



universidade de aveiro
departamento de electrónica,
telecomunicações e informática

Segurança Informática e nas Organizações

Project analysis

André Ribeiro
NMec: 112974

Bruno Lopes
NMec: 68264

Rúben Garrido
NMec: 107927

Violeta Ramos
NMec: 113170

December 29, 2024

Contents

1	Introduction	2
2	Security measures	3
2.1	Encryption	3
2.1.1	URL encryption & escaping	3
2.1.2	Body encryption	3
2.1.3	Same key, different IVs	4
2.2	Hash-based Message Authentication Code (HMAC)	4
2.3	Response obfuscation	4
2.4	SQL injection avoidance	4
2.5	Parameters validation	4
2.6	Session & user validation	4
3	Session	5
3.1	Structure	5
3.2	Token validity & server-side validation	5
3.3	Session data leak issue	5
4	Analysis through OWASP ASVS	7
5	Conclusion	14
6	Bibliography	15

1 Introduction

Within the scope of the IT Security and Organizations discipline, this project has the main goal of implementing a Repository service for documents with a focus to securely be shared between members in organizations.

In this manner, we facilitate secure storage, management, and access to documents and their respective metadata. In any organization, there is a hierarchy chain where each member must have limited access privileges within a company, so it includes access control (ACL) with subjects (users or applications) and their roles (permissions and responsibilities).

An important key on the implementation of the Repository is having a secure connection which is guaranteed through sessions dealing with the authentication of subjects enforcing confidentiality. This involves many aspects of IT security such as:

- managing private and public keys
- generation of IV's (Initialization Vectors) and nonces
- API's with endpoints that enable many operations such as upload and download of documents
- asymmetric and symmetric encryption and decryption of data to avoid many kind of attacks

It will be showned in this report the security implementations and features with compliance in the Application Security Verification Standard(ASVS) regarding controls applied.

2 Security measures

When dealing with sensitive data, security measures must be taken to ensure the confidentiality, integrity, and authenticity of the data, protecting it from unauthorized access, manipulation, and eavesdropping.

2.1 Encryption

Since no existing communication protection protocol could be used, plain use of the API would mean that any man-in-the-middle (MitM) attack could retrieve private data, including private keys and their passwords, as well as whole documents.

To mitigate this vulnerability, encryption is used to protect all requests and responses. Hybrid encryption with AES and RSA was chosen to support this, since it combines the infinite-length content size support of AES, with the versatility and security of RSA.

When sending a request to the repository, there are three types of encryption available, which depend on whether there is a session or not:

Session status	AES key	Response encryption
No	Auto-generated	Same key, different IV
Creating	Auto-generated	Response's AES key is asymmetrically encrypted with user's public key
Yes	Session key (check 3.1)	Same key, different IV

Table 1: Types of encryption available

On requests, all AES keys are asymmetrically encrypted with the repository's public key.

Relevant encrypted keys are sent through HTTP headers, such as **Authorization**, **Encryption** and **IV**. On production mode, all unencrypted requests are declined.

All encrypted data is sent encoded in base64, to ensure that all bytes are suitable for transport. To further enhance that compatibility, specially on HTTP headers, we decided to escape all `\n` and `\r` characters.

2.1.1 URL encryption & escaping

To prevent hackers from potentially reverse engineer the API request and response structures based on the endpoint, URLs are always encrypted and then base64-encoded, for both path and query parameters. This means that, for any external interceptor, the URL is just a bunch of alphanumeric unrelated characters, which vary on each request.

Additionally, query parameters were escaped before being encrypted, so that there would be no possible query injection. For example, without this measure, the dictionary `{"foo": "bar&baz"}` would be interpreted as two parameters — `foo=bar` and `baz` — instead of just one `foo=bar%26baz`.

2.1.2 Body encryption

In the same way as URLs, all request and response bodies are encrypted, so that there is no possible data leak.

2.1.3 Same key, different IVs

To ensure that two data are not encrypted with the same key stream and thus prevent potential decryption, a different unique IV is used in each data transfer, and even differs between request and response. This allows reusing the same AES key, while maintaining security against reuse attacks.

2.2 Hash-based Message Authentication Code (HMAC)

To prevent tampering, transmission errors and deliberate data changing, an HMAC is appended to both URL and body, that the client and the repository verify, depending on the data flow (request or response, respectively).

An Encrypt-then-MAC approach was chosen, to allow integrity checks before the decryption. SHA-256 was chosen for the hash function, with the encryption's symmetric key as the MAC key (check section 2.1 for more info on this).

2.3 Response obfuscation

Instead of returning the HTTP status code in the typical header, it always returns 200 OK, and the true code is embedded in the body, which is encrypted. This causes diffusion & confusion, and hardens attackers' possible reverse engineering, since all requests — valid or not — are returned as valid. Examples of response bodies:

Original	Obfuscated	Obfuscated (not found)
<pre>{ "id": 1, "name": "Example" }</pre>	<pre>{ "code": 200, "data": { "id": 1, "name": "Example" } }</pre>	<pre>{ "code": 404, "data": "Not found" }</pre>

Table 2: Body responses with code included

2.4 SQL injection avoidance

SQLModel, which is powered by SQLAlchemy under-the-hood, automatically escapes query inputs, which prevent any type of SQL injection. All database queries use SQLModel's abstraction functions instead of raw SQL, so that no query can have such vulnerability. This measure prevents internal & external attackers from querying the database to retrieve or modify sensitive data.

2.5 Parameters validation

FastAPI heavily relies on [Pydantic](#), a data validation library for Python, which is also abstracted through SQLModel in this project. All models have typing validations, which prevent users from injecting non-allowed, malicious values in data input requests; invalid requests return a 422 Unprocessable Entity status code, with a brief explanation about the error.

2.6 Session & user validation

Check section 3.2.

3 Session

Sessions are an essential part for managing user authentication and authorization. This section focuses on the structure, storage, and transport mechanisms, while dealing with possible vulnerabilities.

3.1 Structure

Sessions are stored in the database as a nullable JSON column in a PostgreSQL N:M table, which links subjects to organizations. The object has the following type structure:

```
{
  "id": UUID,
  "keys": list[str],
  "expires": datetime,
  "roles": set[str]
}
```

Sessions have a default expiry time of 30 minutes. The **roles** set include the loaded roles in the session, and **keys** specify the session AES keys.

To make sessions transportable to the client, the JSON Web Token (JWT) format was chosen, due to its well-known standardization and digital signature support, which guarantee that only the repository can create or modify such tokens.

3.2 Token validity & server-side validation

To make sure session tokens remain a secure way to handle user authentication & authorization, sessions are always verified against the database, for a server-side, non-tampering validation.

Sessions have a default expiry time of 30 minutes, after which they become invalid. However, due to UUID validation, when a user creates a new session, the last one gets implicitly invalidated, even if it was still valid.

When a role is loaded or removed from the session, a new JWT is issued with the same UUID, which means that the old token is still valid. This might lead to thinking that there is a security flaw, because an old token with a bigger set of roles could still authorize unwanted operations. However, this is not true, due to the server-side database verification: using either token result in the same session, which always contains the latest data - roles included.

In addition to the token validation, each request also validates whether a user is activated or not. Therefore, even if a session token is valid, an invalid user will result in an unauthorized request.

3.3 Session data leak issue

On our implementation, JWT tokens are stored unencrypted on the client. This arises a vulnerability, since any unwanted access - either physically or virtually - within the token's 30 minutes of lifetime can retrieve it and use it to perform actions on behalf of the user. While this might sound out of scope because it's not an implementation fault but a user one, the problem gets worse once we realize this can happen to a user with the **Managers** role.

Web development introduces several solutions to this problem, including XSS protection and HTTP-only cookies, which make them inaccessible to JavaScript. However, since we're using a Python-based CLI, these concepts don't apply, so other workarounds must be used.

Multiple solutions can overcome this problem, however, all of them have some caveats - either related to security or to usability. The following table summaries them:

Description	Advantages	Disadvantages
Encrypt with unknown key	<ul style="list-style-type: none"> • User cannot see the token contents • AES session key is safely stored from everyone 	<ul style="list-style-type: none"> • Client cannot get the AES session key to encrypt the request • Attackers can still use the encrypted JWT to query the repository
Encrypt with user's public key	<ul style="list-style-type: none"> • User can only see and use the token if decrypted with private key • AES session key is safely stored from attackers • User can still get the AES session key to encrypt the request 	<ul style="list-style-type: none"> • Current project guidelines don't allow this, since the private key and its password would be asked in every authenticated command • Attackers can still use the encrypted JWT to query the repository
Encrypt with user's public key and send it decrypted	<ul style="list-style-type: none"> • Same advantages of previous solution • Attackers cannot use the encrypted JWT to query the repository since it only accepts decrypted tokens • Tokens are still safely transported because the request is encrypted 	<ul style="list-style-type: none"> • Current project guidelines don't allow this, since the private key and its password would be asked in every authenticated command

Table 3: Possible solutions for this vulnerability

4 Analysis through OWASP ASVS

V3.1 Fundamental Session Management Security

Control Information

Control ID:	3.1.1
Description:	Verify the application never reveals session tokens in URL parameters.

Assessment Details

Applicability:	Applicable
Justification:	Our application encrypts the Url and body of every request with the key from session token, ensuring confusion and diffusion in every request.

Evidence

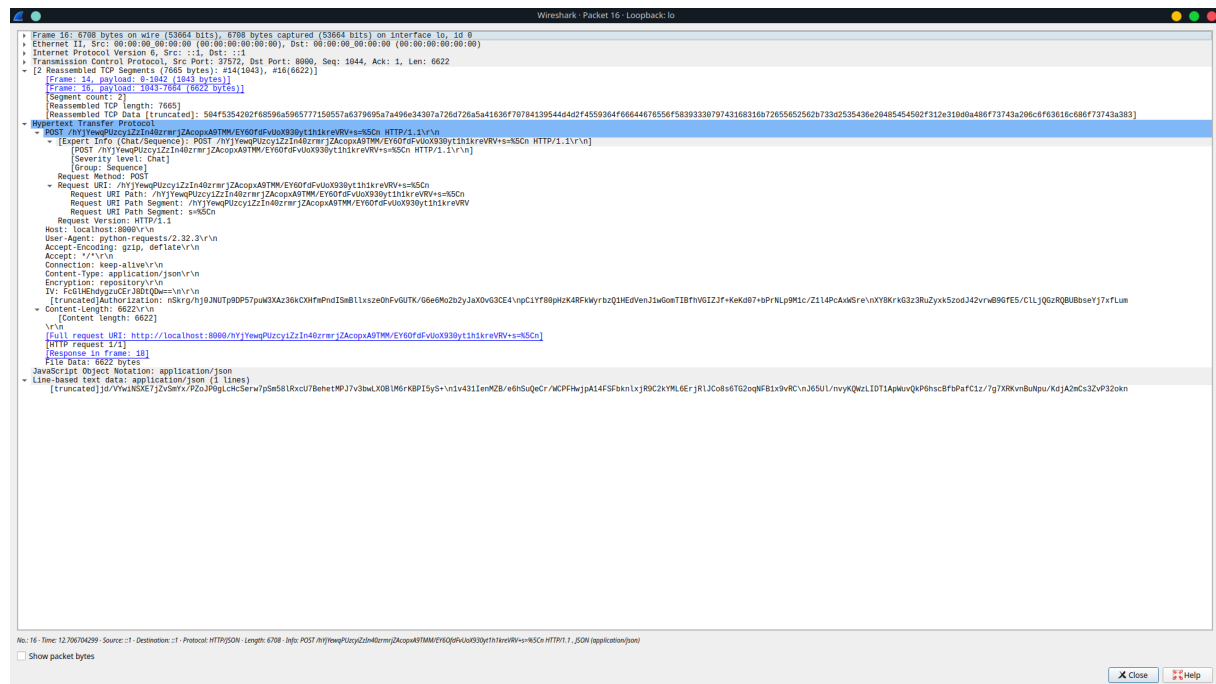


Figure 1: Session request showing encrypted URL and body

V3.2 Session Binding

Control Information

Control ID:	3.2.1
Description:	Verify the application generates a new session token on user.

Assessment Details

Applicability:	Applicable
Justification:	Whenever the user wants to make requests it must have a session. Otherwise, it must create one, in turn this new session will generate a new token.

Evidence

```
@app.command("rep_create_session")
def create_session(
    organization: str,
    username: str,
    password: str,
    private_key_file: Path,
    repository_public_key: RepPublicKey,
    repository_address: RepAddress,
    session_file: Annotated[Path | None, typer.Argument()] = None,
):
    with private_key_file.open() as f:
        private_key_str = f.read().encode()
        private_key = load_private_key(private_key_str, password)[0]

    obj: dict[str, str] = {
        "organization": organization,
        "username": username,
        "password": password,
        "credentials": base64.encodebytes(private_key_str).decode(),
    }

    body, _ = request_without_session_repo(
        "POST",
        repository_address,
        f"{SUBJECT_URL}/session",
        obj,
        private_key,
        repository_public_key,
    )
    token = body.strip('\"')

    session_file = (
        session_file
        or get_storage_dir()
        / "sessions"
        / organization
        / f".{sha512(username.encode()).hexdigest()}"
    )

    if not session_file.parent.exists():
        session_file.parent.mkdir(parents=True)

    with session_file.open("w+") as f:
        f.write(token)

    print(
        f"Session created for organization {organization} and user {username} at {session_file}"
    )
```

Figure 2: New Session token on User

Control Information

Control ID:	3.2.2
Description:	Verify that session tokens possess at least 64 bits of entropy.

Assessment Details

Applicability:	Applicable
Justification:	We have tokens with an entropy ranging between 20 and 343 bytes, ensuring confusion.

Evidence

JWT Information

- The header encoded has at least 24 characters (for a simple algorithm like HS256).
- The payload has at least 30 characters (considering a simple payload with minimal data).
- The signature has 43 characters (using the HMAC SHA256 algorithm with a secret key).
- Considering that each character occupies 1 byte it fulfills the requirements



```
$ echo ${({cat  
storage/sessions/UA/.408b27d3097eea5a46bf2ab6433a7234a33d5e49957b13ec7acc2ca08e1a13c75272c90c8d3385d47ede5420a7a9623aad817d9f8a70bd100a  
0acea7400daa59 | wc -c)*8)) bits  
2432 bits
```

Figure 3: Length in bits of a session token

Control Information

Control ID:	3.2.3
Description:	Verify the application only stores session tokens in the browser using secure methods such as appropriately secured cookies or HTML 5 session storage.

Assessment Details

Applicability:	Not applicable
Justification:	The project doesn't use any form of cookies.

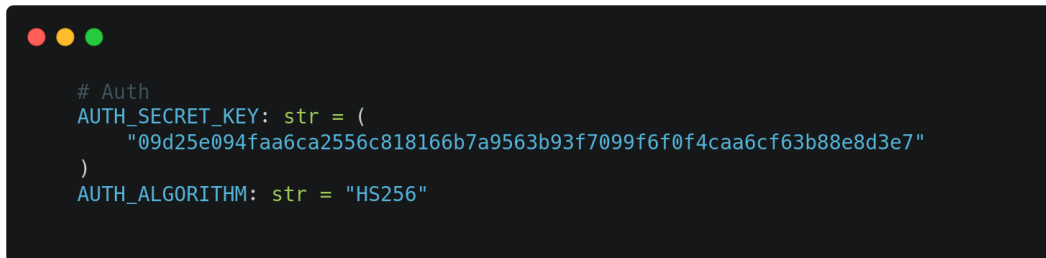
Control Information

Control ID:	3.2.4
Description:	Verify that session tokens are generated using approved cryptographic algorithms.

Assessment Details

Applicability:	Applicable
Justification:	The tokens are generated using HS256, in accord with the chapter Communications Security from OWASP.

Evidence



```
# Auth
AUTH_SECRET_KEY: str = (
    "09d25e094faa6ca2556c818166b7a9563b93f7099f6f0f4caa6cf63b88e8d3e7"
)
AUTH_ALGORITHM: str = "HS256"
```

Figure 4: Algorithm used to generate sessions

V3.3 Session Termination

Control Information

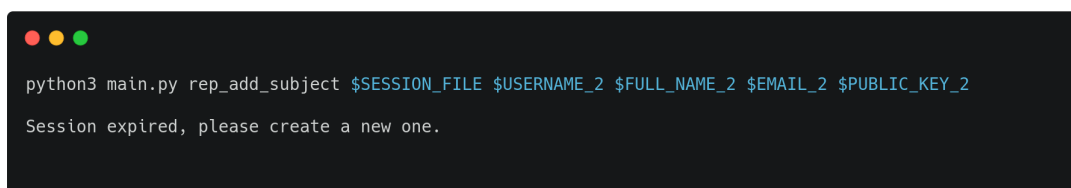
Control ID:	3.3.1
Description:	Verify that logout and expiration invalidate the session token, such that the back button or a downstream relying party does not resume an authenticated session, including across relying parties.

Assessment Details

Applicability:	Applicable
Justification:	The token's expiration causes all new requests to be encrypted with an invalid token, which is rejected by the server. To move forward, the user must renew his session (create a new session).

Evidence

After 30 minutes a session is completely invalid



```
python3 main.py rep_add_subject $SESSION_FILE $USERNAME_2 $FULL_NAME_2 $EMAIL_2 $PUBLIC_KEY_2
Session expired, please create a new one.
```

Figure 5: Example with a session expired

Control Information

Control ID:	3.3.2
Description:	If authenticators permit users to remain logged in, verify that re-authentication occurs periodically both when actively used or after an idle period.

Assessment Details

Applicability:	Not applicable
Justification:	This project doesn't use any form of refresh tokens. Once an session expires the user must create a new one

Control Information

Control ID:	3.3.3
Description:	Verify that the application gives the option to terminate all other active sessions after a successful password change (including change via password reset/recovery), and that this is effective across the application, federated login (if present), and any relying parties.

Assessment Details

Applicability:	Not applicable
Justification:	For this project, there is no concept of password, the user is acknowledged base on his public key.

Control Information

Control ID:	3.3.4
Description:	Verify that users are able to view and (having re-entered login credentials) log out of any or all currently active sessions and devices.

Assessment Details

Applicability:	Not applicable
Justification:	Logging out is not supported by the Service, sessions last for a period of 30 minutes before they cease to exist.

V3.4 Cookie-based Session Management

Assessment Details

Applicability:	Not applicable
Justification:	The app doesn't support any form of Cookies.

V3.5 Token-based Session Management

Control Information

Control ID:	3.5.1
Description:	Verify the application allows users to revoke OAuth tokens that form trust relationships with linked applications.

Assessment Details

Applicability:	Not applicable
Justification:	The app doesn't support any form of OAuth.

Control Information

Control ID:	3.5.2
Description:	Verify the application uses session tokens rather than static API secrets and keys, except with legacy implementations.

Assessment Details

Applicability:	Applicable
Justification:	The project uses JWT, which is composed of: Session key, Expire time, User identification, Uuid, Set of user roles.

Evidence

The session file has a JWT which is decoded bellow.

[illegible]

Figure 6: JWT decode

Control Information

Control ID:	3.5.3
Description:	Verify that stateless session tokens use digital signatures, encryption, and other countermeasures to protect against tampering, enveloping, replay, null cipher, and key substitution attacks.

Assessment Details

Applicability:	Not applicable
Justification:	This app doesn't use stateless sessions.

V3.6 Federated Re-authentication

Assessment Details

Applicability:	Not applicable
Justification:	Writing Relying Party (RP) and Credential Service Provider (CSP) are out of this project's scope.

V3.7 Defenses Against Session Management Exploits

Control Information

Control ID:	3.7.1
Description:	Verify the application ensures a full, valid login session or requires reauthentication or secondary verification before allowing any sensitive transactions or account modifications.

Assessment Details

Applicability:	Applicable
Justification:	All sensitive operations require a valid session, as such all others will be denied.

Evidence

As requested the commands who needs authentication are using a session so it fulfill the control.

5 Conclusion

In a nutshell, this report discussed the implementation of a secured document repository system and the importance of advanced security protocols to ensure that data maintains its confidentiality, integrity, and availability. The integration of encryption technologies, such as AES and RSA, combined with the use of HMAC for message authentication and techniques to prevent SQL injection attacks, builds a strong framework to fight against common security threats.

JWT-based authentication and server-side validation were integrated to address the session management, by ensuring that user sessions are protected and any potential leakage of data is avoided. Additionally, the report examined compliance with OWASP ASVS guidelines in the third chapter, verifying adherence to industry standards and best practices within that scope.

Even if one acknowledges some inherent challenges - such as vulnerabilities related to token storage in specific contexts - the proposed countermeasures depict a holistic approach for reducing risks. The outcome emphasizes the need for an integrated security framework and for continuous verification to safeguard private information.

Overall, this project addresses the proactive security strategies in the architecture of secure systems. Future research could discuss advanced token governance methodologies and other security measures to better enhance system resilience and user trust.

6 Bibliography

- Documentation
 - **FastAPI** <https://fastapi.tiangolo.com>
 - **SQLModel** <https://sqlmodel.tiangolo.com>
 - **SQLAlchemy** <https://docs.sqlalchemy.org/en/20/>
 - **Typer** <https://typer.tiangolo.com>
- Theoretical slides and practical labs on the IT Security and Organizations discipline subject page