable because the project involves cryptographic operations, and ensuring the use of secure algorithms and modes is critical to maintaining data integrity and security.

Implemented: Yes

Evidences

As stated, we employ robust and secure tools to safeguard repository operations, adhering to standards approved by entities such as ***OWASP*** [8] and ***NIST*** [1], ensuring compliance with industry best practices.

For encryption, we utilize **ECC** with the **P-521** (*secp521r1*) curve, providing strong security for **asymmetric** cryptographic needs:

```
You, 1 second ago | 1 author (You)
class ECC:
    def __init__(self, curve=ec.SECP521R1()):
        self.curve = curve

# ------------------------------ ECC & ECDH methods ------------------------------ #

    def generate_keypair(self, password: str = None) -> tuple[bytes, bytes]:
        """ Generates an ECC keypair using the specified curve and password

        Args:
            password (str): password to protect the private key (if None, no password is used)

        Returns:
            tuple[bytes, bytes]: tuple containing the private and public keys in PEM format (serialized)
        """

        self.private_key = ec.generate_private_key(self.curve)
        public_key = self.private_key.public_key()
        password = password.encode() if password else None

        # Serialize private key with password protection
        pem_private_key = self.private_key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.PKCS8,
            encryption_algorithm=serialization.BestAvailableEncryption(password) if password else serialization.NoEncryption()
        )

        pem_public_key = public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo
        )

        return pem_private_key, pem_public_key
```

Figure 37 – Class ECC: generate_keypair() method (ECC.py)

For **symmetric** encryption, we implement **AES-256** in **CBC** mode with properly randomized initialization vectors (IVs) and **PKCS#7** padding. We have also explained that the use of CBC mode, instead of an authenticated mode, stems from the fact that authentication is addressed separately through other mechanisms:

```python
def encrypt_data(self, data: bytes, key: bytes) -> tuple[bytes, bytes]:
    """ Encrypts data using AES in the selected mode

    Args:
        data (bytes): Data to be encrypted
        key (bytes): Key to encrypt data

    Returns:
        tuple[bytes, bytes]: (encrypted_data, initialization_vector)
    """

    if self.mode == AESModes.CBC:
        return self.cbc_encrypt(data, key)
    # elif self.mode == AESModes.GCM:
    #     return self.gcm_encrypt(data, key)
    else:
        print("Unsupported mode for AES.")
        return None, None
```

Figure 38 – Method encrypt_data() (AES.py)

```python
You, 2 weeks ago | 1 author (You)
class AES:
    def __init__(self, mode: AESModes = AESModes.CBC):
        self.mode = mode

# ------------------------------

    def cbc_encrypt(self, data: bytes, key: bytes) -> tuple[bytes, bytes]:
        """ Encrypts data using AES in CBC mode

        Args:
            data (bytes): data to be encrypted
            key (bytes): key to encrypt data

        Returns:
            tuple[bytes, bytes]: (encrypted_data, initialization_vector)
        """

        iv = os.urandom(16)
        cipher = Cipher(algorithms.AES256(key), self.mode(iv))

        encryptor = cipher.encryptor()

        padder = padding.PKCS7(algorithms.AES256.block_size).padder()

        padded_data = padder.update(data)
        padded_data += padder.finalize()

        return (encryptor.update(padded_data) + encryptor.finalize(), iv)
```

Figure 39 – Class AES: cbc_encrypt() method (AES.py)

To ensure efficient and secure signing, we use **SHA-256**, a widely recognized and reliable cryptographic hash function. These measures collectively enhance the security and integrity of repository operations:

```python
# ------------------------------

hashing_algorithm = hashes.SHA256()

# ------------------------------

def sign(data: dict, private_key: ec.EllipticCurvePrivateKey) -> bytes:
    """Signs a document using the provided private key and returns the signature.
    Elliptic Curve Digital Signature Algorithm (ECDSA) is used with SHA256 as the hashing algorithm.

    Args:
        data (dict): Data to be signed
        private_key (ec.EllipticCurvePrivateKey): Private key to sign the data

    Returns:
        bytes: Signature of the data

    """

    data_str = str(data)

    signature = private_key.sign(
        data_str.encode(),
        ec.ECDSA(hashing_algorithm)
    )
    return signature
```

Figure 40 – Method sign() (auth.py)

```python
algorithm = hashes.SHA256()

You, 1 second ago | 1 author (You)
class ECC:
    def __init__(self, curve=ec.SECP521R1()):
        self.curve = curve

    def generate_shared_secret(self, peer_public_key: bytes) -> bytes:
        """ Generates a shared secret using the provided peer public key

        Args:
            peer_public_key (bytes): public key of the peer to generate the shared secret with

        Returns:
            bytes: shared secret

        """

        if self.private_key is None:
            raise Exception("No private key has been generated yet!")

        peer_public_key_ecc = serialization.load_pem_public_key(peer_public_key)

        self.shared_key = self.private_key.exchange(ec.ECDH(), peer_public_key_ecc)

        self.derived_key = HKDF(
            algorithm=algorithm,
            length=64, # So then I can use 32 bytes for the encryption key and 32 bytes for the
            salt=None,
            info=b''
        ).derive(self.shared_key)

        return self.derived_key
```

Figure 41 – Class ECC: generate_shared_secret() method (ECC.py)

```python
def calculate_digest(data: bytes, key: bytes) -> bytes:
    """Calculate the digest of the data using SHA256 algorithm

    Args:
        data (bytes): Data to be hashed
        key (bytes): Key to hash the data

    Returns:
        bytes: Hashed data
    """

    digest = hmac.HMAC(key, hashes.SHA256())
    digest.update(data)

    return digest.finalize()
```

Figure 42 – Method to calculate the digest of the data (integrity.py)

ASVS Requirement: v4.0.3-6.2.6

Description

Verify that nonces, initialization vectors, and other single use numbers must not be used more than once with a given encryption key. The method of generation must be appropriate for the algorithm being used.

Applicable to Project's Concept: Yes

Why is/isn't it applicable to this project?

This requirement is relevant because our solution employs symmetric encryption (AES-256 in CBC mode) where an IV is mandatory.

Additionally, a salt is also used to derive an encryption key from the repository password, in order to encrypt the sensitive data to be stored.

Implemented: Yes

Evidences

Regarding the IVs, we ensure that a new initialization vector (IV) is generated each time symmetric encryption is performed. This is demonstrated in the following images:

```python
def encrypt_data(self, data: bytes, key: bytes) -> tuple[bytes, bytes]:
    """ Encrypts data using AES in the selected mode

    Args:
        data (bytes): Data to be encrypted
        key (bytes): Key to encrypt data

    Returns:
        tuple[bytes, bytes]: (encrypted_data, initialization_vector)
    """

    if self.mode == AESModes.CBC:
        return self.cbc_encrypt(data, key)
    # elif self.mode == AESModes.GCM:
    #     return self.gcm_encrypt(data, key)
    else:
        print("Unsupported mode for AES.")
        return None, None
```

Figure 43 – Method encrypt_data() (AES.py)

```python
class AES:
    def __init__(self, mode: AESModes = AESModes.CBC):
        self.mode = mode

# ------------------------------

    def cbc_encrypt(self, data: bytes, key: bytes) -> tuple[bytes, bytes]:
        """ Encrypts data using AES in CBC mode

        Args:
            data (bytes): data to be encrypted
            key (bytes): key to encrypt data

        Returns:
            tuple[bytes, bytes]: (encrypted_data, initialization_vector)
        """

        iv = os.urandom(16)
        cipher = Cipher(algorithms.AES256(key), self.mode(iv))

        encryptor = cipher.encryptor()

        padder = padding.PKCS7(algorithms.AES256.block_size).padder()

        padded_data = padder.update(data)
        padded_data += padder.finalize()

        return (encryptor.update(padded_data) + encryptor.finalize(), iv)
```

Figure 44 – Class AES: cbc_encrypt() method (AES.py)

Example of use:

```python
def encrypt_payload(data: dict | str, encryption_key: bytes, integrity_key: bytes) -> dict[str, dict]:
    """Encrypts the payload to be sent to the client

    Args:
        data (dict | str): Data to be sent
        encryption_key (bytes): first part of session key, used to encrypt data
        integrity_key (bytes): second part of session key, used to encrypt mac

    Returns:
        dict[str, dict]: Encrypted payload
    """

    # Encrypt data
    if isinstance(data, dict):
        data = json.dumps(data)

    encryptor = AES()
    encrypted_data, data_iv = encryptor.encrypt_data(data.encode(), encryption_key)

    data = {
        "message": base64.b64encode(encrypted_data).decode(),    # Data to be sent to the client
        "iv" : base64.b64encode(data_iv).decode(),               # IV used to encrypt the data
    }

    data_to_digest = (data["message"] + data["iv"]).encode()
    digest = calculate_digest(data_to_digest, integrity_key)

    body = {
        "data": data,
        "signature": base64.b64encode(digest).decode('utf-8') # convert_bytes_to_str(digest)
    }

    return body
```

Figure 45 – Method encrypt_payload() (server_session_utils.py)

To achieve this, we use the ***os*** Python library, specifically the ***urandom*** function, which has been previously verified as secure for cryptographic purposes.

Additionally, for "*other single-use numbers*" we use a random salt each time we need to encrypt data to be stored in the repository. This salt is used to derive an AES key from the repository's master key/password. Similar to the IV, the salt is stored alongside the encrypted data to maintain the encryption context when needed.

```python
# Check if the subject exists
subject = self.subject_dao.get_by_username(subject_username)
if not subject:
    raise ValueError(f"Subject with username '{subject_username}' does not exist.")

# Check if the organization exists
organization = self.organization_dao.get_by_name(organization_name)
if not organization:
    raise ValueError(f"Organization with name '{organization_name}' does not exist.")

encrypted_session_key, iv, salt = self.key_store_dao.create(key, "symmetric")

# Create the session
new_session = Session(
    subject_username=subject_username,
    organization_name=organization_name,
    key_id=encrypted_session_key.id,
    key_salt=salt,
    key_iv=iv,
    nonce=nonce,
    counter=counter
)
```

Figure 46 – Create a Session (SessionDAO.py)

```python
class KeyStoreDAO(BaseDAO):

# ------------------------------

    def create(self, key: bytes, type: str) -> tuple[KeyStore, bytes, bytes] | KeyStore:
        """Create a new KeyStore entry."""
        try:
            if type in ["symmetric", "private"]:
                key, iv, salt = self.encrypt_key(key)

            new_key = KeyStore(key=key, type=type)
            self.session.add(new_key)
            self.session.commit()

            return (new_key, iv, salt) if type in ["symmetric", "private"] else new_key
```

Figure 47 – Create a KeyStore entry (KeyStoreDAO.py)

```python
def encrypt_key(self, key: bytes) -> tuple[bytes, bytes, bytes]:
    """
    Encrypt the key using AES256 with a derived key from the repository password.
    """

    aes = AES(AESModes.CBC)

    # Derive AES key from the repository password
    repository_password = os.getenv("REPOSITORY_PASSWORD")
    salt = secrets.token_bytes(16)
    aes_key = aes.derive_aes_key(repository_password, salt)

    encrypted_key, iv = aes.encrypt_data(key, aes_key)

    return (encrypted_key, iv, salt)
```

Figure 48 – Method to encrypt a key using AES256 (KeyStoreDAO.py)

```python
def derive_aes_key(self, password: str, salt: bytes) -> bytes:
    """ Derive a secure AES key from the repository password using PBKDF2.

    Args:
        password (str): Password to derive the key from

    Returns:
        bytes: Derived AES key
    """

    # Use PBKDF2 to derive the AES key
    kdf = PBKDF2HMAC(algorithm=hashes.SHA256(), length=32, salt=salt, iterations=100000)
    return kdf.derive(password.encode())
```

Figure 49 – Method to derive a secure AES key using PBKDF2 (AES.py)

For the generation of this salt, we rely on the ***secrets*** Python library. According to Python's official documentation [3], the secrets module is designed for generating cryptographically strong random numbers, making it suitable for managing sensitive data such as passwords, account authentication, security tokens, and other related secrets.

ASVS Requirement: v4.0.3-6.2.7

Description

Verify that encrypted data is authenticated via signatures, authenticated cipher modes, or HMAC to ensure that ciphertext is not altered by an unauthorized party.

Applicable to Project's Concept: Yes

Why is/isn't it applicable to this project?

This requirement is applicable because the core function of the project is to create a repository that handles client requests to store various types of data (from client information to document files) and to retrieve that data. Ensuring that encrypted data is authenticated is crucial to prevent unauthorized modifications to the original data, both when in transit and when stored by the repository. This guarantees that the data remains intact, preventing potential attacks or tampering during storage or transmission.

Implemented: Partially

Problem

Currently, the repository ensures the security of data exchanged between the client and the server during transmission by employing different mechanisms based on whether there is already a session context or not. Specifically:

- **If a shared secret has been established**, we generate a Message Authentication Code (MAC) for each message using the integrity key and the HMAC function with SHA-256.
- **If no shared secret exists**, we sign the data using the *data.sign(ECC_private_key)* function, using the ECC private keys of both the client and the server.

These mechanisms effectively authenticate the transmitted data, safeguarding it against tampering or unauthorized modifications while in transit.

However, a key limitation exists in the way we handle sensitive data stored within the repository. Currently, the data is encrypted using a key derived from the repository's master key/password. While this ensures that the data is encrypted and protected at rest, **we do not authenticate the encrypted data itself**. In other words, the encrypted data stored in the repository is not signed to verify its integrity or authenticity when accessing it.

Impact

The difference in approach between securing transmitted and stored data introduces a vulnerability in the system. While the data exchanged between the client and server is effectively protected against tampering through authentication mechanisms, the stored data does not receive the same level of protection. This creates a potential risk, as the stored data could be susceptible to unauthorized modifications or tampering while at rest. Without a signature to verify its authenticity, the server cannot reliably determine whether the stored data has been altered, which undermines the integrity of the repository.

Solution

To address this issue, a straightforward solution would be to generate an HMAC for the stored data, similar to the approach already used for interchanged messages. In this case, a key generated from the password of the repository could be used to create the HMAC, and this digest (and respective key salt) would be stored alongside the encrypted data, much like in other cases where the initialization vector (IV) or salt are needed. Whenever access to the stored data is required, the system would first verify its integrity by checking the HMAC. Only after confirming that the data has not been tampered with would decryption proceed. This approach ensures both the confidentiality and authenticity of the stored data.

Evidences

As previously mentioned, the authentication of interchanged data is achieved through signatures or Message Authentication Codes (MACs), depending on the context of the transmission. Below are the key scenarios and the methods employed:

Session Establishment: During the creation of a session, both parties authenticate each other using their ECC keys. This is accomplished by signing the exchanged data, ensuring that the identities of both parties are verified and the session is secure.

```python
def exchange_keys(private_key: ec.EllipticCurvePrivateKey, data: dict, rep_address: str, rep_pub_key):
    """Exchange keys with the repository
    Sends the signed data (which includes the public key of the subject) to the server (endpoint rep_addr/sessions) and receives the public key of the server.

    Args:
        private_key (ec.EllipticCurvePrivateKey): Subject's private key associated with the public key registered by this subject in that organization
        data (dict): Data to be sent to the repository
        rep_address (str): Repository address
        rep_pub_key (bytes): Public key of the repository

    Returns:
        session_key: bytes: Derived key from the ECDH exchange (shared secret / session key)
        response_data: dict: Data received from the repository
    """

    ### HANDSHAKE ###
    ecdh = ECC()

    # Generate random Private Key and obtain Public Key
    _, session_public_key = ecdh.generate_keypair()
    session_public_key_str = convert_bytes_to_str(session_public_key)

    # Create packet made of public key and data
    data = {
        "public_key" : session_public_key_str,
        **data
    }

    # Generate Signature
    signature = sign(
        data = data,
        private_key = private_key
    )

    signature_str = convert_bytes_to_str(signature)

    # Build Session creation packet
    body = {
        "data": data,
        "signature": signature_str
    }

    endpoint = "/sessions"
    # Send to the server
    response, received_message = anonymous_request(rep_pub_key, "post", rep_address, endpoint, body)
```

Figure 50 – Client side - Method exchange_keys() (client_session_utils.py)

```python
def create_session(data, db_session: SQLAlchemySession):      You, 4 weeks ago • some refactor
    if (not verify_signature(data=data, pub_key=client_pub_key)):
        return return_data("error", f"Invalid signature!", HTTP_Code.BAD_REQUEST)

    # Derive session key
    session_key: bytes
    session_server_public_key: bytes
    session_key, session_server_public_key = exchange_keys(client_session_public_key=base64.b64decode(client_session_pub_key))

    ## Create session
    nonce = secrets.token_hex(16)
    try:
        session = session_dao.create(
            username,
            org_name,
            session_key,
            counter = 0,    # for replay attack prevention
            nonce = nonce,  # for unique session identification
        )
    except IntegrityError:
        return return_data("error", f"Session for user '{username}' already exists.", HTTP_Code.BAD_REQUEST)

    # Create response
    result = {
        "session_id": session.id,
        "username": session.subject_username,
        "organization": session.organization_name,
        "roles": [role.name for role in session.session_roles],
        "public_key": base64.b64encode(session_server_public_key).decode('utf-8'), # So that the client can generate the shared
        "nonce": nonce,
    }

    # Sign response
    signature = sign(
        data = str(result),
        private_key = rep_priv_key,
    )

    # Finish response packet
    result = json.dumps({
        "data": result,
        "signature": base64.b64encode(signature).decode('utf-8')
    })

    # Return response to the client
    return result, HTTP_Code.CREATED
```

Figure 51 – Server side - Create Session (session_service.py)

Key Agreement for Encryption Outside a Session Context: When communication occurs outside an established session context, it requires both parties to agree on an ephemeral shared secret, for both encryption and integrity – a one-time-use key. To achieve this, the server transmits its ephemeral public key accompanied by a signature. This signature validates the authenticity of the key and, by extension, the source, ensuring the integrity of the key exchange process and safeguarding against unauthorized interception or tampering.

```python
@organization_blueprint.route("/organizations", methods=["GET", "POST"])
def organizations():
    db_session = g.db_session
    if request.method == 'GET':
        print("SERVER: Getting organizations")
        data = request.json
        if data and "public_key" in data:
            return get_ephemeral_server_public_key(data, db_session)

        encryption_key = get_shared_secret(data["client_ephemeral_public_key"]) # Might not be necessary to always send t
        return_data, code = list_organizations(db_session)
        encrypted_return_data, iv_encrypted_return_data = encrypt_anonymous_content(return_data.encode(), encryption_key)

        return json.dumps({"data": convert_bytes_to_str(encrypted_return_data),
                           "iv": convert_bytes_to_str(iv_encrypted_return_data)
                           }), code
```

Figure 52 – Organization controller: /organizations endpoint (organization_controller.py)

```python
ephemeral_keys = {}

def get_ephemeral_server_public_key(data, db_session):
    client_ephemeral_pub_key = convert_str_to_bytes(data["public_key"])
    result, encryption_key, code = generate_anonymous_signed_shared_secret(client_ephemeral_pub_key, db_session)
    ephemeral_keys[data["public_key"]] = encryption_key
    return result, code

def get_decrypted_request(data, encryption_key):
    ephemeral_keys.pop(data["client_ephemeral_public_key"])
    encrypted_message = convert_str_to_bytes(data["message"])
    iv = convert_str_to_bytes(data["iv"])
    decrypted_data = decrypt_anonymous_content(encrypted_message, encryption_key, iv).decode()

    return json.loads(decrypted_data), encryption_key

def generate_anonymous_signed_shared_secret(client_ephemeral_pub_key: bytes, db_session: Session):
    repository_dao = RepositoryDAO(db_session)
    encryption_key, ephemeral_server_public_key = exchange_keys(client_ephemeral_pub_key)
    data = {
        "public_key": convert_bytes_to_str(ephemeral_server_public_key)
    }

    # Get repository private key using the respective password to decrypt it
    rep_priv_key_password: str = os.getenv('REP_PRIV_KEY_PASSWORD')
    rep_priv_key = ECC.load_private_key(repository_dao.get_private_key(), rep_priv_key_password)

    # Sign response
    signature = sign(
        data = data,
        private_key = rep_priv_key,
    )

    # Finish response packet
    result = json.dumps({
        "data": data,
        "signature": convert_bytes_to_str(signature)
    })

    # Return response to the client
    return result, encryption_key[:32], HTTP_Code.OK
```

Figure 53 – get-ephemeral_server_public_key(), get_decrypted_request(), generate_anonymous_signed_shared_secret() methods (utils.py)

Authenticated Messaging Within a Session Context: Once a session is established and a shared secret is in place, both parties authenticate messages by calculating a MAC for the encrypted data. This is done using the HMAC function with the shared integrity key, ensuring that the data transmitted within the session remains both confidential and tamper-proof.

```python
def encrypt_payload(data: dict | str, encryption_key: bytes, integrity_key: bytes) -> dict[str, dict]:
    """Encrypts the payload to be sent to the client

    Args:
        data (dict | str): Data to be sent
        encryption_key (bytes): first part of session key, used to encrypt data
        integrity_key (bytes): second part of session key, used to encrypt mac

    Returns:
        dict[str, dict]: Encrypted payload
    """

    # Encrypt data
    if isinstance(data, dict):
        data = json.dumps(data)

    encryptor = AES()
    encrypted_data, data_iv = encryptor.encrypt_data(data.encode(), encryption_key)

    data = {
        "message": base64.b64encode(encrypted_data).decode(),    # Data to be sent to the client
        "iv" : base64.b64encode(data_iv).decode(),               # IV used to encrypt the data
    }

    data_to_digest = (data["message"] + data["iv"]).encode()
    digest = calculate_digest(data_to_digest, integrity_key)

    body = {
        "data": data,
        "signature": base64.b64encode(digest).decode('utf-8') # convert_bytes_to_str(digest)
    }

    return body
```

Figure 54 – Method encrypt_payload() (server_session_utils.py)

```python
def calculate_digest(data: bytes, key: bytes) -> bytes:
    """Calculate the digest of the data using SHA256 algorithm

    Args:
        data (bytes): Data to be hashed
        key (bytes): Key to hash the data

    Returns:
        bytes: Hashed data
    """

    digest = hmac.HMAC(key, hashes.SHA256())
    digest.update(data)

    return digest.finalize()
```

Figure 55 – Method to calculate the digest of the data using SHA256 (integrity.py)

ASVS Requirement: v4.0.3-6.2.8

Description

Verify that all cryptographic operations are constant-time, with no 'short-circuit' operations in comparisons, calculations, or returns, to avoid leaking information.

Applicable to Project's Concept: Yes

Why is/isn't it applicable to this project?

This requirement is applicable as encryption is a fundamental piece in the project as almost all client-server messages exchanged are encrypted, and in case of short-circuit operations data could be accidentally leaked or attackers could gather information that gives information such as Oracle Padding Attack

Implemented: Partially

Problem

As mentioned in requirement 6.2.1, the client always sends a digest of the packet to the server, so before we do any decrypt, the digest of the received message is compared to its signature, which means that, if any part of the packet is modified between the client and the server then we don't perform a decrypt, otherwise, a decryption is made and guaranteed to succeed, unless the client performs an invalid encryption.

But there's a problem.

In *load_session* method, when attempting to decrypt the data, the server first checks whether the received session ID doesn't exist, whether the session has expired and whether the subject is suspended, and, if positive, gives the root cause of failure to the client.

This behaviour allows attackers to create a session and send packets with modified session IDs to probe the database. By doing so, they can determine:

- Which session IDs exist in the database;
- Whether users associated to the sessions are suspended;
- Which sessions have expired.

Thereby exposing some information to attackers.

```python
def load_session(data: dict, db_session: SQLAlchemySession, organization_name: str) -> tuple[dict, Session, bytes]:
    session_dao = SessionDAO(db_session)

    # Get session
    session_id = data.get("session_id")
    session = session_dao.get_by_id(session_id)

    if session is None:
        raise ValueError(
                f"Session with id {session_id} not found", HTTP_Code.NOT_FOUND, None
            )

    if subject_invalid(session, db_session):
        raise ValueError(
                f"Subject {session.subject_username} is suspended", HTTP_Code.FORBIDDEN, None
            )

    if session_expired(session):
        raise ValueError(
                f"Session with id {session_id} expired", HTTP_Code.FORBIDDEN, None
            )

    session_dao.update_last_interaction(session.id)
    session_key = session_dao.get_decrypted_key(session_id)

    decrypted_data = decrypt_payload(data, session_key[:32], session_key[32:])
```

Figure 56 – Method load_session() (server_session_utils.py)

When creating a session, information could also be exposed. Attackers could attempt to create sessions with other usernames and organizations to determine if a user exists, if they belong to a specific organization, and whether they are suspended.

This vulnerability arises because the server discloses this information before validating the provided signature against the saved public key. Only after confirming the existence of the user and their organization, the server verifies if the sent signature is valid according to the saved public key.

```python
def create_session(data, db_session: SQLAlchemySession):
    # Read client data from request and load it
    msg_data = data.get("data")
    client_session_pub_key = msg_data.get("public_key")
    org_name = msg_data.get('organization')
    username = msg_data.get('username')

    # Verify if there is no active session for the user in the organization
    last_session_user_org = session_dao.get_last_session_of_user_in_org(username, org_name)
    if last_session_user_org is not None:
        if not session_expired(last_session_user_org):
            return json.dumps({"error": f"Session for user '{username}' in organization '{org_name}' already exists."}), HTTP_Code.BAD_REQUEST

    try:
        client = organization_dao.get_org_subj_association(
            org_name=org_name,
            username=username
        )
        if client.status == Status.SUSPENDED.value:
            return json.dumps({"error": f"User '{username}' is suspended."}), HTTP_Code.FORBIDDEN
    except ValueError as e:
        message = e.args[0]
        return return_data("error", message, HTTP_Code.NOT_FOUND)

    if (client is None):
        return return_data("error", f"User not found!", HTTP_Code.NOT_FOUND)

    # Get client public key
    client_pub_key = keystore_dao.get_by_id(client.pub_key_id).key

    # Verify Signature
    if (not verify_signature(data=data, pub_key=client_pub_key)):
        return return_data("error", f"Invalid signature!", HTTP_Code.BAD_REQUEST)
```

Figure 57 – Create Session (session_service.py)

Impact

An attacker could use the first problem to identify active sessions and, if a session belongs to a suspended user, reveal the username associated with that session.

The second problem is more severe, as it allows an attacker to discover usernames of repository users, identify the organizations they are linked to, whether they are suspended in those organizations and if they have an active session to that organization. Therefore, being able to map some relations of the database.

When combined, these vulnerabilities could be used to efficiently compile a list of repository users and map their associated organizations, using the first exploit to identify active sessions and the second to reveal organizational affiliations.

Solution

To fix the first vulnerability, the three error messages could be merged into one that doesn't specify the root cause of the error, only that the session isn't valid. Another solution to this problem could be implementing a cookie-based session file that would be sent to the server to verify that session file hadn't been tampered with.

To fix the second vulnerability, the server should first verify the identity of the client before leaking sensitive information, and if the identity could not be checked, a general error message should be sent to the client.

V6.3 - Random Values

ASVS Requirement: v4.0.3-6.3.1

Description

Verify that all random numbers, random file names, random GUIDs, and random strings are generated using the cryptographic module's approved cryptographically secure random number generator when these random values are intended to be not guessable by an attacker.

Applicable to Project's Concept: Yes

Why is/isn't it applicable to this project?

This requirement is applicable to the project because the repository generates and relies on random values for critical security operations. Examples include generating initialization vectors (IVs) for encryption, salts for key derivation, and ephemeral ECC keys for session establishment. Ensuring these random values are generated using a cryptographically secure random number generator is vital to prevent attackers from predicting or guessing these values, which could compromise the overall security of the system.

Implemented: Yes

Evidences

In this project, various random values are generated to ensure security, including Initialization Vectors (IVs), salts, ECC private keys (from which corresponding public keys are derived), and ECC shared secrets.

For IVs and salts, randomness is ensured using Python's ***os.urandom*** and ***secrets.token_bytes*** modules, both of which are cryptographically secure random number generators. Evidence of their usage and cryptographic security has been detailed earlier in this report.

For ECC ephemeral private keys and shared secret agreements, the ***cryptography*** Python module is employed. This module [3] adheres to the *NIST* standard ***Special Publication 800-90A Revision 1*** (which is explained on "*Recommendation for Random Number Generation Using Deterministic Random Bit Generators*" [12]). The Deterministic Random Bit Generator (DRBG) algorithm used by this module (see reference [7], chapter 8) is explicitly recommended by *NIST* for generating cryptographically secure random numbers and symmetric keys. Below are specific examples where this module is applied.

Generation of the Ephemeral ECC Private Key: This key is generated solely for the purpose of secret agreement and is discarded after use. Once the ephemeral private key is created, the corresponding public key is derived from it.

```python
class ECC:
    def __init__(self, curve=ec.SECP521R1()):
        self.curve = curve

# ------------------------------ ECC & ECDH methods ------------------------------ #

    def generate_keypair(self, password: str = None) -> tuple[bytes, bytes]:
        """ Generates an ECC keypair using the specified curve and password

        Args:
            password (str): password to protect the private key (if None, no password is used)

        Returns:
            tuple[str, str]: tuple containing the private and public keys in PEM format (serialized)
        """

        self.private_key = ec.generate_private_key(self.curve)
        public_key = self.private_key.public_key()
```

Figure 58 – Class ECC: generate_keypair() method (ECC.py)

Generation of the Shared Secret: The shared secret is computed using the peer's ephemeral ECC public key and the exchange function provided by the *cryptography* module. Following the best practices outlined in the module's documentation [4], the shared secret is immediately passed through a key derivation function (KDF). This step ensures the generated key always have a standardized size (32-bytes for anonymous shared secret and 64-bytes for sessions shared secrets) as well as any inherent structure in the shared secret is destroyed, enhancing security.

This approach aligns with established cryptographic standards, ensuring that the generated keys are secure and resistant to potential attacks.

```python
def generate_shared_secret(self, peer_public_key: bytes):
    """ Generates a shared secret using the provided peer public key

    Args:
        peer_public_key (bytes): public key of the peer to generate the shared secret with

    Returns:
        bytes: shared secret

    """

    if self.private_key is None:
        raise Exception("No private key has been generated yet!")

    peer_public_key_ecc = serialization.load_pem_public_key(peer_public_key)

    self.shared_key = self.private_key.exchange(ec.ECDH(), peer_public_key_ecc)

    self.derived_key = HKDF(
        algorithm=algorithm,
        length=64, # So then I can use 32 bytes for the encryption key and 32 bytes for the MAC (integrity) key
        salt=None,
        info=b''
    ).derive(self.shared_key)

    return self.derived_key
```

Figure 59 – Method generate_shared_secret() (ECC.py)

ASVS Requirement: v4.0.3-6.3.2

Description

Verify that random GUIDs are created using the GUID v4 algorithm, and a Cryptographically-secure Pseudo-random Number Generator (CSPRNG). GUIDs created using other pseudo-random number generators may be predictable.

Applicable to Project's Concept: Yes

Why is/isn't it applicable to this project?

The repository manages sensitive data and a good solution would be to employ unique identifiers for various purposes, such as identifying stored objects or sessions. Ensuring that these identifiers are unpredictable is critical to prevent potential attacks, such as enumeration or guessing of identifiers. Therefore, using random GUIDs generated with a secure algorithm like GUID v4, in combination with a Cryptographically-Secure Pseudo-Random Number Generator (CSPRNG), is essential to maintaining the integrity and security of the system.

Implemented: No

Problem

Currently, our implementation does not consider the use of random GUIDs for object identification. Instead, IDs for database objects are assigned using an auto-incrementing mechanism (provided by the *SQLAlchemy ORM* in use). While this approach may suffice for certain objects, it introduces a significant vulnerability for sensitive objects like Sessions.

With auto-incremented IDs, an attacker attempting to access a valid session ID – even though not being possible to access the session data itself – could potentially infer information about the organization or the subject of the session through error messages or failed access attempts. For instance, they might determine:

- Whether the subject of a session is suspended in the organization based on the system's response, which would also indicate that the subject with that username exists on the organization.
- Whether a session is expired or active, inferred from error messages, also indicating that a session with that ID existed/exists.

Below are the possible error messages that an attacker could obtain with a valid session ID:

```python
if subject_invalid(session, db_session):
    raise ValueError(
            f"Subject {session.subject_username} is suspended", HTTP_Code.FORBIDDEN, None
        )

if session_expired(session):
    raise ValueError(
            f"Session with id {session_id} expired", HTTP_Code.FORBIDDEN, None
        )
```

Figure 60 – Session error messages (server_session_utils.py)

These types of inferences, made possible by predictable auto-incremented IDs, could expose sensitive organizational or user-related information, thereby compromising security and privacy.

Impact

The use of predictable IDs, such as auto-incremented integers, for sessions creates a vulnerability that can be exploited by attackers to gather unauthorized information. This compromises the confidentiality and integrity of the system, as it allows attackers to infer sensitive details about users and their interactions with the organization. Such vulnerabilities could lead to privacy breaches, unauthorized access, and reputational damage.

Solution

To address this issue, the implementation should change slightly from how it currently stands, by simply replace auto-incremented IDs with random GUIDs generated using the GUID v4 algorithm and a Cryptographically-Secure Pseudo-Random Number Generator (CSPRNG). Specifically for sessions, the "*autoincrement*" option would be removed from the *id* column:

```python
# ------------------------------------------------------------------------------------------ #

You, 7 days ago | 1 author (You)
class Session(Base):
    __tablename__ = 'session'

    id: Mapped[int] = mapped_column(primary_key=True, autoincrement=True)
    subject_username: Mapped[str] = mapped_column(ForeignKey('subject.username'), nullable=False)
    organization_name: Mapped[str] = mapped_column(ForeignKey('organization.name'), nullable=False)
    key_id: Mapped[int] = mapped_column(ForeignKey('key_store.id'), nullable=False)  # Foreign key column
    key_salt: Mapped[bytes] = mapped_column(nullable=False)
    key_iv: Mapped[bytes] = mapped_column(nullable=False)

    nonce: Mapped[str] = mapped_column(nullable=True)
    counter: Mapped[int] = mapped_column(nullable=True)

    last_interaction: Mapped[datetime] = mapped_column(nullable=False, default=lambda: datetime.now())

    # Relationships
    subject: Mapped["Subject"] = relationship()
    organization: Mapped["Organization"] = relationship()
    session_roles: Mapped[list["Role"]] = relationship(secondary=SessionRoles)
    key: Mapped["KeyStore"] = relationship()  # Relationship to KeyStore

    def __repr__(self):
        return f"<Session(id={self.id}, subject_username={self.subject_username}, organization_name={self.organization_name}, key_id={self.key_id})>"

# ------------------------------------------------------------------------------------------ #
```

Figure 61 – Class Session (database_orm.py)

When creating a new Session, as it is represented by the following figure, a random GUID would be generated and assigned to the *id*:

```
## Create session
nonce = secrets.token_hex(16)
try:
    session = session_dao.create(
        username,
        org_name,
        session_key,
        counter = 0,    # for replay attack prevention
        nonce = nonce,  # for unique session identification
    )
except IntegrityError:
    return return_data("error", f"Session for user '{username}' already exists.", HTTP_Code.BAD_REQUEST)
```

Figure 62 – Exception handling creating a session (session_service.py)

This would ensure that session IDs are unpredictable and unique, effectively mitigating the risk of enumeration or inference attacks.

ASVS Requirement: v4.0.3-6.3.3

Description

Verify that random numbers are created with proper entropy even when the application is under heavy load, or that the application degrades gracefully in such circumstances.

Applicable to Project's Concept: Yes

Why is/isn't it applicable to this project?

This requirement is applicable because the project involves cryptographic operations, and ensuring that the application is always cryptographically secure and in all circumstances is mandatory, otherwise it could be attacked in that manner

Implemented: Yes

Evidences

As mentioned in requirement *6.2.3*, *IVs* are generated using *os.urandom* and, as mentioned in requirement *6.2.6*, *salts* are generated using Python's library ***secrets*** which, according to *Python Documentation*, uses *os.urandom* under the hood.

Python Documentation [3] states that *os.urandom* reads from */dev/urandom*, which, according to ***urandom manual*** [10], provides high-entropy data, and although entropy levels might degrade under heavy load, no knowledge on how to predict the values is available in current non-classified literature. As such, */dev/urandom* can be considered secure, even under significant load.

V6.4 – Secret Management

ASVS Requirement: v4.0.3-6.4.1

Description

Verify that a secrets management solution such as a key vault is used to securely create, store, control access to and destroy secrets.

Applicable to Project's Concept: Yes

Why is/isn't it applicable to this project?

This requirement is applicable to the project because the repository handles sensitive data, including encryption keys (e.g. for files encryption data) and other secrets (e.g. session ones), that need to be securely managed. Ensuring proper creation, storage, controlled access, and eventual destruction of these secrets is critical to maintaining the overall security of the system. Without a robust secrets management solution, the risk of unauthorized access or exposure of sensitive information increases, potentially compromising the confidentiality and integrity of the stored data.

Implemented: Partially

Problem

Currently, the repository has a dedicated table for storing cryptographic keys, including public, private, and symmetric keys. The implemented security measures include:

- **Public Keys**: Stored unencrypted, as they are not sensitive by nature.
- **Private and Symmetric Keys**: Stored encrypted. The encryption key for this data is derived from the repository password stored in an environment variable (*.env* file). The key derivation process uses *PBKDF2HMAC* with *SHA-256*, a unique salt for each key, and number of iterations.

Session keys are symmetric keys associated with sessions. These sessions have a limited lifetime, which implicitly means that also the related session keys have limited validity.

Files encryption metadata are also stored encrypted.

For each piece of data that is encrypted to the database, a different salt is used to always derive different encryption keys from the repository password.

These measures align with the ASVS requirement by encrypting sensitive keys, limiting the scope of session secrets, having a different key for each encryption and having a separate place for storing the repository password, where only the repository knows where to find it.

However, the current implementation has a critical flaw: the repository password used for encrypting keys is not protected by a dedicated secrets management solution. If this password is compromised, all encrypted keys in the database become vulnerable to decryption.

Additionally, although a different encryption key is used for each case, there is no mechanism for password rotation itself and re-encryption of stored keys, which would add more robustness since the way that sensitive data is encrypted would be always changing.

Another problem is, even though session keys are associated with sessions that have a limited lifetime, those keys and others related to other contexts are created, stored but never deleted from the database.

Impact

The current implementation introduces the following main risks:

1. **Password Exposure**: If the *.env* file is compromised, an attacker can access the repository password and decrypt all stored keys. This could lead to unauthorized access to sensitive data or operations.
2. **Lack of Password Rotation**: Without periodic rotation of the repository password, the system remains vulnerable if the password is exposed over time.

These vulnerabilities could undermine the confidentiality and integrity of sensitive data managed by the repository, potentially leading to data breaches or unauthorized operations.

Solution

A simple and effective solution to meet the ASVS requirement is to implement:

1. **Password Management Tool**: Instead of storing the password of the repository on an environment variable, we could use something presented in the practical classes: a secure *password management tool*. For scalability, for example, we could integrate a tool like *Bitwarden* to securely store and manage the repository password. Bitwarden's *CLI* or *API* provides ease on the access to passwords, while maintaining strong security practices. Alternatively, for a simpler solution, we could use *KeePass* to store sensitive data locally in an encrypted database.

2. **Key Rotation**: Periodically we could decrypt all the encrypted data, rotate the repository password and re-encrypt all stored keys with the new password. This would minimize the impact of long-term exposure.

By adopting these measures, the repository can securely manage secrets and keys while addressing the identified vulnerabilities, ensuring compliance with the ASVS requirement.

Evidences

Evidence on how we encrypt the different types of keys was provided earlier in the report.

Regarding sessions having limited lifetime, here we could revisit the session class, where alongside the session key information, there is also a last interaction attribute, which is updated at every session interaction. Also, at every request, on the *load session* function, the session status is verified in order to determine if it has expired or not. The lifetime value is stored on an environment variable. The following are the code snippets:

```python
class Session(Base):
    __tablename__ = 'session'

    id: Mapped[int] = mapped_column(primary_key=True, autoincrement=True)
    subject_username: Mapped[str] = mapped_column(ForeignKey('subject.username'), nullable=False)
    organization_name: Mapped[str] = mapped_column(ForeignKey('organization.name'), nullable=False)
    key_id: Mapped[int] = mapped_column(ForeignKey('key_store.id'), nullable=False)  # Foreign key column
    key_salt: Mapped[bytes] = mapped_column(nullable=False)
    key_iv: Mapped[bytes] = mapped_column(nullable=False)

    nonce: Mapped[str] = mapped_column(nullable=True)
    counter: Mapped[int] = mapped_column(nullable=True)

    last_interaction: Mapped[datetime] = mapped_column(nullable=False, default=lambda: datetime.now())

    # Relationships
    subject: Mapped["Subject"] = relationship()
    organization: Mapped["Organization"] = relationship()
    session_roles: Mapped[list["Role"]] = relationship(secondary=SessionRoles)
    key: Mapped["KeyStore"] = relationship()  # Relationship to KeyStore

    def __repr__(self):
        return f"<Session(id={self.id}, subject_username={self.subject_username}, organization_name={self.organization_name}, key_id={self.key_id})>"
```

Figure 63 – Session Class. Has a key_id and last_interaction timestamp (database_orm.py)

```python
def load_session(data: dict, db_session: SQLAlchemySession, organization_name: str) -> tuple[dict, Session, bytes]:
    Args:
        data (dict): Data received from the client
        session_dao (SessionDAO): DAO to access the session
        organization_name (str): Name of the organization

    Returns:
        tuple[dict, Session, bytes]: Decrypted data, Session, Session key
    """

    session_dao = SessionDAO(db_session)

    # Get session
    session_id = data.get("session_id")
    session = session_dao.get_by_id(session_id)

    if session is None:
        raise ValueError(
                f"Session with id {session_id} not found", HTTP_Code.NOT_FOUND, None
            )

    if subject_invalid(session, db_session):
        raise ValueError(
                f"Subject {session.subject_username} is suspended", HTTP_Code.FORBIDDEN, None
            )

    if session_expired(session):
        raise ValueError(
                f"Session with id {session_id} expired", HTTP_Code.FORBIDDEN, None
            )
```

Figure 67 – Exception handling loading a session: session_expired (server_session_utils.py)

```python
SESSION_LIFETIME = eval(os.getenv("SESSION_LIFETIME"))

def session_expired(session: Session) -> bool:
    session_last_interaction = session.last_interaction
    current_time = datetime.datetime.now()
    return (current_time - session_last_interaction).total_seconds() > SESSION_LIFETIME
```

Figure 68 – Method to check if a session has expired (server_session_utils.py)

```
# Session Lifetime
SESSION_LIFETIME = 60*10 # 10 minutes
```

Figure 69 – Session Time To Live: 10 minutes (.env)

Regarding files metadata being stored encrypted, evidences were shown previously on this report (specifically, on requirement *6.1.1*).

Regarding the retrieval of the repository password from the *.env* file, these are the main snippets where it is implemented (secondary ones were already shown in previous requirements):

```python
def encrypt_key(self, key: bytes) -> tuple[bytes, bytes, bytes]:
    """
    Encrypt the key using AES256 with a derived key from the repository password.
    """

    aes = AES(AESModes.CBC)

    # Derive AES key from the repository password
    repository_password = os.getenv("REPOSITORY_PASSWORD")
    salt = secrets.token_bytes(16)
    aes_key = aes.derive_aes_key(repository_password, salt)

    encrypted_key, iv = aes.encrypt_data(key, aes_key)

    return (encrypted_key, iv, salt)
```

Figure 70 – Method to encrypt a key using AES256 (KeyStoreDAO.py)

```python
def derive_aes_key(self, password: str, salt: bytes) -> bytes:
    """ Derive a secure AES key from the repository password using PBKDF2.

    Args:
        password (str): Password to derive the key from

    Returns:
        bytes: Derived AES key
    """

    # Use PBKDF2 to derive the AES key
    kdf = PBKDF2HMAC(algorithm=hashes.SHA256(), length=32, salt=salt, iterations=100000)
    return kdf.derive(password.encode())
```

Figure 71 – Method to derive a secure AES key using PBKDF2 (AES.py)

```
# ------------------------- Repository Variables ------------------------- #

# Password to derive a master key for storing the private data encryted
REPOSITORY_PASSWORD = 'amao8!ms-aosaç02m#-gdkcbz'
```

Figure 72 – Repository password (.env)

ASVS Requirement: v4.0.3-6.4.2

Description

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

Applicable to Project's Concept: Yes

Why is/isn't it applicable to this project?

The repository relies on its password to encrypt and decrypt sensitive stored data. As such, this password must become accessible without being exposed. Implementing a security module for cryptographic operations ensures that passwords are not directly accessible to the application, thereby enhancing the overall security posture and protecting sensitive data from potential breaches.

Implemented: No

Problem

The problem with this approach is that the application's KeyStore DAO handles the repository password directly by reading it from the environment variables and performing decryption itself. This violates the principle of isolating key material and cryptographic operations from the application.

Impact

In the event the application or its environment is compromised, an attacker could potentially access the password and the decryption logic, leading to a significant security risk as he would have all the necessary info to decrypt all database encrypted data, including sensitive information, such as session keys and file encryption keys.

Solution

This issue could be fixed by implementing, in the repository, a module specialized in:

1. Gathering the repository password from a secure key vault
2. Encrypting
    a. Deriving an encryption key from the repository password with a random salt
    b. Encrypting the data
    c. Storing the encrypted data and the encryption context (such as salt and IV or nonce) in the database
3. Decrypting
    a. Gathering encrypted data and encryption context from the database
    b. Deriving the same key used to encrypt by using the same salt (provided in the encryption context)
    c. Decrypting the data

Conclusion

In this project, we implemented a comprehensive system for managing organizations, roles, subjects, and document repositories, ensuring robust security measures were in place to protect sensitive data and maintain the integrity of operations. Through careful design and implementation, we addressed key aspects such as role-based access control, session management, and cryptographic techniques to guarantee confidentiality, authentication, and data integrity.

The inclusion of the Application Security Verification Standard (ASVS) further strengthened our approach by providing a structured framework to test and validate the security of the system. This allowed us to identify potential vulnerabilities and learn more about best practices in secure software development.

Throughout this course, we gained valuable knowledge about security principles and their practical applications. We learned how to design and implement secure systems by integrating encryption, authentication protocols, and access control mechanisms. Additionally, the importance of security testing, especially with standards like ASVS, has become evident as a critical step in developing resilient applications.

This project was a significant learning experience that combined theoretical concepts with hands-on implementation. It reinforced the importance of security in software development and provided us with the tools and methodologies to approach similar challenges in future endeavors.

References

1. OpenSSL FIPS 140-2 Security Policy. (2024). https://csrc.nist.gov/CSRC/media/projects/cryptographic-module-validation-program/documents/security-policies/140sp4282.pdf

2. OWASP Cheat Sheet Series. (2024). Cryptographic Storage Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html

3. Python Documentation. https://docs.python.org/3/

4. pyca/cryptography. (2024). https://cryptography.io/en/latest/

5. Flask. (2024). https://flask.palletsprojects.com/en/stable/

6. SQLAlchemy. (2024). https://www.sqlalchemy.org/

7. Sohl, E. (2021). Cryptopals: Exploiting CBC Padding Oracles. NCC Group. https://www.nccgroup.com/us/research-blog/cryptopals-exploiting-cbc-padding-oracles/

8. OWASP. (2024). Application Security Verification Standard. https://owasp.org/www-project-developer-guide/draft/requirements/application_security_verification_standard/

9. Python | os.urandom() Method. (2019). GeeksforGeeks. https://www.geeksforgeeks.org/python-os-urandom-method/

10. urandom Manual. https://man.urandom.at/

11. Generate Secure Random Numbers for Managing Secrets. (2024). Python Documentation. https://docs.python.org/3/library/secrets.html

12. Barker, E., & Kelsey, J. (2015). *Recommendation for Random Number Generation Using Deterministic Random Bit Generators (SP 800-90Ar1)*. NIST. https://nvlpubs.nist.gov/nistpubs/specialpublications/nist.sp.800-90ar1.pdf

13. Elliptic Curve Cryptography. (2024). pyca/cryptography. https://cryptography.io/en/latest/hazmat/primitives/asymmetric/ec/#cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePrivateKey.exchange