SIO Project Report

LEI, December 2024

Work done by:

David Amorim, 112610

Francisca Silva, 112841

Guilherme Amaral, 113207

Table of Contents

Introduction

The goal of this project is to develop a Repository for documents for organizations that can be securely shared among several people. For this objective a group of guidelines for its internal functioning and commands was given to us, which we followed thoroughly, making our own decisions on the topics that were not defined in the mentioned. Besides that, it was also requested an analysis of our program based on a chosen chapter of the ASVS.

Our choice was to analyze each control of chapter **V6** (**Stored Cryptography**) since it was very related to one of the main topics of SIO, encryption algorithms. Maintaining secure and authenticated message exchanges is one of our main priorities during the development of this project. We saw this chapter as the best way to display the effort and thought put into the project.

Repository

Storage

From the server side, persistence of documents (public) metadata, organizations and subjects were implemented using PostgreSQL as a database. The files and private metadata are stored on disk.

Sessions are not persistent, which means, in case of server restart, all previous sessions stop being valid.

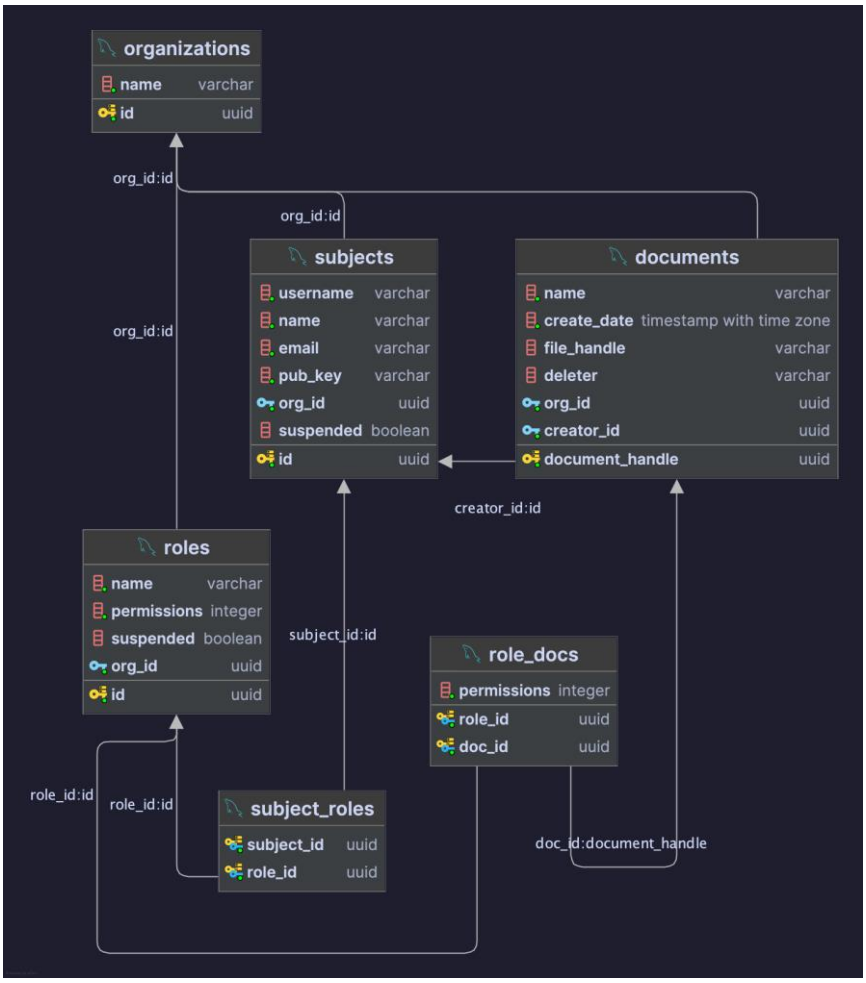

Fig. 1 – Database Diagram

Master Key

The server master_key is derived from a password asked by the server once it starts (in case of using our docker compose file, the password is pre-defined for ease of use). This master_key will be used to create the repository private/public key pairs, that should be known by each client and will be stored with the name repo_key.pub when the repository is started for the first time. This master_key is also used to derive a symmetric key, that will be used to encrypt the document's metadata locally.

Algorithms

Symmetric encryption: AES256 with CBC mode, because it's a proven secure cryptographic algorithm.

Digest operations: SHA256, because it's secure due to its 256-bit output and collision resistance.

Asymmetric encryption: ECC, because it provides more cryptographic strength with smaller keys than RSA. And we use ECDH for getting a shared secret and ECDSA for signatures.

Password key derivation: PBKDF2

Symmetric encryption padding: PKCS7

Sessions

Create session flow

- We start by generating an ephemeral EC key pair on the client side.
- We construct a JSON object with the organization name and the subject's username.
- Using the private key derived from the password (using PBKDF2 to derive an EC private key), we sign the ephemeral public key, and the contents of the JSON object created above.
- The client, already having the server's public key, derives a 64-byte shared secret by ECDH.

- The first 32 bytes of the shared secret are the symmetric key, and the last 32 bytes are used for HMAC.
- We construct the packet | json_size | json | signature |.
- We encrypt that packet with AES256 in CBC mode, with the derived symmetric key. (Let's call this encrypted packet *ciphered_body*).
- We construct the final packet |eph_pub_key_size|eph_pub_key|ciphered_body| and send it to the server.
- The server extracts the ephemeral public key, derives the same shared secret as the client, and uses the symmetric key to decrypt the *ciphered_body*.
- Since the server already has the public key derived from the password stored in its database (it would have been sent previously, either during rep_create_org or rep_add_subject), it validates the ephemeral public key and the JSON content against the signature using this public key.
- If the validation is successful, the server accepts the request and sends an encrypted packet containing the generated session id (UUID v4) and a HMAC calculated for the encrypted body.
- The client, after validating the MAC, decrypts the server's response content using the first part of the shared secret.
- The client stores the keys and the session_id in the session file, and the server stores them in memory, so if the server goes down, all sessions are automatically invalidated.

From now on, all information shared between commands that require a session can be encrypted with the shared secret, using the keys stored by both sides, along with a sequence number (incremented in every request that uses the session), to prevent replay attacks, and a MAC to guarantee integrity and authentication. The sessions have a default expiration time of 1 hour.

Authentication

As the ephemeral pub key and json body containing the organization and subject name are signed with the subject's private key (generated with rep_subject_credentials), the server can authenticate the subject (via the subject’s public key stored in the database). This authentication is transitive, because in the subsequent requests in a session, the server identifies a subject by a session_id, and even if someone stoles the session_id, that would be useless, because they must also know the ephemeral private key and the current message sequence number, in order to communicate with the server using that session.

Session file

The current session file structure is:

| json size (2 bytes big endian) | json with { "session_id": "uuid", "expires_at": "timestamp" } | sequence number (4 bytes big endian) | 32 bytes secret key for encryption | 32 bytes secret key for MAC calculation |

The server also keeps this data, in memory in a dictionary with the key being the session id, and the value being an instance of our SessionContext class, that contains the organization_id, the subject_id, the symmetric and mac keys, the sequence number, the expiration time and a set of the current session role id's.

Non-sessioned endpoints

We have some endpoints that do not require a session (list orgs, create org, get file). For them we also implemented secure communications. These endpoints require special treatment, as the client does not have an active session with the server, and such, there are no shared keys between both parties.

The communication flow for those endpoints is the following:

- The client generates an ephemeral EC key pair and performs an ECDH with the server's public key, deriving a shared key.
- The client builds the JSON body and encrypts it with AES256-CBC with that shared key.
- The client sends to the server its ephemeral public key and the encrypted body.
- The server extracts the client ephemeral public key and performs an ECDH, deriving the same shared key.
- The server decrypts the JSON body and processes the request.
- The server responds with an encrypted JSON and a signature of that encrypted JSON, using ECDSA.
- The client verifies the signature with the server's public key.
- The client then decrypts the JSON body, if needed.

All the error messages (for example, organization does not exist, file does not exist, etc.) are also encrypted and signed the same way as described above.

Document Uploading

For document uploading, we followed the project rules. The client generates a symmetric key, encrypts that document, and uploads the encrypted document along with the secret key, the file handle, name, and cryptographic descriptions. Because the command for uploading the file requires a session, all this data is encrypted with the session key and a MAC is appended before making the HTTP request to the server. The cryptographic description is a JSON with the format { “crypto_alg”: “AES256_CBC”, “digest_alg”: “SHA256” }. (digest_alg is the algorithm for computing the file handle). When the server receives and decrypts that message, it validates if the cryptographic and digest algorithms are supported (for now they must be AES256_CBC and SHA256 as they are the only ones implemented, but in the future, we could support more).

The server stores the private metadata (json with cryptographic descriptions, secret key and IV) **encrypted** with the server’s **master key**. The remaining document metadata (public metadata) is stored in the Postgres database (see Fig. 1).

ACLs

In this application, there are roles. Roles have permissions within the organization but can also have a list of permissions (ACL) for each document in that organization.

For the permissions implementation, we have decided to use bitfields, so we can compress a list of permissions in a single integer, so in the database we only store an integer that represents a list of permissions, instead of storing all the permissions as strings and duplicating information.

We defined the permissions as an Int Enum, in python, where each permission is a number starting with ROLE_ACL = 1 << 0, SUBJECT_NEW = 1 << 1, ..., 1 << n, where n is the number of permissions.

Then for adding a permission to a role we only do a bitwise OR between the current role permissions and the permission to be added. For removing a permission, we do a bitwise AND between the current role permissions and the bitwise NOT of the permission to be removed, and to check if a role has a permission, we check if permission bit is active (1) in the role permissions bitfield.

V6 Stored Cryptography (ASVS Analysis)

V6.1 Data Classification

6.1.1 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.

CWE-311: Missing Encryption of Sensitive Data (L2, L3)

Our application does not comply with this requirement. We could improve our application by encrypting the Postgres database. This way all data would be encrypted.

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

CWE-311: Missing Encryption of Sensitive Data (L2, L3)

Since we can store any type of data in the document, including health data, this requirement applies. All the files are stored on the server side and encrypted. This way if we were attacked no one could have access to them.

6.1.3 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.

CWE-311: Missing Encryption of Sensitive Data (L2, L3)

Since we can store any type of data in the document, including financial data, this requirement applies. All the files are stored on the server side and encrypted. This way if we were attacked no one could have access to them.

V6.2 Algorithms

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

CWE-310: Cryptographic Issues (L1,L2, L3)

To prevent Padding Oracle attacks all the responses with error indicating if the padding failed are not sent to the client, so the client doesn't have access to them.

If we hadn't taken this precaution, this attack could allow an attacker to decrypt and encrypt private information which is dangerous and lacks confidentiality.

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

CWE-327: Use of a Broken or Risky Cryptographic Algorithm (L2, L3)

As mentioned above in the algorithms chapter, all the cryptographic algorithms, modes and libraries that we use are approved and recognized by the government and industry. We decided to use them because they are secure and trusted by many professionals.

We didn't use any custom cryptography, to reduce the number of vulnerabilities and the security risk.

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

CWE-326: Inadequate Encryption Strength (L2, L3)

We follow the latest advice in our application to configure securely iv, cipher configuration and block modes.

For the initialization vector, we use os.urandom() instead of random to ensure high-entropy randomness and we never reuse the same IV with the same key to ensure uniqueness.

We use AES256 for the security level, PKCS7 for the padding and the block mode CBC, because it's a proven secure cryptographic algorithm, as mentioned above.

6.2.4 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.

CWE-326: Inadequate Encryption Strength (L2, L3)

In both our client and server sides we have a specific file "**crypto.py**" where all the encryption and hashing algorithms are implemented, therefore, reconfiguring, updating or swapping the algorithms or its values (key lengths, iv's, salt, etc) is easy and very direct. These files have functions that execute the different algorithms and have several input arguments that can be easily changed upon call from the different commands. Being both the encryption and decryption functions defined in the file, one can easily reconstruct the whole algorithm without needing to change the code in each command.

One single exception is in the **rep_subject_credentials**, in which we don't use the crypto file to execute the algorithms used, but which can be easily changed upon a new update.

Therefore, in general we comply with this control.

```python
def encrypt_aes256_cbc(plaintext: bytes, secret_key: bytes, iv: bytes) -> bytes:
    algorithm = algorithms.AES256(secret_key)

    padder = PKCS7(algorithm.block_size).padder()
    padded_data = padder.update(plaintext) + padder.finalize()

    cipher = Cipher(algorithm, modes.CBC(iv))
    encryptor = cipher.encryptor()

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

```python
def decrypt_aes256_cbc(secret_key: bytes, iv: bytes, ciphertext: bytes) -> bytes:
    algorithm = algorithms.AES256(secret_key)
    cipher = Cipher(algorithm, modes.CBC(iv))
    decryptor = cipher.decryptor()

    padded_data = decryptor.update(ciphertext) + decryptor.finalize()

    unpadder = PKCS7(algorithm.block_size).unpadder()

    return unpadder.update(padded_data) + unpadder.finalize()
```

Fig.2 - Example of Functions in crypto.py

6.2.5 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.

CWE-326: Inadequate Encryption Strength (L2, L3)

As mentioned in the **Algorithms** section, all the algorithms we use are secure, since, for example, we use CBC instead of ECB, AES256 to guarantee big block sizes, and strong hashing algorithms such as SHA-256 instead of MD5 or other weaker algorithms, all this while using PKCS7 for padding. There is no current backwards compatibility configuration in our program, so insecure algorithms aren't used for that either.

We can conclude that our application complies with this control.

6.2.6 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.

CWE-326: Inadequate Encryption Strength (L2, L3)

When we need to use single numbers, such as IV's, in our application, both on the client and server side they are passed to the functions present in "crypto.py" as arguments. These numbers are always generated using secure methods, since we use **os.urandom()** instead of **random**, which ensures secure randomness. Besides that, we ensure uniqueness in each number by never reusing the same IV more than once with a key, since for each message exchange (even with the same key) we generate a new IV before each encryption.

Therefore, it is safe to conclude that we comply with this control.

```
iv = os.urandom(16)
signature = sign_ec_dsa(priv_key, serialized_pub_key + body)
cipherbody = iv + encrypt_aes256_cbc(body_size + body + signature, secret_key, iv)
data = len(serialized_pub_key).to_bytes(2, "big") + serialized_pub_key + cipherbody
```

Fig. 3 - Example of IV being generated randomly before each use

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

CWE-326: Inadequate Encryption Strength (L3)

In all the endpoints that use a session, the message is appended with a HMAC, in the client -> server the message is `IV + encrypted json + sequence number (4 bytes big endian) + MAC(IV+ encrypted json + seq number)`, then the server validates the MAC before trying to decrypt anything. In the server -> client response, the message is `IV + encrypted json + MAC(IV + encrypted json)`, then the client validates the MAC before decrypting the json. In the endpoints that do not require a session, every response from the server is signed with the server's private key using ECDSA algorithm. The client then validates that signature with the server's public key and then decrypts the data.

```python
def encrypt_body(body: bytes, secret_key: bytes, mac_key: bytes) -> bytes:
    iv = os.urandom(16)

    data = iv + encrypt_aes256_cbc(body, secret_key, iv)
    mac = compute_hmac(data, mac_key)

    return data + mac
```

Fig. 4 – Data encryption server side

```python
# Must at least have the seq + MAC
if len(data) < 4 + 32:
    res = { "message": "Message is too short" }
    return json.dumps(res), 400

seq = int.from_bytes(data[-32 - 4:-32], "big")
mac = data[-32:]

if not verify_hmac(data[:-32], mac, session.mac_key):
    res = { "message": "Request body integrity check failed" }
    return json.dumps(res), 400

if seq != session.seq:
    res = { "message": "Sequence number mismatch" }
    return json.dumps(res), 400

# If we actually have a request body
if len(data) > 32 + 4:
    iv = data[:16]
    ciphertext = data[16:-32 - 4]
    plaintext = decrypt_aes256_cbc(session.secret_key, iv, ciphertext)
    g.json = json.loads(plaintext)

session.seq += 1
```

Fig. 5 – Data validation and decryption server-side

```python
def encrypt_body(body: bytes, seq: int, secret_key: bytes, mac_key: bytes) -> bytes:
    iv = os.urandom(16)
    cipherbody = iv + encrypt_aes256_cbc(body, secret_key, iv) + seq.to_bytes(4, "big")
    mac = compute_hmac(cipherbody, mac_key)

    return cipherbody + mac

def decrypt_body(data: bytes, secret_key: bytes, mac_key: bytes) -> bytes:
    if len(data) < 16 + 32:
        raise Exception("Message is too short")

    iv = data[:16]
    cipherbody = data[16:-32]
    mac = data[-32:]

    if not verify_hmac(iv + cipherbody, mac, mac_key):
        raise Exception("Integrity verification failed")

    return decrypt_aes256_cbc(secret_key, iv, cipherbody)
```

Fig. 6 – Data encryption and data validation & decryption client-side

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

CWE-385: Covert Timing Channel (L3)

This requirement is important to prevent timing attacks, so the execution time of cryptographic operations should not depend on the input data, like the secret keys, IVs, etc. If all the cryptographic operations are constant time, a hacker cannot infer things based on how long an operation took to ran.

For all cryptographic operations, we use the cryptography python library. This library depends on OpenSSL C library for all cryptographic operations, as stated here https://cryptography.io/en/latest/openssl/, which implement constant time functions (for example in comparisons, even if they find a byte that mismatches, they continue comparing, to be constant time).

However, to test this, we made a small python script for comparing the average wall times of some encryption and decryption operations, using AES256_CBC (the same algorithm we use in our project's symmetric encryption operations).

```
def test_encrypt(data):
    times = []

    for _ in range(16):
        iv = os.urandom(16)
        secret_key = os.urandom(32)
        t = time.monotonic_ns()
        encrypted = encrypt_aes256_cbc(data, secret_key, iv)
        wall_time = time.monotonic_ns() - t

        times.append(wall_time)

    return round(fmean(times))
```

Fig. 7 – Test function for measuring encryption wall times.

```
def test_decrypt(iv, secret_key, ciphertext):
    times = []

    for _ in range(16):
        t = time.monotonic_ns()
        try:
            decrypted = decrypt_aes256_cbc(secret_key, iv, ciphertext)
        except:
            pass
        wall_time = time.monotonic_ns() - t

        times.append(wall_time)

    return round(fmean(times))
```

Fig. 8 – Test function for measuring decryption wall times.

```python
def main():
    data1 = b'A' * 1024
    data2 = os.urandom(1024)
    data3 = b'AA'

    t1 = test_encrypt(data1)
    t2 = test_encrypt(data2)
    t3 = test_encrypt(data3)

    print(f"1024 A's = {t1}ns")
    print(f"Random 1024 bytes = {t2}ns")
    print(f"AA = {t3}ns")

    iv = os.urandom(16)
    secret_key = os.urandom(32)
    ciphertext = encrypt_aes256_cbc(data1, secret_key, iv)

    t4 = test_decrypt(iv, secret_key, ciphertext)
    wrong_sk = os.urandom(32)
    t5 = test_decrypt(iv, wrong_sk, ciphertext)
    wrong_iv = os.urandom(16)
    t6 = test_decrypt(wrong_iv, secret_key, ciphertext)
    wrong_ciphertext = os.urandom(len(ciphertext))
    t7 = test_decrypt(iv, secret_key, wrong_ciphertext)

    print(f"Right decryption = {t4}ns")
    print(f"Wrong secret key = {t5}ns")
    print(f"Wrong IV = {t6}ns")
    print(f"Wrong ciphertext = {t7}ns")
```

Fig. 9 – Main function for benchmarking encryption and decryption

```
❯ python3 main.py
1024 A's = 100130ns
Random 1024 bytes = 8562ns
AA = 6883ns
Right decryption = 7693ns
Wrong secret key = 7393ns
Wrong IV = 6836ns
Wrong ciphertext = 7031ns
```

Fig. 10 – Benchmark results

With these results, we can conclude that the cryptographic operations are nearly constant time. The differences are not significative (10^-7 seconds) in decryption times. And because this is a server application, with the network latency, it would be nearly impossible for an attacker to infer anything related to secret keys and IVs with the response times.

V6.3 Random Values

6.3.1 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.

CWE-338: Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG) (L2, L3)

As mentioned earlier, all random numbers used, such as salts and IV, are created using **os.urandom()**, which is a secure random number generator. The file handle for each file (which is supposed to be known only by few people) is also generated using a sha256 digest based on each file content, resistant to collisions and therefore also secure and hard to be guessed by an attacker. Each of our orgs, subjects and roles UUID in the database is also generated randomly using **uuid.uuid4** which is secure, although they are not accessible through interactions with our app, therefore not aligning with this control.

To conclude, we comply with this control.

```
iv = os.urandom(16)
```

```
id = db.Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
```

Fig. 11 - Examples of use

6.3.2 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.

CWE-338: Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG) (L2, L3)

As confirmed in the previous control, our app uses the **uuid.uuid4** to generate random UUIDs (since we do not use GUIDs), along with **os.urandom()** which is a **CSPRNG** for other values.

Therefore, we comply with this control.

6.3.3 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.

CWE-338: Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG) (L2, L3)

Since we use a **CSPRNG** as random number generator, even when our application is under heavy load, numbers are generated with proper entropy, being unpredictable. To prove that we went and executed the following code to simulate a heavy load of requests to one of the main CSPRNG used by our program, **os.urandom()**, which returned fast responses without any error and/or non unique values. 0 duplicated values were returned.

```python
import threading
import os
from queue import Queue

def generate_secure_tokens(n, output_queue):
    generated = []
    for _ in range(n):
        token = os.urandom(16)
        generated.append(int.from_bytes(token, byteorder='big'))
    output_queue.put(generated)

results_queue = Queue()

threads = [threading.Thread(target=generate_secure_tokens, args=(100, results_queue)) for _ in range(10)]

for thread in threads:
    thread.start()

for thread in threads:
    thread.join()

results = []
while not results_queue.empty():
    results.extend(results_queue.get())

duplicates = len([x for x in results if results.count(x) > 1])

print(f"Results: {results}")
print(f"Duplicated values: {duplicates}")
```

Fig. 12 – Entropy tester code

This way, it is expected that our app generators will not decrease in entropy even when under heavy load and so, even though we do not present any graceful degradation mode for it, it will comply with this control.

V6.4 Secret Management

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

CWE-798: Use of Hard-coded Credentials (L2, L3)

This requirement is not applicable to our project, because we do not store any secrets.

Even though our application does not store secrets, if it was the case, this is an important requirement to consider, because if a hacker exploits our application and gain access to its memory space, it could steal the secrets.

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

CWE-320: Key Management Errors (L2, L3)

Our application does not comply with this requirement. Our application stores key material in-memory and does not perform cryptographic operations on specific hardware like TPMs. If a hacker exploits this app and gain access to its memory space, he could possibly steal some key material, like the session keys, etc.

We could improve this by storing the keys in a vault for cryptographic operations, for example AWS KMS, Google Cloud KMS, or even try to use hardware modules for this like the TPM.

Conclusion

With this project we could apply concepts learned throughout the semester in this subject and understand the importance of the high impact of CWE's. Through the project, our focus was on the security of our application, that is, that documents and organizations could be securely shared among several users, being resistant to external attacks such as man in the middle and replays.

By analyzing one of the chapters of the ASVS, we learned the importance of having the project well defined and the impacts each control can have on our system security. With this analysis we were also able to identify how easily one can end up falling in a rabbit hole and using the wrong tools from the large catalogue that is available (choosing a wrong algorithm is one example).

Reviewing everything that was done and learned, it's safe to conclude that the project allowed us to gain proper knowledge and experience with real-life issues, being very useful outside of the subject.