SIO Project Report

Rúben Gomes (113435) · João Bastos (113470) · Tiago Garcia (114184) – 30/12/2024

Table of Contents

---

1. Introduction

This document serves as the final report for the SIO-2425 project. This project serves as a way to demonstrate the practical application of some of the concepts learned throughout the course (Authentication, Access Control, Session Management and Stored Cryptography). On a analysis perspective, it will be focused on the V3 (Session Management) chapter of the OWASP ASVS.

This report will cover the features implemented, the decisions made as a group, the analysis of the implementation, and finally the results and conclusions of the project.

2. Features

The features of the project are the ones present in the course project description, but with an extra feature, the possibility to reset the database of the server. This was shown to be useful for testing purposes, but it should be disabled/deleted in a production environment.

2.1. API Endpoints

The API has a list of endpoints that require different permission levels to access. Mainly, it's divided into 3 categories:

- Anonymous: No authentication required.
- Authenticated: Authentication required.
- Authorized: Authentication and permissions required.

2.1.1. Anonymous Endpoints

Endpoint	Required headers	Required payload fields	Optional payload parameters
GET / → Returns a ping message.	N/A	N/A	N/A
POST /reset → Resets the database and deletes all data.	• Content-Type: application/json	• password : The reset password. **Note: The reset password is** 123 **(very secure!).**	N/A
GET /org/list → Returns a list of all organizations.	N/A	N/A	N/A
POST /org/create → Creates a new organization.	• Content-Type: application/octet-stream	• name : Organization name. • username : Manager username. • full_name : Manager full name. • email : Manager email. • public_key : Manager public key.	N/A
GET /file/get/<file_handle>/content → Downloads the file content.	N/A	N/A	N/A
POST /user/login → Logs in a user.	• Content-Type: application/json	• org : Organization name. • username : User username.	N/A

Endpoint	Required headers	Required payload fields	Optional payload parameters
	• Content-Type: application/octet-stream • Authorization: token	• signature : Signature of the challenge using the private key.	N/A

2.1.2. Authenticated Endpoints

Endpoint	Required headers	Required payload fields	Optional payload parameters
GET /user/list → Returns a list of all users	• Content-Type: application/octet-stream • Authorization: token	N/A	• username : Filter by username.
GET /user/<username>/roles → Returns a list of all roles of a user.	• Authorization: token	N/A	N/A
GET /file/list → Returns a list of all files.	• Content-Type: application/octet-stream • Authorization: token	N/A	• username : Filter by username. • datetime : Filter by datetime. The datetime filter has the following fields: ◦ value : Epoch time in seconds. ◦ relation : ot | eq | nt . (One of the following: older than, equal to, newer than)
POST /user/logout → Logs out a user.	• Authorization: token	N/A	N/A
POST /role/session/assume/<role> → Assumes a role in the session.	• Authorization: token	N/A	N/A
POST /role/session/drop/<role> → Drops a role from the session.	• Authorization: token	N/A	N/A

Endpoint	Required headers	Required payload fields	Optional payload parameters
GET /role/session/list → Lists the roles for the session.	• Authorization: token	N/A	N/A
GET /role/<role>/list/users → Lists the users for a role.	• Authorization: token	N/A	N/A
GET /role/<role>/list/perms → Lists the permissions for a role.	• Authorization: token	N/A	N/A
GET /role/perm/<perm>/roles : → Lists the roles with a permission.	• Authorization: token	N/A	N/A

2.1.3. Authorized Endpoints

Endpoint	Required headers	Required payload fields	Required permission
POST /user/create → Creates a new user.	• Content-Type: application/octet-stream • Authorization: token	• username : User's username. • name : User's name. • email : User's email. • public_key : User's public key.	SUBJECT_NEW
POST /user/<username>/suspend → Suspends a user.	• Authorization: token	N/A	SUBJECT_DOWN
POST /user/<username>/activate → Activates a user.	Authorization: token	N/A	SUBJECT_UP
POST /file/upload/metadata → Uploads a file's metadata.	• Content-Type: application/octet-stream • Authorization: token	• document_name : Document name. • key : Document key. • alg : Document algorithm.	DOC_NEW
POST /file/upload/content → Uploads a file's content.	• Authorization: token • Content-Type: multipart/form-data	• file's content as request data	DOC_NEW

Endpoint	Required headers	Required payload fields	Required permission
GET /file/get/<document_handle>/metadata → Downloads a file's metadata.	• Authorization: token	N/A	DOC_READ
POST /file/delete/<document_handle> → Deletes a file.	• Authorization: token	N/A	DOC_DELETE
POST /file/acl → Updates the ACL of a file.	• Content-Type: application/octet-stream • Authorization: token	• document_handle : Document handle. • role : Role name. • perm : Permission name. • operation : add | remove . (One of the following: add, remove)	DOC_ACL
POST/role/create → Creates a new role.	• Content-Type: application/octet-stream • Authorization: token	• role : Role name.	ROLE_NEW
POST /role/<role>/suspend → Suspends a role.	• Authorization: token	N/A	ROLE_DOWN
POST /role/<role>/activate → Activates a role	• Authorization: token	N/A	ROLE_UP
POST /role/<role>/user/add/<username> → Adds a user to a role.	• Authorization: token	N/A	ROLE_MOD
POST /role/<role>/user/remove/<username> → Removes a user from a role.	• Authorization: token	N/A	ROLE_MOD
POST /role/<role>/perm/add/<perm> → Adds a permission to a role.	• Authorization: token	N/A	ROLE_MOD
POST /role/<role>/perm/remove/<perm> → Removes a permission from a role.	• Authorization: token	N/A	ROLE_MOD

2.2. Client Interaction

For the client, each command is executed via terminal, and it is used multiple tools with specific functionalities:

- argparse [1] → Check for errors in arguments given by the user.
- logging [2] → Logging system to send out messages such as errors.
- os [3] → Add path to local folder `~/.sio` to save or load any files used by the current command.
- requests [4] → Main library to allow communication from the client to the API.
- JSON [5] → Main library to store and exchange data between the client and API.

For every command the argument `-r` is present to set the APIs address. It is needed to define if it wasn't previously, otherwise an error is cast with the corresponding message.

2.2.1. Generating the Subject's Credentials

To use the API, it is first needed to create a public key to create an organization with the key. The command `rep_subject_credentials` generates a key-pair using RSA with a given password and saves both public and private keys different files under the folder mentioned before.

2.2.2. Creating an Organization

The command `rep_create_org` creates an organization. In order to do that, the client must give the file containing his public key in order to create a session afterwards.

2.2.3. Creating a session

For the client to use the Authenticated API, the command `rep_create_session` allows the user to create a session and assume an identity. This command also protects information that shouldn't be visible to outsiders when it's being transferred between the client and the server.

To protect the information, the client and the server initiate a Diffie-Hellman [6] key exchange where both create a key pair with the same parameters and share each other their public key to derive with their own private key and obtain a common key which can be used to encrypt and decrypt information between both entities.

PYTHON

```
# generate the parameters and the key pair
generator = 2; key_size = 1024
parameters = generate_parameters(generator, key_size)
private_key, public_key = generate_key_pair(parameters)

# send the parameters and the public key to the server
req = requests.post('json with parameters and public key...')

# receive the server's public key
response = req.get('json with server public key...')
server_public_key = serialization.load_pem_public_key(bytes.fromhex(response['public_key']))

# obtain the derived key
derived_key = derive_keys(private_key, server_public_key)
```

If the exchange is successful, the client will attempt to log in using its private key that should be given when executing this command. This process is explained in the decision chapter, under Authentication, but in short, the server will send a challenge to the client, which the client will sign with its private key (from a different key pair from Diffie-Hellman) and send back to the server. If the server can verify the signature, the client is authenticated and a session is created. This is the same method used in SSH.

2.2.4. Listing Organizations

The command `rep_list_org` lists all organizations present in the server. Since it requires no authentication, it is a simple command that sends a GET request to the API and prints the response.

However, it is important to note that the response is encrypted and must be decrypted before being printed. In order to achieve this, the client uses a symmetric encryption algorithm that uses an Initialization Vector (IV) to decrypt the response. This is done by the function `decrypt_request_with_iv` that receives the encrypted response, where it separates the IV from the encrypted text (first 16 bytes being the IV) and returns the decrypted response.

PYTHON

```
def decrypt_request_with_iv(response):
    iv = response[:16]
    cipher = response[16:]

    # decrypt the response (simplified)
    plaintext = decrypt(cipher, iv)

    # need to decode the bytes to string
    return plaintext.decode()
```

It could be argued that if a user is authenticated, the response could be encrypted/decrypted using the derived key from the Diffie-Hellman key exchange, but since the information is publicly available, it's not really needed.

2.2.5. Listing Users of an Organization

The command `rep_list_users` lists all users of an organization. It requires the user to be authenticated and the response is encrypted with the derived key from the Diffie-Hellman key exchange. Optionally, the user can query for a specific user by providing the username as an argument.

First, the API checks if the user is authenticated (has a session file) and the username was given as an argument. Then, it sends a `GET` request to the server. The data of the request is encrypted with the derived key, created with the initialization of the session.

Finally, if the request was successful, the data is decrypted and printed to the screen.

2.2.6. Assuming a role

With the command `rep_assume_role`, the user can assume a role in the session. This command requires the user to be authenticated and the role to be assumed. The role is sent to the server in a `POST` request, where the role name is used as a parameter in the URL. This way, there is no need to send anything in the body of the request, therefore there is no need to encrypt the data. The server will check if the user has the permission to assume the role and if the role exists.

If everything is correct, the server will return a 200 status code, otherwise it will return an error message. The API will print the message to the screen and exit with the corresponding status code.

2.2.7. Adding a role to the organization

The command `rep_add_role` allows the user to add a role to the organization. This command requires the user to be authenticated and the role to be added. The role is sent to the server in a `POST` request, where the role name is sent in the body of the request. The data is encrypted with the derived key from the Diffie-Hellman key exchange.

If the request is successful, the server will return a 201 status code, otherwise it will return an error message. The API will print the message to the screen and exit with the corresponding status code, as per usual.

2.2.8. Adding permissions/user to a role

Given a role name and a valid permission/username, the `rep_add_permission` command allows a user to add certain permissions to a specific role in an organization (if the user has permissions to do that) or a username to that role.

First, since there's only some permissions, it's checked if it corresponds to one of them and if not, it assumes it's a username.

It then sends a `POST` request to the server, with the role and permission/username on the URL. With this of course, there is no need for encryption. It also sends the session token (which could be encrypted as mentioned in the analysis) in the header.

Like the other commands, if the request is successful, the server will return a 201 status code, otherwise it will return an error message. The API will print the message to the screen and exit with the corresponding status code.

2.2.9. Removing a permission/user from a role

Similar to the previous command, the `rep_remove_permission` command allows a user to remove certain permissions from a specific role in an organization (if the user has permissions to do that) or a username from that role.

It follows the same steps as the `rep_add_permission` command, but instead of adding, it removes the permission/username from the role.

2.2.10. Suspending a role

The command `rep_suspend_role` allows the user to suspend a role in the organization. This command requires the user to be authenticated and the role to be suspended. The role is sent to the server in a `POST` request, where the role name is used as a parameter in the URL. This way, there is no need to send anything in the body of the request, therefore there is no need to encrypt the data. The server will check if the user has the permission to suspend the role.

The session token is also sent in the header, so that the server knows who is attempting to access the endpoint.

An error can occur if the role provided oes not exist (since the endpoint doesn't exist, it will return a "Failed to obtain response from server." error).

If everything is correct, the server will return a 200 status code, otherwise it will return an error message. The API will print the message to the screen and exit with the corresponding status code.

2.2.11. Reactivating a role

The command `rep_reactivate_role` allows the user to reactivate a role in the organization. This command requires the user to be authenticated and the role to be reactivated. The role is sent to the server in a `POST` request, where the role name is used as a parameter in the URL. This way, there is no need to send anything in the body of the request, therefore there is no need to encrypt the data. The server will also check if the user has the permission to reactivate the role.

This command is similar to `rep_suspend_role` , but instead of suspending a role, it reactivates a suspended role.

2.2.12. Dropping a role

This command (rep_drop_role) is, once again, similar to the previous commands in terms of how it works. The only difference being that it makes the current user drop the provided role.

2.2.13. Listing the existing roles

The command rep_list_roles requires a session token in order to function. Given that the token is present, it sends a simple GET request to the server, where it returns the existing roles in the current organization.

The returning content is encrypted using the Diffie-Hellman derived key, calculated when the session was created.

Like before, the roles are printed to the screen, and any errors would also be printed, with the corresponding exit codes properly returned.

2.2.14. Listing subjects with a role

The following command rep_list_role_subjects is similar to the previous command (rep_list_roles), except this command will return the subjects that have a specific role. This role is given as an argument and should be provided beforehand when executing the command.

2.2.15. Listing roles of a subject

This command (rep_list_subject_roles) is very similar to the rep_list_role_subjects command, but with one key difference. This command is meant to list the roles of a subject. The implementation of the command is, as mentioned before, similar to the role subjects command.

2.2.16. Listing a role's permissions

Another command with a similar implementation of the previous ones, the rep_list_roles returns a list of the permissions that someone with that role would have on the organization.

2.2.17. Listing roles with a permission

Once again, this command has an almost identical implementation of the previous commands, the command rep_list_permission_roles gives the user a list of roles that have a specific permission, given as an argument before calling said command.

2.2.18. Adding a subject to an organization

The command rep_add_subject aims to give the possibility of a user to be able to add another user to the current organization. This is, of course, a command that requires specific permissions in order to do this.

First, it checks for the existence of a session file. Then, it sends a POST request to the server, with the information of the subject to be added. These include the username , full_name , email and public_key . This content is encrypted with the derived key from the Diffie-Hellman key exchange from the user that is executing the command.

According to the success of the previous request, a message will be printed back to the user.

2.2.19. Suspending a subject

This command (rep_suspend_subject) allows for a user (authenticated and with the required permission) to suspend a subject from an organization.

The username is parsed in the URL, and the session token is, as usual, sent in the header of the request.

2.2.20. Activating a subject

The following command `rep_activate_subject` has a similar implementation as the `rep_suspend_subject` command. The key difference is that instead of activating a subject, it suspends a subject. In order to activate a subject, the subject must be in a suspended state.

2.2.21. Adding a document

Using the command `rep_add_doc` allows the user to upload a document to the current organization his in. The file given is encrypted with symmetric encryption and the key, the document's name and the algorith used is upload as it's metadata to the server.

PYTHON

```
#Encrypt content
key, content = encrypt_file(BASE_DIR + args.file, BASE_DIR + 'encryptedText')

metadata = {'document_name' : args.name, 'key' : key.hex(), 'alg' : 'AES-CFB' }
```

Of course, the information is then encrypted with the derived key located in the session file before sending to the server with a `POST` request.

PYTHON

```
# Encrypting metadata
derived_key = bytes.fromhex(args.session['derived_key'])
metadata = encrypt(metadata, derived_key).hex()

headers = {
    'Authorization': args.session['token'],
    'Content-Type': 'application/octet-stream'
}
req = requests.post(f'http://{state['REP_ADDRESS']}/file/upload/metadata',
                    data=metadata,
                    headers=headers)
```

If the upload is successful, the client proceeds to upload the encrypted content, along with a hash of the content for the server to check for integrity.

PYTHON

```
#Upload Document content
file = {'file' : (BASE_DIR + args.file, content)}

req = requests.post(f'http://{state['REP_ADDRESS']}/file/upload/content',
                    files=file,
                    headers={'Authorization': args.session['token'],
                    'File-Checksum' : digest.get_hash(content)})
```

2.2.22. Modifying Access Control List(ACL) from a document

In order to modify the ACL of an organization, the user must have the permission `DOC_ACL` . Once that permission is present, the command `rep_acl_doc` can be used to change specific role's permissions such as

- `DOC_ACL` → Allows the person holding the given role to change the ACL.
- `DOC_READ` → Allows the person holding the given role to read the document.
- `DOC_DELETE` → Allows the person holding the given role to delete the **file_handle**.

2.2.23. Getting a document's metadata

Every document has it's metadata that includes the `file handle` and `key`. To get the metadata of a specific document, the command `rep_get_doc_metadata` sends a `GET` request and with the derived key the response is decrypted. Using the `file handle` we can then get the file.

2.2.24. Getting a file's content

This command (`rep_get_file`) is relatively simple, the client provides the file handle to the server, and it is given the file's content which is either written on the screen or saved in a file given by the user.

2.2.25. Decrypting a file

Given the document's metadata and the file's content, what's left to do is to get the actual content by decrypting it. As such, using the command `rep_decrypt_file` the file's content is decrypted using the key given in its metadata

PYTHON

```
content = symmetric_encryption.decrypt_file(bytes.fromhex(metadata['key']), BASE_DIR + args.encrypted)
```

2.2.26. Getting a document's file

The command `rep_get_doc_file` is essentially the union of the last three commands. The document's name given by the user is hashed and a `GET` request is sent to the server to acquire the metadata.

PYTHON

```
doc_name = digest.get_hash(bytes(args.name, encoding='utf-8'))
metadata = requests.get(f'http://{state['REP_ADDRESS']}/file/get/' + doc_name + '/metadata', headers={'Authorization': args.session['token']})
```

The metadata is then decrypted, and we get the `file handle` needed to get it's content by sending another `GET` request. If successful, the hash of the file's content must be equal to the file handle, this way we see if the file's integrity is maintained.

PYTHON

```
file = requests.get(f'http://{state['REP_ADDRESS']}/file/get/' + metadata['file_handle'] + '/content')
file = file.content
if not digest.get_hash(file) == metadata['file_handle']:
    logger.error("File's integrity was lost.")
    sys.exit(-1)
```

Once having both the metadata and the file's content, the client decrypts it like it's being done in the command `rep_decrypt_file`. The file's content is written to a temporary file, and it's used symmetric decryption to get the plaintext using the key stored in the metadata.

PYTHON

```
with open(BASE_DIR + 'encrypted_file', 'wb') as f:
    f.write(file)

content = symmetric_encryption.decrypt_file(bytes.fromhex(metadata['key']), BASE_DIR + 'encrypted_file')
os.remove(BASE_DIR + 'encrypted_file')
```

2.2.27. Listing documents

The command `rep_list_docs` allows the user to get a list of all documents in the organization.

To make it organized, the command can be executed with a date and specify if the documents must be older, newer or created in that specific date, or it can be filtered by who created the document.

The payload simply contains nothing if there's no subject or date given, or it contains the username, date or both.

The following example shows how the endpoint may be called specifying the date and the username.

PYTHON

```python
payload = {} # Payload may be sent empty if no filters are needed

# If the username is given
payload['username'] = args.username[0]
# If the date is given
payload['datetime'] = {'value': args.date[0], 'relation': args.date[1]}

payload = json.dumps(payload)
payload = encrypt(payload, derived_key).hex()

headers = {
  'Authorization': args.session['token'],
  'Content-Type': 'application/octet-stream'
}
req = requests.get(f'http://{state['REP_ADDRESS']}/file/list', data=payload, headers=headers)
```

3. Decisions

In this topic, it'll be presented and discussed the decisions made by the authors of the project thorough its development. These can range from choices taken to be used for encryption to the implementation of the API itself.

3.1. Usage of the Diffie-Hellman Key Exchange

The Diffie-Hellman key exchange is used in this project to securely create and maintain a key where both ends (client and server) know that key, but never exchange it. Diffie-Hellman is great for this, since by exchanging the parameters used to generate the key, and each other's public key, both can end up with the same key, without ever needing to exchange it on a public channel.

The key generation (public and private) is done using the parameters agreed on both ends (using the `dh` package from `cryptography`).

3.2. Symmetric Encryption

The symmetric encryption used in this project is AES-CFB. This is a symmetric encryption algorithm that is widely used and considered secure. This cipher was chosen due to the FeedBack functionality, that avoids the same plaintext to be encrypted to the same ciphertext, which is a vulnerability present in the ECB mode.

This type of encryption is used to encrypt anonymous requests and files. Both use an Initialization Vector (IV) of 16 bytes, with it being randomly generated.

3.3. Authentication

When it comes to authentication, there were multiple options to create a secure and reliable system. The chosen method was the same as the one used by ssh, which uses a key pair to authenticate the user.

The key pair is generated using the RSA algorithm, with a key size of 2048 bits. The public key is stored on the server, associated to the user, and the private key is kept privately by the user.

During the login, the client sends a login request with the username and organization, to which the server generates and returns a challenge (a string of 256 random characters). The client then signs this challenge with its private key and sends the signature back to the server. The server then verifies the signature using the public key associated with the user and, if the signature is valid, the user is authenticated and thus the session is validated.

3.4. File Handling

As mentioned before, the files are encrypted symmetrically. The key used to decrypt them is within its metadata, which can only be accessed with the required permissions. The encryption and decryption is done by blocks (or chunks) of 2^{11} bytes.

3.5. Session-Related Content

By default, all keys, files and session files are stored under `~/.sio` on the client side. In order to clear all files generated by the application, the user must run `rm -r ~/.sio/`

3.6. Hashing

For the hashing, it was used the SHA256 hashing mechanism, since it produces a 256 byte value. This is used to check the file integrity when going from Client → Server and vice versa.

3.7. Database

The authors of this project have chosen to use a database to store all server-side content (except for the files' content).

The database used in this project is SQLite3. This was chosen due to the authors' familiarity with it, its simplicity and the fact that it is a single file, which makes it easier to manage and distribute. SQLAlchemy was also used to interact with the database, as it provides a more abstract way to interact with the database and to have the ability to have the dataclasses corresponding to the tables. When needed, it can be reset using an endpoint mentioned in the Features chapter.

3.8. Modularized Server

The server is modularized in order to make it easier to maintain and expand.

For this, Flask's Blueprints were used. This allows for the server to be divided into multiple modules, each with its own set of endpoints.

Each service has its own file and class, which makes it easier to understand and maintain. This also makes it easier to add new services to the server, as they can be added as new files and classes. This also allows for the endpoints to be easily changed as there are no real operations being done on them, but instead they are just calling the respective service.

3.9. Roles

The roles are used to define the permissions of each user. The permissions are stored, viewed and treated as seen in other services, like Discord. This approach is as follows:

PYTHON

```
class Perm(Enum):
    DOC_ACL = 0b000000000001
    DOC_READ = 0b000000000010
    DOC_DELETE = 0b000000000100
    #...
```

Since it is stored in bits, validating a permission or adding it to a role is very easy, since it'll be just bit-wise operations.

3.9.1. Checking a permission

To check if a role has a permission, it can be easily done by looking at the bit corresponding to the permission, and do a simple AND operation with said bit (that bit has to be 1). For this, the following function was created:

PYTHON

```
def check_perm(bit_array: int, perm_to_check: int) -> bool:
    return bit_array & perm_to_check == perm_to_check
```

3.9.2. Changing a permission

To change a permission associated with a role, all that is needed is a OR operator (to add) or a AND operator (to remove) with the current role's permissions and the bit we want to enable (permission to give). This bit has to be also 1. For this, the following function was created, returning the resulting bit array:

PYTHON

```
def calc(bit_array: int, perm: Perm, operation: PermOperation) -> int:
    if operation == PermOperation.ADD:
        return bit_array | perm.value
    return bit_array & ~perm.value
```

4. ASVS Analysis

For the analysis section, the project will be evaluated under the scope of the V3 (Session Management) chapter of the OWASP ASVS, using version v4.0.3. This will include an assessment of the session management mechanisms implemented, as well as any vulnerabilities identified and possible mitigations.

4.1. V3 Session Management

4.1.1. Fundamental Session Management Security

Requirement	Description	Applicable	Implemented
3.1.1	Verify the application never reveals session tokens in URL parameters.	✔	✔

3.1.1

The current implementation meets the requirement, as the session tokens are not exposed in the URL parameters but instead are sent in the Authorization header.

This way, instead of parsing the URL for the token, the server can directly access the token from the header, which is a more secure method of handling session tokens. This is done with the following line of code (including the necessary error handling):

PYTHON

```
token = request.headers.get("Authorization")
if not session_token:
    return jsonify({"error": "No session token"}), 400
```

This piece of code is present in all endpoints that require a session token, ensuring that the token is always sent in the header and never in the URL.

4.1.2. Session Binding

Requirement	Description	Applicable	Implemented
3.2.1	Verify the application generates a new session token on user authentication.	✔	✔
3.2.2	Verify that session tokens possess at least 64 bits of entropy.	✔	✔
3.2.3	Verify the application only stores session tokens in the browser using secure methods such as appropriately secured cookies (see section 3.4) or HTML 5 session storage.	✗	✗
3.2.4	Verify that session tokens are generated using approved cryptographic algorithms.	✔	✔

3.2.1, 3.2.2, 3.2.4

The application generates a new session token on session creation when a user logs in.

This token is generated using the `secrets.token_hex(128)` function, which generates a 256-character hexadecimal string, providing more than 64 bits of entropy. This function has been certified as secure by OWASP in their cheat sheet series [7].

This generation is implemented in the code as follows:

PYTHON

```python
def create_session(user: User, org: Organization) -> Session:
    session = Session(
        user_id=user.id,
        org_id=org.id,
        token=secrets.token_hex(128), # 256-character hexadecimal string
        roles=[],
        challenge=secrets.token_hex(128),
        verified=False
    )
    db.add(session)
    db.commit()
    db.refresh(session)
    return session
```

3.2.3

This requirement is not applicable to the current implementation, as there is no browser involved with the application and therefore not used to store any session tokens.

4.1.3. Session Termination

Requirement	Description	Applicable	Implemented
3.3.1	Verify that logout and expiration invalidate the session token, such that the back button or a downstream relying party does not resume an authenticated session, including across relying parties.	✔	✔
3.3.2	If authenticators permit users to remain logged in, verify that re-authentication occurs periodically both when actively used or after an idle period.	✔	✗
3.3.3	Verify that the application gives the option to terminate all other active sessions after a successful password change (including change via password reset/recovery), and that this is effective across the application, federated login (if present), and any relying parties.	✗	✗
3.3.4	Verify that users are able to view and (having re-entered login credentials) log out of any or all currently active sessions and devices.	✔	✗

3.3.1

Upon logout, or any session deletion, the session data is completely removed from the database also deleting the session token, thus invalidating the session.

This is implemented in the code as follows, and accessed through the `POST /user/logout` endpoint:

PYTHON

```
def delete_session(session: Session) -> None:
    db.delete(session)
    db.commit()
```

3.3.2

The application does not currently implement re-authentication after a period of inactivity.

This could be implemented by storing, for each session, the last time it was used, and checking this timestamp against the current time when a request is made. If the time difference exceeds the threshold defined in ASVS (12 hours or 15 minutes of inactivity, with 2FA), the user would be required to re-authenticate. This could be implemented in the code as follows:

PYTHON

```
def check_session_timeout(session: Session) -> bool:
    return (datetime.now() - session.last_used).total_seconds() > SESSION_TIMEOUT
```

3.3.3

This requirement is not applicable to the current implementation, as the application doesn't use password for logins, and therefore doesn't have a password change mechanism and thus not having the mechanism to terminate all other active sessions after a successful password change.

3.3.4

Currently, there isn't a mechanism to view and log out of active sessions and devices.

This could be implemented by storing the device information in the session data and enabling an endpoint for the user to view all active sessions and devices, and then revoke access to them. This endpoint could be implemented as follows:

PYTHON

```
@user_bp.route("/sessions", methods=["GET"])
def list_sessions():
    session_token = request.headers.get("Authorization")
    if not session_token:
        return jsonify({"error": "No session token"}), 400

    try:
        session = SessionService.validate_session(session_token)
    except SessionException as e:
        return jsonify({"error": e.message}), e.code

    user = UserService.get_user(session.user_id)
    if not user:
        return jsonify({"error": "User not found"}), 404

    sessions = SessionService.get_user_sessions(user)
    return jsonify({"sessions": sessions}), 200
```

This would return a list of all active sessions for the user, and then the user could choose to revoke access to any of them, using the session id (not the token) through the following endpoint:

PYTHON

```
@user_bp.route("/logout/<session_id>", methods=["POST"])
def user_logout_session(session_id):
    session_token = request.headers.get("Authorization")
    if not session_token:
        return jsonify({"error": "No session token"}), 400

    current_session = SessionService.get_session(session_token)
    if not current_session:
        return jsonify({"error": "Not authenticated"}), 401

    session = SessionService.get_session_by_id(session_id)
    if not session:
        return jsonify({"error": "Session not found"}), 404

    if session.user_id != current_session.user_id:
        return jsonify({"error": "Unauthorized"}), 403

    SessionService.delete_session(session)
    return jsonify({"message": f"Logged out from session with id {session_id}"}), 200
```

4.1.4. Cookie-based Session Management

Requirement	Description	Applicable	Implemented
3.4.1	Verify that cookie-based session tokens have the 'Secure' attribute set.	✗	✗
3.4.2	Verify that cookie-based session tokens have the 'HttpOnly' attribute set.	✗	✗
3.4.3	Verify that cookie-based session tokens utilize the 'SameSite' attribute to limit exposure to cross-site request forgery attacks.	✗	✗
3.4.4	Verify that cookie-based session tokens use the "__Host-" prefix so cookies are only sent to the host that initially set the cookie.	✗	✗
3.4.5	Verify that if the application is published under a domain name with other applications that set or use session cookies that might disclose the session cookies, set the path attribute in cookie-based session tokens using the most precise path possible.	✗	✗

3.4.1, 3.4.2, 3.4.3, 3.4.4, 3.4.5

None of the requirements are applicable to the current implementation, as the application does not use cookies to store any data or to manage sessions.

4.1.5. Token-based Session Management

Requirement	Description	Applicable	Implemented
3.5.1	Verify the application allows users to revoke OAuth tokens that form trust relationships with linked applications.	✗	✗
3.5.2	Verify the application uses session tokens rather than static API secrets and keys, except with legacy implementations.	✔	✔

Requirement	Description	Applicable	Implemented
3.5.3	Verify that stateless session tokens use digital signatures, encryption, and other countermeasures to protect against tampering, enveloping, replay, null cipher, and key substitution attacks.	✔	✗

3.5.1

This requirement is not applicable to the current implementation, as there is no OAuth service implemented in the application.

3.5.2

The application uses that exact implementation, using session tokens instead of static API secrets and keys to manage sessions as mentioned in the section 3.2 of the ASVS.

The server generates a session token upon login, using the function `secrets.token_hex(128)`, and sends it to the client. The client then sends the token in the Authorization header in every request that requires authentication.

3.5.3

Since this topic involves a few different types of attacks, it's best to address them individually:

Digital Signatures and Replay

These attacks could be tackled with the use of JWT (JSON Web Tokens). They support a `exp` field, that indicate the expiry date of said packet (protects against Replay attacks). Expiry time could be adjusted dynamically according to client-server ping.

JWT also supports signatures, so with a given secret key, the server can verify the integrity of the token. The signature is calculated using the header and payload of the token (with the content of the token being base64 encoded).

Tampering

With the support of signatures, the server can verify the integrity of the token, and if it was tampered with, the signature would not match the content of the token.

Another added layer of security could be the use of a counter, that is incremented with each request, where the server verifies if the counter is correct, and vice versa.

Null Cipher

The cipher python package used in the project, `cryptohazmat`, uses AES-CFB, that avoids null ciphers. Of course, any failure within the package (or any other package) due to an update or a bug could eventually lead to a null cipher attack.

Key Substitution

Since it is being used a Diffie-Hellman key exchange, the key is never sent over the network, the server and client end up with the same key, so there is no key to be substituted. It is also not possible to substitute a key, that would require re-authentication.

Encryption

The current implementation does not encrypt the session token, which could be a vulnerability. This could be implemented by encrypting the session token with a symmetric key (and use a strong algorithm, like AES-256), and then decrypting it on the server side. This would protect the token from being read by an attacker, even if they manage to intercept it.

4.1.6. Federated Re-authentication

Requirement	Description	Applicable	Implemented
3.6.1	Verify that Relying Parties (RPs) specify the maximum authentication time to Credential Service Providers (CSPs) and that CSPs re-authenticate the user if they haven't used a session within that period.	✗	✗
3.6.2	Verify that Credential Service Providers (CSPs) inform Relying Parties (RPs) of the last authentication event, to allow RPs to determine if they need to re-authenticate the user.	✗	✗

3.6.1, 3.6.2

These requirements are not applicable to the current implementation, as the application does not have a federated authentication system.

4.1.7. Defenses Against Session Management Exploits

Requirement	Description	Applicable	Implemented
3.7.1	Verify the application ensures a full, valid login session or requires re-authentication or secondary verification before allowing any sensitive transactions or account modifications.	✔	✗

3.7.1

Currently, the application does not require re-authentication or secondary verification before allowing sensitive transactions or account modifications, it just checks if the user is authenticated and has the required permissions.

This could be implemented by adding a challenge signature to the request using the rsa key pair, which would be verified by the server before allowing the transaction. This is the same mechanism already used for the login endpoint. To implement this, every endpoint that involves sensitive transactions or account modifications would need to be updated to include the challenge generation and signature verification.

5. Conclusions

The SIO-2425 project successfully demonstrates the practical application of critical security principles, including authentication, access control, session management, and cryptography. Through the implementation of modularized server architecture, robust session handling, and encrypted communication mechanisms, the project adheres to some industry standards, where analyzing the OWASP ASVS gives the developers a broader landscape of what are the best practices, and what needs to be done in order to achieve a more secure application, even though of course it'll never be fully secure.

Despite these achievements, the analysis highlighted areas requiring further improvement, such as enhanced mechanisms for session re-authentication, secure session management features, and the ability to terminate active sessions after sensitive changes. Addressing these issues would further improve the system's security posture and resilience against potential vulnerabilities.

The methodologies and decisions applied throughout this project underline the importance of secure design in software development. By integrating tools like Diffie-Hellman key exchange, AES encryption, and SHA256 hashing, the project ensures data confidentiality, integrity, and authenticity. It was also a great learning opportunity for the authors, when it comes to server-side development and design, as well as the importance of secure coding practices.

Future work could focus on refining the system to meet additional ASVS requirements and expanding its usability in real-world applications. Overall, this project stands as a testament to the successful implementation of secure application principles and the importance of continuous learning and iteration in cybersecurity practices.

---

1. https://docs.python.org/3/library/argparse.html
2. https://docs.python.org/3/library/logging.html
3. https://docs.python.org/3/library/os.html
4. https://requests.readthedocs.io/en/latest/
5. https://docs.python.org/3/library/json.html
6. https://cryptography.io/en/latest/hazmat/primitives/asymmetric/dh/
7. https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html#secure-random-number-generation