Report and ASVS analysis of the Repository Service

Project developed under the “Segurança Informática e Nas Organizações” class at University of Aveiro

30/12/2024

Guilherme Rosa P2 - 113968

Rui Machado P2 - 113765

Martim Santos P2 - 114614

Table of Content

Report

Introduction

This report describes the development and the functionalities of a secure repository service. This software consists of a secure storage system that allows clients to upload, download and control access to documents within an organization, all while keeping privacy and integrity.

In this report the group describes:

- Development strategy;
- Implemented features - Walkthrough;

General development strategy

For the development of the service the group started by implementing the basic features of each entity. Starting from the most basic to the most complex (Subject to Session).

With the basis for the project implemented the group started implementing the APIs starting with the anonymous, then the authenticated and finally the authorized.

Project Structure

The project has two main components - client and server - The code for each of these parts is in the respective folders.

```
├── client
│   ├── document
│   ├── organization
│   ├── role
│   ├── session
│   └── subject
├── repository
│   ├── controller
│   ├── db
│   └── handlers
└── utils
```

Repository

The repository has the server's code and is subdivided into: controllers, db and handlers.

The `controller` folder has controllers for each of the project's entitites. Some of the endpoints have an entity's name but are in a different controller since the concrete operation refers to another entity. Ex: add_doc calls consumes an endpoint in the organization controller. Some endpoints also have separate handlers for better code readability.

The `db` folder has some functions for the handling of the storage and manipulation of sessions, organizations, documents, etc.

Client

In the client there is a directory for each entity and the commands are distributed among the respective directories where they logically belong.

Shared code

There is also a `utils` directory that contains utility functions for both the server and client encrypt/decrypt messages, load/store keys and other common operations.

Database

At the database level, we are using some JSON files, which are located on the server side. We have three files: organizations.json, sessions.json, and files.json.

organizations.json

```
{
    "organizations": [
        {
            "name": "tesla",
            "subject": [

            ],
            "documents": [

            ],
            "acl": [

            ]
        }
    ]
}
```

As we can see, an organization has a list of subjects, a list of documents, and an ACL (Access Control List).

subject

```
{
    "username": "elon_musk",
    "name": "Elon Musk",
    "email": "elon@tesla.com",
    "public_key": "LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUZrd0V3WUhLb1pJem...",
    "active": true,
    "roles": [
        "manager"
    ]
}
```

A subject in an organization is represented by:

- *username*: a unique string that represents a fictitious name, usually related to their real name;
- *name*: a string that represents their real name;
- *email*: a unique string;
- *public_key*: the subject's public key;
- *active*: a field that indicates whether the subject is active in the organization or not;
- *roles*: a list of roles that the subject can assume in the session.

document

```
"documents": [
    {
        "public-metadata": {
        "document_handle": "a569c070-268b-46...",
        "document_name": "Frankenstein",
        "create_date": "2025-01-25",
        "creator": "user2",
        "file_handle": "74d701a8bec280786ce691bc6193cc521c...",
        "deleter": "none",
        "acl": [
                    {
                        "name": "manager",
                        "permissions": [
```

```
                            "DOC_READ",
                            "DOC_ACL",
                            "DOC_DELETE"
                        ]
                    },
                    {
                        "name": "role2",
                        "permissions": [
                            "DOC_ACL",
                            "DOC_DELETE",
                            "DOC_READ"
                        ]
                    }
                ]
                },
        "private-metadata": {
            "alg": "AES-CBC",
            "key": "Eyod1akoGlUkbkaf7akvmt0kRrAUgvkBUEqn39m4MTU="
        },
    },
```

A document in an organization is represented by:

- **public_metadata**: a set of parameters that are public about the document:
  - **document_handle**: a unique identifier for a document. To ensure uniqueness, we use a Python function that generates completely random numbers with a very low probability of collision. `str(uuid.uuid4())`
  - **document_name**: a string representing the document's name.
  - **create_date**: represents the creation date of the document.
  - **creator**: a string representing the username of the person who created the document.
  - **file_handle**: a unique identifier for the file associated with the document. To ensure data integrity, this is a digest of the files' content.
  - **deleter**: completely deleting a document can be a complex and resource-intensive task. For this reason, we use what is known as a "tombstone." If this field is `None`, it means the document has not been deleted. If it contains the username of someone, it means the document was deleted by that person.
  - **acl**: restrict the acess and manipulation of the document. A list of roles.
- **private_metadata**: a set of confidential parameters about the document:
  - **alg**: the encryption algorithm used to encrypt the file.
  - **key**: the key used to encrypt the file.

acl

```
"acl": [
    {
        "name": "manager",
        "permissions": [
            "DOC_ACL",
            "DOC_READ",
            "DOC_NEW",
            "DOC_DELETE",
            "ROLE_ACL",
            "ROLE_NEW",
            "ROLE_DOWN",
            "ROLE_UP",
            "ROLE_MOD",
            "SUBJECT_DOWN",
```

```
            "SUBJECT_UP",
            "SUBJECT_NEW"
        ],
        "subjects": [
            "elon_musk"
        ],
        "status": "active"
    }
]
```

The ACL is a list of roles that can be assumed by subjects within the organization. When an organization is created, the subject who creates it is automatically allowed to assume the role of "manager," which is the organization's default role and contains all existing permissions. This role cannot be suspended, nor can the subject who holds this role, as in certain circumstances, the organization could become inoperable.

role

```
{
        "name": "manager",
        "permissions": [
            "DOC_ACL",
            "DOC_READ",
            "DOC_NEW",
            "DOC_DELETE",
            "ROLE_ACL",
            "ROLE_NEW",
            "ROLE_DOWN",
            "ROLE_UP",
            "ROLE_MOD",
            "SUBJECT_DOWN",
            "SUBJECT_UP",
            "SUBJECT_NEW"
        ],
        "subjects": [
            "elon_musk"
        ],
        "status": "active"
    }
```

A role in an organization is represented by:

- **name**: a unique string.
- **permissions**: a list of permissions granted when the role is assumed.
- **subjects**: the subjects who are allowed to assume the role.
- **status**: indicates whether the role can or cannot be assumed.

Roles are added using the *rep_add_role* command, which takes the *session file* and the *name* of the role to be added as arguments. To add a role, the subject must have a role with the *ROLE_NEW* permission. Once the role is added, any subject with a role that has the *ROLE_MOD* permission can assign permissions to it. Additionally, we have a command that adds a specific role to the list of roles that an authenticated subject can assume.

In general, we have adopted a dynamic strategy for managing roles and the ACL, considering that when an organization is created, it initially only has a default role within the ACL.

files.json

```
{
    "files": {
        "e4bb27c5516063484281c45a6cc...": {
            "file_content":"ft3lsVJqZ2hpry46sc7XibCNTK9Na9zuebdazBjLbLa..."
        },
        "921e318712937af355dc2b....": {
            "file_content": "5b6u0/Xy91oBJO1Ztj+/jpQToKAvJUXV3IhW..."
        },
        "492f3f38d6b5d3ca859514e250e25...": {
            "file_content": "cmX4/7XLDJ5oOPxVItof9249ktB1PSDUei98IhBg32g=" },
        },
    }
}
```

The **files.json** file is where we store the contents of the files. It can be described as our file manager. Each file stored here is represented by its **file_handle**, which, as explained earlier, is derived from the content of the file. In addition to the **file_handle**, we store the file's content, which is already encrypted.

We chose to use only one encryption algorithm, specifically **AES-CBC** (Advanced Encryption Standard in Cipher Block Chaining mode). The reason for this choice is that it is better than simpler modes like ECB because it prevents pattern leakage by XORing each plaintext block with the previous ciphertext block before encryption. This ensures that identical plaintext blocks produce different ciphertexts. Additionally, CBC uses a random Initialization Vector (IV) for the first block, adding randomness and protecting against replay attacks. These features make CBC more secure, especially for encrypting structured or repetitive data.

sessions.json

```
 {
    "sessions": {
        "221f7c99-07f0-4b21-bdcf-d8c2a143344f": {
            "shared_key": "ez/egchLIiuTJ0cGEeI8AZe2LW3qvh+fXh3kN5UVfOA=",
            "life_span": 1737834405.8368752,
            "org": "org1",
            "roles": [
                "manager"
            ]
        },
        "5a2d8b2f-78aa-485d-9881-995762e552e4": {
            "shared_key": "8L2X9LWEahk4omaCROHvlMh/1zTPliislNDbU7jf20w=",
            "life_span": 1737834406.5282009,
            "org": "org1"
        },
    }
}
```

The **sessions.json** file is where we store everything related to the subjects' sessions. A user can have more than one "active" session, but each session refers to one and only one user. For each session, the following data is stored:

- **shared_key**: the key shared between the client and the server.
- **life_span**: the session's lifespan.
- **organization**: the organization associated with the session.
- **roles**: the roles of the user associated with the session.

Building/Running the project

- The server runs on a docker virtual machine that contains flask and can be initiated with `docker-compose up`.

- The client has external dependencies that need installing. The recommended way is creating a virtual environment and adding the `src` directory to the python path.

API documentation

Entity	Prefix	Endpoint	Method	Description	Requires Authentication	Requires Authorization
Document	`/doc`	`/delete_doc`	`POST`	Deletes a document from an organization.	✅	✅
Document	`/doc`	`/file/<file_handle>`	`GET`	Retrieves the content of a file by its handle.		
Document	`/doc`	`/metadata`	`GET`	Retrieves metadata for a specific document in an organization.		
Document	`/doc`	`/update/acl`	`PUT`	Updates Access Control List (ACL) for a document.	✅	✅
Organization	`/org`	`/`	`GET`	Lists all organizations.		
Organization	`/org`	`/<organization>`	`GET`	Retrieves documents belonging to a specific organization.	✅	
Organization	`/org`	`/<organization>/add_doc`	`POST`	Adds a document to a specific organization.	✅	✅
Organization	`/org`	`/`	`POST`	Creates a new organization.	✅	
Organization	`/org`	`/subjects`	`GET`	Lists all subjects in an organization.	✅	
Organization	`/org`	`/<organization>/roles`	`GET`	Lists all roles in a specific organization.	✅	
Role	`/role`	`/create`	`POST`	Creates a new role in an organization.	✅	✅
Role	`/role`	`/subjects`	`GET`	Lists all subjects assigned to a specific role in an organization.	✅	
Role	`/role`	`/permissions`	`GET`	Lists all permissions assigned to a specific role in an organization.	✅	

Role	`/role`	`/permission-roles`	`GET`	Lists all roles that have a specific permission in an organization.	✅	
Role	`/role`	`/suspend`	`POST`	Suspends a specific role in an organization.	✅	✅
Role	`/role`	`/reactivate`	`POST`	Reactivates a specific role in an organization.	✅	✅
Role	`/role`	`/add-permission`	`POST`	Adds a permission to a specific role in an organization.	✅	✅
Role	`/role`	`/remove-permission`	`POST`	Removes a permission from a specific role in an organization.	✅	✅
Session	`/session`	`/list`	`GET`	Lists all sessions.	✅	✅
Session	`/session`	`/{id}`	`GET`	Retrieves session details by ID.	✅	
Session	`/session`	`/delete/{id}`	`DELETE`	Deletes a session by ID.	✅	✅
Session	`/session`	`/validate`	`POST`	Validates a signature for a session.		
Session	`/session`	`/create`	`POST`	Creates a new session with shared key.		
Session	`/session`	`/add-role`	`POST`	Adds a role to the current session.	✅	✅
Session	`/session`	`/remove-role`	`POST`	Removes a role from the current session.	✅	✅
Subject	`/subject`	`/create`	`POST`	Creates a new subject in an organization.	✅	✅
Subject	`/subject`	`/list`	`GET`	Lists all subjects in a specific organization.	✅	
Subject	`/subject`	`/suspend`	`POST`	Suspends a specific subject in an organization.	✅	✅
Subject	`/subject`	`/activate`	`POST`	Reactivates a specific subject in an organization.	✅	✅

Subject	`/subject`	`/add-role`	`POST`	Adds a role to a specific subject in an organization.	✅	✅
Subject	`/subject`	`/remove-role`	`POST`	Removes a role from a specific subject in an organization.	✅	✅
Subject	`/subject`	`/roles`	`GET`	Lists all roles assigned to a specific subject in an organization.	✅	

Organization

Organizations maintain a list of documents and associated subjects. They can be listed without requiring authentication or authorization. Each of these have its unique identifier `name` and the fields `subject`, `documents` and `acl`.

Commands

- Create an Organization
- List organizations
- List subjects (of my organization)

Security mechanisms for organizations

- **ACL Management:** Define the roles that exist in the organization. It specifies which roles are allowed to access to specific resources, operations, or actions.

Subject

Subjects are users or applications that associated to one or more organizations. Each one has a unique identifier (username), along with associated metadata like name, email, and cryptographic keys.

Commands

- Generate subject credentials
- Add subject to organization
- Activate/Suspend subject

Role

Subjects can be assigned to roles within an organization and within a session. Roles define a set of permissions that subjects can possess within an organization.

Commands

- Add role
- List roles
- Suspend/Reactivate role
- Assume/Drop role
- Add/Remove permission
- List roles subjects/permissions

- List permission roles
- List role permissions

Security mechanisms for subjects and roles

1. Cryptographic Protection of Subjects

- **Public Key Infrastructure:** Each subject is associated with a public key generated during the credential creation process. The public key is distributed for secure interactions, while the private key is securely stored to prevent unauthorized access.

```
def generate_subject_credentials(password, credentials_file):
    # hash the password
    salt = os.urandom(16)
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
        backend=default_backend(),
    )
    password = password.encode()
    key = kdf.derive(password)
    key = int.from_bytes(key, "big")

    private_key = ec.derive_private_key(key, ec.SECP256R1())

    private_pem = private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.NoEncryption(),  # No encryption for private ke
    )

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

    # Create credentials structure
    credentials = {
        "public_key": base64.b64encode(public_pem).decode(),
        "private_key": base64.b64encode(private_pem).decode(),
        "salt": base64.b64encode(salt).decode(),
    }

    # Save credentials to the file
    with open(credentials_file, "w") as f:
        json.dump(credentials, f, indent=4)

    return credentials
```

In this project we used **Elliptic Curve Cryptography (ECC)** and **PBKDF2HMAC** - **ECC** was used for secure key exchange (via ECDH) and digital signatures, providing strong encryption with smaller key sizes - For key derivation, we used **PBKDF2HMAC**. PBKDF2HMAC adds computational complexity through multiple iterations, making it resistant to brute-force attacks. This is particularly important for protecting user credentials and ensuring that derived keys are secure even if the underlying password is weak.

2. Prevention of Unauthorized Access

- **Permission Assignment:** Access control is enforced through a role-based permission system. Permissions are explicitly defined for each role, and subjects can only execute actions aligned with their assigned roles. This minimizes the risk of privilege escalation or unauthorized access to sensitive data and functions.
- **Granular Control:** Sensitive actions, such as creating or managing subjects and roles, require specific permissions. Only subjects assigned roles with these permissions can perform these operations, ensuring a clear separation of duties and reducing the attack surface.

Document

Documents maintain a **public-metadata**, a **private-metadata**, and an associated **file**.

Commands

- Change document ACL
- Upload a document
- Delete a document
- Download file (note that for this you need to know the file handle)
- Download a document metadata
- Download file encrypted yet
- List documents (with filtering options, such as dates or creators)

Security mechanisms for documents

- **ACL Management:** Ensures only authorized users can manage or access documents.
- **Authentication and Authorization:** Required for sensitive operations like metadata access, file downloads, and document deletion.
- **Encryption:** Files and sensitive metadata are encrypted to protect confidentiality.

```
def encrypt_file(file_path):
    try:
        key = os.urandom(32)
        iv = os.urandom(16)

        with open(file_path, "rb") as f:
            plaintext = f.read()

        pad_length = 16 - (len(plaintext) % 16)
        padded_plaintext = plaintext + bytes([pad_length] * pad_length)

        cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=default_backend())
        encryptor = cipher.encryptor()

        ciphertext = encryptor.update(padded_plaintext) + encryptor.finalize()

        encrypted_content = iv + ciphertext

        return base64.b64encode(key).decode("utf-8"), "AES-CBC", encrypted_content
    except Exception as e:
        raise Exception(f"Error during encryption: {e}")
```

- **Restricted Access to Metadata:** Public and private metadata are treated differently, with stricter access controls for private information.

Session

A session is a security context between the Repository and a subject.

Commands

- Create session
- Every command in the authenticated and authorized API

Security mechanisms for sessions

Our system uses symmetric key pairs generated with Diffie-Hellman to create a private communication channel between the client and the repository.

A session is associated with an organization and a subject's credentials. After a message exchange with the server a shared key is created in the client and server, allowing for secure messaging from that point on.

Key Exchange: both the client and the server have an ECC key pair and these keys are combined to make a shared key without ever needing to exchange sensitive information.

```
def derive_shared_key(private_key, peer_public_key):
    """Derive the shared key using ECC private and public keys"""
    peer_public_key = deserialize_public_key(peer_public_key)

    shared_key = private_key.exchange(ec.ECDH(), peer_public_key)

    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b"handshake data",
    ).derive(shared_key)

    return derived_key
```

Sessions are identified by a unique uuid and in the client is stored:

- **shared_key**: for message confidenciality
- **org:** to identify the organization associated with the session
- **username:** the username associated with the session
- **private_key:** used to sign messages

In the server a session contains the same fields but instead of the private key the server stores the public key associated with the session so that it can confirm the identity of the messenger. It also contains a life span for each session and a list of roles associated with that session.

Session validation:

For the authenticated and authorized API a session is needed to use the commands.

The server receives the payload from the client and with the `session_id` the session's key is retrieved and used to decrypt the message. The signature (ECDSA) is verified as well as the session's life span and permissions. Encrypted messages also employ **AES-256-GCM** generating a tag that ensures the integrity of the message.

```
def decrypt_message(encrypted_payload, shared_key, public_key=None):
    shared_key = shared_key.encode("utf-8")
    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        backend=default_backend(),
        info=b"handshake data",
    ).derive(shared_key)
```

```
    iv = base64.b64decode(encrypted_payload["iv"])
    ciphertext = base64.b64decode(encrypted_payload["ciphertext"])
    tag = base64.b64decode(encrypted_payload["certificate"]["message_integrity_tag"])
    signature = None
    if "signature" in encrypted_payload["certificate"] and public_key:
        signature = base64.b64decode(encrypted_payload["certificate"]["signature"])

    cipher = Cipher(
        algorithms.AES(derived_key), modes.GCM(iv), backend=default_backend()
    )
    decryptor = cipher.decryptor()

    decrypted_message = decryptor.update(ciphertext) + decryptor.finalize_with_tag(tag)

    if public_key:
        if isinstance(public_key, str):
            public_key = public_key.encode("utf-8")
            public_key = base64.b64decode(public_key)
            public_key = serialization.load_pem_public_key(
                public_key,
                backend=default_backend(),
            )

        public_key.verify(
            signature,
            decrypted_message,
            ec.ECDSA(hashes.SHA256()),
        )

    return json.loads(decrypted_message.decode("utf-8"))
```

ASVS analysis

- The application was evaluated under the Chapter 6 - Stored Cryptography - of the OWASP Application Security Verification Standard 4.0.3 (October 2021 version).

6.1 - Data Classification

6.1.1

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.

Applicability: Not implemented, not Applicable.

Justification: The only PII that is dealt with in this project are: `private key`, `username`, `name` , `email` and `password` .

Relatively to the `private key`, there's no need to store it encrypted. It is generated using the hash of the user's password.

```
def generate_subject_credentials(password, credentials_file):
    # hash the password
    salt = os.urandom(16)
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
        backend=default_backend(),
    )
    password = password.encode()
    key = kdf.derive(password)
    key = int.from_bytes(key, "big")

    private_key = ec.derive_private_key(key, ec.SECP256R1())
    ...
```

Relatively to the `username`, `name` and `email`, these are stored in plaintext. It is evident that this makes them vulnerable to unauthorized access, which can lead to data breaches and loss of user trust. However, their exploitation would never put the user in a position of risk. Even if someone were to gain access to this information, they would not be able to use it for any meaningful or malicious purpose, as these pieces of data alone do not grant access to sensitive systems or resources, nor do they expose critical personal or financial information. Furthermore, there are some benefits on not encrypting these:

- Operational necessity for plaintext access: It facilitates the user authentication, communication, and efficient data retrieval.
- Performance and efficiency considerations: Encrypting these fields would introduce significant processing overhead, negatively impacting the application's performance and scalability.

Relatively to the `password`, they are not stored at all, they are just used to generate the user's private key.

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.

Applicability: Not Implemented. This control is **not applicable** to our application as it does not handle any regulated health data.

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.

Applicability: Not implemented. This control is **not applicable** to our application as it does not handle any regulated financial data.

6.2 - Algorithms

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.

Applicability: Applicable

Benefit: The application deals with file storage, which involves sensitive data. If cryptographic operations fail insecurely, it could expose sensitive information or allow attackers to exploit vulnerabilities like Padding Oracle attacks. The code utilizes and AES-GCM encryption mechanisms, which are known to prevent common cryptographic flaws like Padding Oracle attacks.

```
def encrypt_message(message, shared_key, private_key=None):
    message_json = json.dumps(message).encode("utf-8")

    shared_key = shared_key.encode("utf-8")

    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        backend=default_backend(),
        info=b"handshake data",
    ).derive(shared_key)

    iv = os.urandom(16)

    cipher = Cipher(
        algorithms.AES(derived_key), modes.GCM(iv), backend=default_backend()
    )
    encryptor = cipher.encryptor()

    ciphertext = encryptor.update(message_json) + encryptor.finalize()

    signature = None
    if private_key and private_key != "":
        if message_json is None or message_json == "":
            message_json = b""

        if isinstance(private_key, str):
            private_key = base64.b64decode(private_key)

        private_key = load_pem_private_key(
            private_key,
            password=None,
            backend=default_backend(),
        )

        if not isinstance(message_json, bytes):
            message_json = message_json.encode("utf-8")
        signature = sign_message(private_key, message_json)
```

```
    certificate = {
        "encryption_method": "AES-256-GCM",
        "key_derivation": "HKDF-SHA256",
        "timestamp": time.time(),
        "message_integrity_tag": base64.b64encode(encryptor.tag).decode("utf-8"),
    }

    if signature:
        certificate["signature"] = base64.b64encode(signature).decode("utf-8")

    encrypted_payload = {
        "iv": base64.b64encode(iv).decode("utf-8"),
        "ciphertext": base64.b64encode(ciphertext).decode("utf-8"),
        "certificate": certificate,
    }

    return encrypted_payload
```

6.2.2

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

Applicability: Implemented

Benefit: File storage services handle sensitive data, making strong cryptography crucial to prevent attacks. Using proven cryptographic libraries reduces the risk of side-channel attacks and implementation flaws.

Primitives used:

- **Elliptic Curve Cryptography (ECC)**:
  - `SECP256R1` (NIST P-256)
  - **ECDSA** for digital signatures
  - **ECDH** for key exchange
- **Symmetric Encryption**:
  - **AES-256-GCM** for authenticated encryption
- **Key Derivation**:
  - **HKDF-SHA256**
- **Hashing**:
  - **SHA-256**

These are **industry-standard choices** and align with recommendations from trusted regulatory bodies.

6.2.3

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

Applicability: Applicable

Benefit: Proper IV and cipher configurations prevent vulnerabilities like IV reuse or padding oracle attacks.

- Each encryption operation uses a unique IV, reducing risks of IV reuse `os.urandom(16)`
- Used **AES-256-GCM** (`algorithms.AES(derived_key), modes.GCM(iv)`) which is secure

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.

Applicability: Aplicable

Benefit: Its is useful for rapid response to vulnerabilities and addition of newer algorithms

Hardcode problem: Most of the modules are hardcoded and a swap would mean a change in the code, not in runtime. This creates a problem that if there's a breach the configuration can't be quickly changed to fix the issue.

Possible solution: A possible solution would be to make the desired algorithm a parameter in messages or in the application settings and select it at runtime

```
{
    "algorithm": "AES-GCM",
    "key_size": 256,
    "iv": "base64encodedIV",
    "tag": "base64encodedTag"
}
```

```
def get_cipher(mode="GCM", key=None, iv=None):
    if mode == "GCM":
        return Cipher(algorithms.AES(key), modes.GCM(iv))
    elif mode == "CBC":
        return Cipher(algorithms.AES(key), modes.CBC(iv))
    else:
        raise ValueError("Unsupported mode")
```

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.

Applicability: Applicable

Benefit: Prevents vulnerabilities like padding oracle attacks.

Block Modes

- Uses **AES-GCM** (authenticated encryption).

Padding Modes

- **No padding** used - AES-GCM

Ciphers

- Uses **AES-256**

Hashing Algorithms

- Uses **SHA-256** (secure, part of SHA-2 family).

Key Exchange and Signatures

- Uses **ECDSA** (with SHA-256) and **ECDH** (NIST-approved).

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.

Applicability: Applicable.

Unique Initialization Vectors (IVs): A unique IV is generated for each encryption operation using a cryptographically secure random number generator (`os.urandom`), ensuring that IVs are not reused with the same encryption key.

Benefit: This practice prevents vulnerabilities associated with IV reuse, maintaining the security and integrity of encrypted data.

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.

Applicability: Applicable

Authenticated Encryption Using AES-GCM: The application encrypts users' private keys using AES-GCM (Advanced Encryption Standard - Galois/Counter Mode), an authenticated cipher mode that provides both confidentiality and integrity for the encrypted data.

Benefit: This ensures that any alteration of the can be detected during the decryption process, preventing unauthorized modifications and maintaining the integrity of the encrypted data.

Evidence:

```
cipher = Cipher(algorithms.AES(derived_key), modes.GCM(iv), backend=default_backend()

encryptor = cipher.encryptor()

certificate = {
    "encryption_method": "AES-256-GCM",
    "key_derivation": "HKDF-SHA256",
    "timestamp": time.time(),
    "message_integrity_tag": base64.b64encode(encryptor.tag).decode("utf-8"),
}
```

6.2.8

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

Applicability: Implemented.

Constant-Time Cryptographic Operations: The application utilizes the `cryptography` library, which provides constant-time implementations for all critical cryptographic operations, including encryption, decryption, and secure comparisons.

Benefit: By leveraging the `cryptography` library, the application ensures that all cryptographic operations are executed in constant-time, effectively mitigating the risk of timing attacks that could exploit variations in operation execution time to infer sensitive information.

6.3 - Random Values

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.

Benefit: Using cryptographically secure random number generators (CSPRNGs) ensures unpredictability, prevents collision and predictability attacks, strengthens cryptographic systems, and facilitates compliance with security standards, safeguarding sensitive resources and reducing vulnerabilities.

Applicability: We believe the implementation of this requirement is fundamental for the developed project, precisely because of the benefits that using this type of random values brings to us.

Evidences:

```
def encrypt_file(file_path):
    try:
        key = os.urandom(32)
        iv = os.urandom(16)
```

```
def generate_subject_credentials(password, credentials_file):
    salt = os.urandom(16)
```

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.

Benefit: Using the GUID v4 algorithm with a Cryptographically-secure Pseudo-random Number Generator (CSPRNG) ensures unique and unpredictable identifiers, aligns with industry standards, enhances security against enumeration or brute-force attacks, and strengthens reliability in distributed systems.

Applicability: We believe the implementation of this requirement is fundamental for the developed project, precisely because of the benefits that using this type of random values brings to us.

Evidence:

```
 document_id = str(uuid.uuid4())

 key, alg, ciphertext = encrypt_file(file_path)
```

```
@session.route("/create", methods=["POST"])
def create_session_controller():
    payload = request.json
    if not payload:
        return {"error": "No payload provided"}, 400

    client_public_key = payload.get("public_key")
    org = payload.get("organization")
    username = payload.get("username")
    signature = payload.get("signature")

    session_id = uuid.uuid4()
```

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.

Applicability: Implemented

Benefit: Ensuring that random numbers are generated with proper entropy under heavy load guarantees the unpredictability and security of critical operations, such as cryptographic key generation, session management, and token creation. This minimizes the risk of attacks exploiting predictable random values and ensures the application remains reliable and secure, even in high-traffic scenarios.

Evidence: The application uses `os.urandom` for generating secure keys and initialization vectors (IVs), and `uuid.uuid4()` for creating unique document and session IDs, both of which rely on the operating system's secure entropy sources, ensuring unpredictability and resilience under heavy load.

6.4 - Secret Management

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.

Applicability: Partially Implemented.

Benefit: Using a secrets management solution, such as a key vault, ensures that sensitive data is securely created, stored, and managed. This minimizes the risk of unauthorized access, facilitates controlled access through role-based permissions, supports secure rotation and destruction of secrets, and improves overall compliance with security standards and best practices.

Evidence: The application partially implements a secrets management solution by creating, storing, and controlling access to sensitive data using a key vault. However, the current implementation does not include mechanisms for securely destroying secrets, which leaves this aspect incomplete.


```
{
    "sessions": {
        "95561b02-20b2-43a8-98bb-1f91ab151256": {
            "shared_key": "uTUktPgVubxPEJyFDyx6J4dbe/tXh7Y47YhHPatf3E4=",
            "life_span": 1738109745.7796636,
            "org": "org1",
            "client_public_key": "LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUZrd0V3WUhLb1pJemowQ0F
            "roles": [
                "role2"
            ]
        }
    }
}
```


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.

Applicability: Implemented

Benefit: Ensuring that key material is not exposed to the application but handled by an isolated security module, such as a vault, significantly reduces the attack surface by preventing unauthorized access to sensitive cryptographic keys. It enhances security by delegating cryptographic operations to a trusted environment, supports secure key lifecycle management, and aligns with industry standards for protecting cryptographic material.

Evidence: The key material is securely stored and managed within an isolated module, ensuring it is never exposed to the client side. All cryptographic operations are performed within this security module, never being exposed and therefore reducing the risk of key information being leaked.