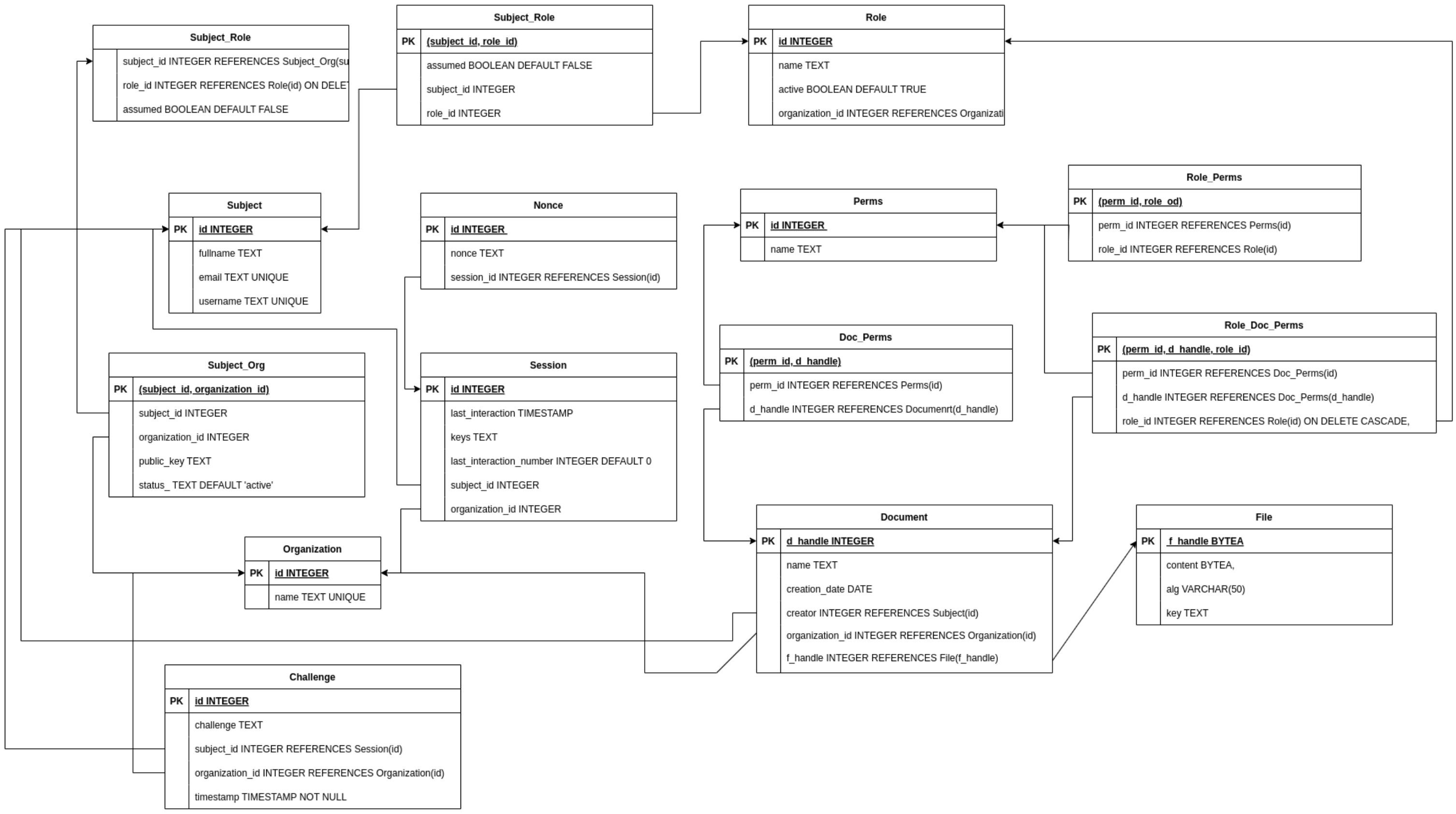
Afonso Ferreira - 113480
Ricardo Antunes - 115243
Tomás Brás - 112665

Índice

Database developed

We developed an SQLite database to save all the data.



The schema of the database is saved in *schema.sql* and the data is stored in *database.db*

Commands implemented

We implemented several commands that can be split into 4 categories:

- Local commands
- Commands that use the anonymous API
- Commands that use the authenticated API
- Commands that use the authorized API

All of these commands can be used running the script ./command_name <arguments>. The script is interpreted as a python file by using `#!/usr/bin/env python3` and then each file calls a function in client.py which will deal with the communication with the repository (when necessary)

Local Commands

- **rep_subject_credentials <password> <credentials_file>**

We use this command to generate an ECC pair of keys that will be stored in two different files (key.pem and key_public.pem). It's important to note that the user when using this command has to necessarily specify that the key will be saved in a .pem file or it won't work.

- **rep_decrypt_file <encrypted file> <encryption metadata>**

This command decrypts a file that has been encrypted using the AES-GCM encryption algorithm. It uses a .json file that contains the metadata that provides the necessary decryption information, including the encryption parameters and a password for key derivation.


```
{
    "f_handle": "gaEDMw1owTdpJ1/JdZWozngVrV5tk7krNxQDDI64rQM=",
    "alg": "SHA256|AES-GCM|bf1c6b42ca2fe6c0f3e2cada770c53d2|4fc809196b5572cf47758837|09c1a1421f779e5b6396424dde2db5be",
    "password": "8993e8b733f3a8730e5caf3ff364c5ca7a9b50be8955789fb02006916a85f27e"
}
```


Example of the json file used containing the metadata information

Commands that use the anonymous API

- **rep_create_org <organization> <username> <name> <email> <public key file>**

We use this command to create a new organization and then make the specified user the manager of that organization. The organization name and username cannot be usernames that have existed before (but the public key file can be duplicated from other subjects)

- **rep_list_orgs**

List the organizations that are defined in the repository.

- **rep_create_session <organization> <username> <password> <credentials file> <session file>**

Used to create a session for the user in the respective organization, first realizing an authentication process and then if authentication was successful creating a session file. The credentials file is the user's private key and the password is the one used by the subject when creating his credentials, and it is used to access the private key

- **rep_get_file <file handle> [file]**

We use this to get a file given its handle. The file contents are written to the console or written to a file (if specified).

Commands that use the authenticated API

- **rep_assume_role <session file> <role>**

Subject tries to assume a role which verifies if it belongs to the subject and if it's not suspended.

- **rep_drop_role <session file> <role>**

Subject tries to drop a role which was previously assumed by the subject.

- **rep_list_roles <session file> <role>**

This command lists the current roles of a session.

- **rep_list_subjects <session file> [username]**

Lists the information of all the subjects (or a specific user if specified) belonging to the organization on the current session.

- **rep_list_role_subjects <session file> <role>**

Lists the subjects that have the specified role in the current session's organization.

- **rep_list_subject_roles <session file> <username>**

Lists the roles of a specific user of the organization with which I have currently a session.

- **rep_list_role_permissions <session file> <role>**

Lists the permission of a specific role in the current session's organization.

- **rep_list_permission_roles <session file> <permission>**

Lists the roles in the current session's organizations which contain a specific permission.

- **rep_list_docs <session file> [-s username] [-d nt/ot/et date]**

Lists documents in an organization. We can filter documents by the user that created it and the date it was created.

Commands that use the authorized API

- **rep_add_subject <session file> <username> <name> <email> <credentials file>**

Adds or updates a user in an organization after validating the session, permissions, and data integrity. Security is ensured through mechanisms such as signatures, non-repetition (nonce), and sequence validation.

- **rep_suspend_subject <session file> <username>**

Suspends a user in an organization. It validates the session, checks permissions, ensures the user exists and isn't already suspended, and then updates their status to "suspended." It prevents suspending managers and uses encryption and signed payloads for secure communication.

- **rep_activate_subject <session file> <username>**

Validates authentication, permissions, and the session before reactivating a user in the organization. It updates the user's status to "active" in the database, ensures security with encryption and digital signatures, and robustly handles errors.

- **rep_add_role <session file> <role>**

Adds a role to the organization with which I have currently a session if the role does not already exist in the organization.

- **rep_suspend_role <session file> <role>**

Suspends a role in an organization by validating authentication, ensuring the session is valid and updating the role's status to inactive in the database, except for protected roles like "manager"

- **rep_reactivate_role <session file> <role>**

Allows authorized users to reactivate a suspended role in an organization.It ensures the user has the necessary permissions. The API verifies the existence of the organization and role, checks if the role is suspended, and then updates the role's status to active.

- **rep_add_permission <session file> <role> <username>**

Allows adding a role to a user within an organization. It ensures that both the role and the subject exist within the organization. If the user already has the role assigned, an error is returned. If everything is valid, the role is assigned to the user. This function ensures that only users with the appropriate permissions can assign roles to others, logging all actions and using encryption to protect sensitive data.

- **rep_remove_permission <session file> <role> <username>**

Remove roles from users, ensuring that only authorized subjects can perform these operations.

- **rep_add_permission <session file> <role> <permission>**

Ensures that only authorized users (with a specific permission) can add permissions to roles within an organization.

- **rep_remove_permission <session file> <role> <permission>**

Ensures that only authorized users (with a specific permission) can remove permissions from roles within an organization.

- **rep_add_doc <session file> <document name> <file>**

Adds a document with a given name to the organization with which I have currently a session

- **rep_get_doc_metadata <session file> <document name>**

Retrieve the metadata of a document from the database. This functionality is typically used to fetch details about a document, including encryption metadata, which could be useful for decrypting its contents.

- **rep_get_doc_file <session file> <document name> [file]**

Allows a user with the appropriate permissions to access and retrieve the file content of a document, ensuring that it has not been tampered with. The document is securely decrypted using AES-GCM, and the decrypted content is returned to the user.

- **rep_delete_doc <session file> <document name>**

Clears file_handle in the metadata of a document with a given name on the organization with which I have currently a session in.

- **rep_acl_doc <session file> <document name> [+/-] <role> <permission>**

Adds or removes permissions for a specific document based on the role, while ensuring that only authorized users can modify the ACL. We verify if the document is present in the organization and we also verify if the subject has permissions to access the document.

Features implemented

Masterkey

```
MASTERKEY = b'\xd3\xce\xc9\x91\x12%]\xb9\xbfIc\xf7y\x85b\xb6\xa3o\x1b\xd0\xb2\x01i\x18b\x9e\x00}GM\xebp'
```

The master key is a pre-defined encryption value used to protect the repository from offline attacks. The repository cannot be initialized without first entering a password (masterkey), which is stored, encrypted. The password is encoded into bytes (`attempted_password.encode()`) and then passed through the SHA-256 hash function to generate a digest (a fixed-size output from the hash function). The result is a 32-byte (256-bit) hash.

When someone attempts to initialize the repository, the input is compared with the master key and if they match, the password is considered correct, and the repository is initialized. If they do not match, the password check is repeated by calling the `checkpassword()` function again, prompting the user to enter the password once more.

The master key serves as a reference to ensure that the entered password matches a known, securely stored value. This comparison provides a method for protecting the database's data by allowing only trusted entities to manipulate the repository.

```
def checkpassword():
    attempted_password = input("Insert password for database: ")

    digest = hashes.Hash(hashes.SHA256())
    digest.update(attempted_password.encode())
    d = digest.finalize()

    if d == MASTERKEY:
        return
    checkpassword()
```

Document encryption algorithm

To encrypt the file, we use the AES-GCM algorithm, a technique that guarantees confidentiality, authenticity, and integrity at the same time. We begin by reading the content of the plaintext and generating the two random values that ensure each block of information is secure: the salt and the nonce. The salt is used to derive the encryption key.

We start by deriving the AES encryption key, using the random password and the generated salt.

Next, we use the derived AES key along with the nonce to encrypt the plaintext data. AES-GCM, in addition to providing encryption, also generates an authentication tag, which is used to verify the integrity of the encrypted data. This tag is crucial for ensuring that the data has not been altered.

After the encryption, we generate the file handle. The file handle is a SHA-256 hash of the ciphertext, which serves as a unique identifier for the encrypted content. This handle will later be used to verify the integrity of the file during decryption.

Finally, the encrypted data, along with the nonce and the authentication tag, is stored. The file handle is also saved, ensuring that any changes to the encrypted file can be detected. The encryption process ensures that the data is securely encrypted and that any modification of the data can be detected through the tag and the file handle during decryption.

Document decryption algorithm

To decrypt a file, we use the AES-GCM (Galois/Counter Mode) encryption algorithm to ensure the integrity and authenticity of the data, ensuring that they have not been tampered with.

We begin by loading the metadata, which will help us decrypt the file. This includes the algorithm, the password used to derive the encryption key, and the f_handle (which helps verify the integrity of the file).

The algorithm field contains several pieces of information, which will later need to be split into salt (a random value), nonce, and tag to ensure the authenticity of the data.

Before decrypting the data, we calculate the SHA-256 hash of the content, which, after being converted to base64, is compared with the f_handle from the metadata. If these two values are different, it means the data has been altered.

After verifying the integrity of the data, the key is derived along with the salt, using the PBKDF2 algorithm. The encrypted text, along with the authentication tag, is then decrypted using the derived key, the nonce, and the encrypted content, through AES-GCM.

The AES counter mode is a symmetric encryption/decryption algorithm. As mentioned earlier, the authenticity of the data is verified through the use of the tag, and the decryption of the data is done using CTR, which involves combining the ciphertext with the nonce and counter value. The nonce is unique for each operation and ensures the decryption of different data sets. The counter, along with the key (using AES), is processed, and then an XOR operation is performed with the ciphertext to recover the original data.

The result of this operation is the plaintext file.

This guarantees the authenticity, integrity, and confidentiality of the data.

Protection against replay

Our repository is protected against replay which protects from malicious attacks where a malicious user tries to intercept a valid request or message and replay it to perform unauthorized actions or gain access.

To protect the repository we use a nonce, a unique random value that is used once, that ensures a message is unique and can't be replayed. To verify this we have a verify_nonce function that checks if a nonce has already been utilized. If it has already been utilized the function will return true letting the repository know that a malicious user is trying to replicate a request.

Protection against Eavesdropping

To ensure protection against eavesdropping, we implement a hybrid model that uses symmetric and asymmetric encryption techniques. In symmetric encryption, we use AES-CBC, and for asymmetric encryption, we use RSA.

To encrypt the message content, we use AES. To use CBC mode, we need to generate a random IV to ensure that identical blocks of code produce different results when encrypted. We also need to generate a random AES key and apply padding to the messages to ensure they all have the appropriate length. Using the padded text, the key, and the IV, we calculate the encrypted text. Next, to ensure data integrity, we need to encrypt the symmetric key using the recipient's public RSA key, which is used to encrypt the content, so that we can transmit the information securely. This means that only the recipient, who possesses the corresponding RSA private key, can decrypt the message and access the plaintext.

HMAC is used to ensure that the message has not been altered during transmission. The sender generates the HMAC by encrypting the key with the encrypted content, and the recipient, to check if the text has been altered, calculates the HMAC and verifies if it matches the one sent by the sender.

We use this process whenever we send messages to the repository to ensure their confidentiality. We protect the messages from possible attacks or eavesdropping.

```python
def encrypt(plaintext: str, pubkey: str):
    # Symmetric encryption
    iv = os.urandom(16)
    public_key = serialization.load_pem_public_key(pubkey.encode())

    if isinstance(public_key, ec.EllipticCurvePublicKey):
        # Generate ephemeral EC private key
        ephemeral_private_key = ec.generate_private_key(public_key.curve)
        shared_key = ephemeral_private_key.exchange(ec.ECDH(), public_key)
        derived_key = HKDF(
            algorithm=hashes.SHA256(),
            length=64,  # Derive a longer key to split into encryption and MAC keys
            salt=None,
            info=b"ecdh-encryption"
        ).derive(shared_key)

        # Split the derived key into encryption key and MAC key
        key, mac_key = derived_key[:32], derived_key[32:]

        ephemeral_public_key = ephemeral_private_key.public_key().public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo
        )
        encrypted_key = b""  # Placeholder since we're deriving the key directly
    else:
        # RSA encryption for symmetric key and MAC key
        key = os.urandom(32)
        mac_key = os.urandom(32)
        encrypted_key = public_key.encrypt(
            key,
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None
            )
        )
        ephemeral_public_key = None

    # Encrypt the payload
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
    padder = PKCS7(128).padder()
    padded_data = padder.update(plaintext.encode()) + padder.finalize()
    encryptor = cipher.encryptor()
    encrypted_text = encryptor.update(padded_data) + encryptor.finalize()

    # Compute MAC
    h = hmac.HMAC(mac_key, hashes.SHA256())
    h.update(encrypted_key if encrypted_key else key)
    signature = h.finalize()

    encrypted_data = {
        "encrypted_payload": encrypted_text,
        "encrypted_key": encrypted_key,
        "iv": iv,
        "signature": signature,
        "encrypted_mac_key": mac_key,
    }
```

```python
def decrypt(encrypted_data: dict, privkey: str):
    encrypted_data = decode_binary_data(encrypted_data)
    private_key = serialization.load_pem_private_key(privkey.encode(), password=None)

    if isinstance(private_key, ec.EllipticCurvePrivateKey):
        # Derive the shared key using ephemeral public key
        ephemeral_public_key = serialization.load_pem_public_key(
            encrypted_data["ephemeral_public_key"]
        )
        shared_key = private_key.exchange(ec.ECDH(), ephemeral_public_key)
        derived_key = HKDF(
            algorithm=hashes.SHA256(),
            length=64,  # Same as in encryption
            salt=None,
            info=b"ecdh-encryption"
        ).derive(shared_key)

        # Split the derived key into encryption key and MAC key
        key, mac_key = derived_key[:32], derived_key[32:]
    else:
        # RSA decryption for symmetric key and MAC key
        key = private_key.decrypt(
            encrypted_data["encrypted_key"],
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None
            )
        )
        mac_key = encrypted_data["encrypted_mac_key"]

    # Verify the MAC
    h = hmac.HMAC(mac_key, hashes.SHA256())
    h.update(encrypted_data["encrypted_key"] if encrypted_data["encrypted_key"] else key)
    h.verify(encrypted_data["signature"])

    # Decrypt the payload
    cipher = Cipher(algorithms.AES(key), modes.CBC(encrypted_data["iv"]))
    decryptor = cipher.decryptor()
    decrypted_data = decryptor.update(encrypted_data["encrypted_payload"]) + decryptor.finalize()
    unpadder = PKCS7(128).unpadder()
    unpadded_data = unpadder.update(decrypted_data) + unpadder.finalize()
    return unpadded_data.decode()
```

Protection against manipulation

When protecting against “Eavesdropping” as explained above, we also add a protection against manipulation, by using a key to generate a MAC address which we then encrypt by using the repository / session’s public key to guarantee integrity, since any tampering with the data would result in a mismatch when the MAC is verified, rendering the data invalid and ensuring that unauthorized modifications are detected, without even needing to decrypt the content itself.

Protection against Hijacking

The `verify_sequential_number()` function ensures requests are processed in the correct order by verifying the sequence number sent with the request. It prevents replay and hijacking attacks by checking that each incoming request contains a sequence number greater than the last one recorded for the user’s session. This function should be called after the user's authenticity has been verified but before processing any critical actions that change the system's state.

```python
def verify_sequential_number(subject_id, organization_id, seq_number):
    conn = sqlite3.connect(DATABASE)
    conn.row_factory = sqlite3.Row
    cur = conn.cursor()
    cur.execute(
        "SELECT last_interaction_number FROM Session WHERE subject_id = ? AND organization_id = ?",
        (subject_id, organization_id)
    )
    result = cur.fetchone()
    print(result["last_interaction_number"])
    print(seq_number)
    if result["last_interaction_number"] > seq_number:
        return False
    cur.execute(
        "UPDATE Session SET last_interaction_number = ? WHERE subject_id = ? AND organization_id = ?",
        (seq_number + 1, subject_id, organization_id)
    )
    conn.commit()
    conn.close()
    return True
```

Authentication

In the second iteration we implemented authentication, by adding a signature to the user’s session file which will be used to authenticate the respective user without asking for a password. When the user creates a session using `./rep_create_session` we verify if the user exists in the organization and if it does we get the user’s public key which is saved in the database and create a challenge with it. After that we send the challenge to the subject which will be solved with the subject’s private key. This way only the user with the correct password and private key will be able to use the private key and solve the challenge. The user sends the challenge solved to the repository and if it’s solved correctly, the user will be able to create a session.

When making a request using the Authenticated API the request will be signed using the user’s private key. Upon arriving at the repository, several checks are made, the nonce check, the sequence_number check, the signature in the session file check and additionally the signature from the client is also checked against the subject's public key. This authenticates the message as having come from the subject and not from someone else. In case the content of the message is altered, the signature will no longer be valid, also protecting against manipulation.

Analysis of the software

V2 - Authentication

- **2.1.1** - Verify that user set passwords are at least 12 characters in length (after multiple spaces are combined) - Not fulfilled

How to fix:

In the command `./rep_subject_credentials` we could have implemented a password strengthening verification instead of allowing any kind of password:

```python
def main():
    if len(sys.argv) != 3:
        print("Usage: ./rep_subject_credentials <password> <credentials_file>")
        sys.exit(1)

    password = sys.argv[1]
    credentials_file = sys.argv[2]

    client.subject_credentials(password, credentials_file)
```

Currently we just accept any password and we move on with the credential creation, but we could have implemented a verification to check if the password has 12 characters in length:

```python
def main():
    if len(sys.argv) != 3:
        print("Usage: ./rep_subject_credentials <password> <credentials_file>")
        sys.exit(1)

    password = sys.argv[1]
    if len(sys.argv[1]) < 12:
        print("Password must have at least 12 characters")
        sys.exit(1)

    credentials_file = sys.argv[2]

    client.subject_credentials(password, credentials_file)

if __name__ == "__main__":
    main()
```

This failure could make breaching an account easier, as passwords can become easy to brute-force (such as 1234 passwords), forcing longer passwords would help mitigate this issue (however, as it will be explained later, this issue is largely mitigated by how authentication is done)

- **2.1.2** - Verify that passwords of at least 64 characters are permitted, and that passwords of more than 128 characters are denied - Not fulfilled

How to fix:

We currently permit passwords of any length - we just use the password to access the private key used to authenticate in the repository. Therefore, we only need to limit the amount of characters allowed in our password:

```
def main():
    if len(sys.argv) != 3:
        print("Usage: ./rep_subject_credentials <password> <credentials_file>")
        sys.exit(1)

    password = sys.argv[1]
    if len(sys.argv[1]) < 12:
        print("Password must have at least 12 characters")
        sys.exit(1)
    elif len(sys.argv[1]) > 128:
        print("Password must have at most 128 characters")
        sys.exit(1)

    credentials_file = sys.argv[2]

    client.subject_credentials(password, credentials_file)

if __name__ == "__main__":
    main()
```

- **2.1.3** - Verify that password truncation is not performed. However, consecutive multiple spaces may be replaced by a single space. - Not fulfilled

How to fix:

Currently, we allow passwords to be created in any shape or form. This means that we would only need to detect whenever multiple spaces are used and replace said spaces with a single space.

```
def main():
    if len(sys.argv) != 3:
        print("Usage: ./rep_subject_credentials <password> <credentials_file>")
        sys.exit(1)

    password = sys.argv[1]
    if len(sys.argv[1]) < 12:
        print("Password must have at least 12 characters")
        sys.exit(1)
    elif len(sys.argv[1]) > 128:
        print("Password must have at most 128 characters")
        sys.exit(1)

    password = " ".join(password.split())

    credentials_file = sys.argv[2]

    client.subject_credentials(password, credentials_file)

if __name__ == "__main__":
    main()
```

- **2.1.4** - Verify that any printable Unicode character, including language neutral characters such as spaces and Emojis are permitted in passwords - Fulfilled

Evidence:

```
alof@alof-IdeaPad-5-14IAL7:~/Desktop/SIO/sio-2425-project-115243_113480_112665/delivery2$ ./rep_subject_credentials "💀 😎 😭 🐐" chavel.pem
ECC key pair generated successfully.
Encrypted private key saved to chavel.pem
Public key saved to chavel_public.pem
alof@alof-IdeaPad-5-14IAL7:~/Desktop/SIO/sio-2425-project-115243_113480_112665/delivery2$ ./rep_create_org "MyOrg" "userNew" "User One" "usernew@example.com" "chavel_public.pem"
Response: {'message': 'Organization created successfully'}
alof@alof-IdeaPad-5-14IAL7:~/Desktop/SIO/sio-2425-project-115243_113480_112665/delivery2$ ./rep_create_session "MyOrg" "userNew" "💀 😎 😭 🐐" "chavel.pem" sessao1.json
Session created successfully, session file in sessao1.json
```

- **2.1.5** - Verify users can change their password - Not Fulfilled

How to fix:

We could create a command `./rep_change_credentials <session_file> <password> <credentials_file> <new_password> <new_credentials_file>` that would send a message with the new public key to the repository, making the repository delete all active sessions pertinent to the user in question. The password would need to fulfill the criteria delineated above, and this change could only be made if the user possessed both the old private key, as well as knowing the password, as well as the session file with all the security measures (such as nonce, sequence number, etc) in place.

- **2.1.6** - Verify that password change functionality requires the user's current and new password. - Not fulfilled

How to fix: The implementation above would also cover this point.

- **2.1.7** - Verify that passwords submitted during account registration, login, and password change are checked against a set of breached passwords either locally (such as the top 1,000 or 10,000 most common passwords which match the system's password policy) or using an external API. If using an API a zero knowledge proof or other mechanism should be used to ensure that the plain text password is not sent or used in verifying the breach status of the password. If the password is breached, the application must require the user to set a new non-breached password. - Not fulfilled

How to fix:

We could access the file in this link:

https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/10-million-password-list-top-100000.txt

And each time a credential file is created, we check the created password against this file and we verify if the password matches any of the passwords in that file:

- **2.1.8** - Verify that a password strength meter is provided to help users set a stronger password. - Not fulfilled

How to fix:

Upon creating credential files, output the password strength to the user, so that they can decide to use that password or not (remember that the `./rep_subject_credentials` doesn't interact with the repository.

```python
def main():
    if len(sys.argv) != 3:
        print("Usage: ./rep_subject_credentials <password> <credentials_file>")
        sys.exit(1)

    password = sys.argv[1]
    if len(password) < 12:
        print("Password must have at least 12 characters")
        sys.exit(1)
    elif len(password) > 128:
        print("Password must have at most 128 characters")
        sys.exit(1)

    password = " ".join(password.split())

    with open("common_passwords.txt", "r") as file:
        common_passwords = file.read().splitlines()
        common_passwords = [password for password in common_passwords if 12 < len(password) < 128]

    if password in common_passwords:
        print("Password is too common - please pick a different password")
        sys.exit(1)

    result = check_password_strength(password)
    print(f"Strength: {result['strength']} (Score: {result['score']})")
    if result['feedback']:
        print("Suggestions for improvement:")
        for suggestion in result['feedback']:
            print(f"- {suggestion}")
    if result['strength'] == "Weak":
            print("We strongly advise you choose a stronger password.")
            sys.exit(1)

    credentials_file = sys.argv[2]

    client.subject_credentials(password, credentials_file)

def check_password_strength(password):
    score = 0
    feedback = []

    if len(password) >= 8:
        score += 2
    elif len(password) >= 5:
        score += 1
    else:
        feedback.append("Password is too short (minimum 5 characters).")

    if re.search(r'[A-Z]', password):
        score += 1
    else:
        feedback.append("Add at least one uppercase letter.")

    if re.search(r'[a-z]', password):
        score += 1
    else:
        feedback.append("Add at least one lowercase letter.")

    if re.search(r'[0-9]', password):
        score += 1
    else:
        feedback.append("Add at least one digit.")

    if re.search(r'[!@#$%^&*(),.?":{}|<>]', password):
        score += 1
    else:
        feedback.append("Add at least one special character (e.g., !, @, #).")


    # Evaluate final score
    if score >= 6:
        strength = "Strong"
    elif score >= 4:
        strength = "Medium"
    else:
        strength = "Weak"

    return {
        "strength": strength,
        "score": score,
        "feedback": feedback
    }
```

```
result = check_password_strength(password)
print(f"Strength: {result['strength']} (Score: {result['score']})")
if result['feedback']:
    print("Suggestions for improvement:")
    for suggestion in result['feedback']:
        print(f"- {suggestion}")
if result['strength'] == "Weak":
        print("We strongly advise you choose a stronger password.")
        sys.exit(1)

credentials_file = sys.argv[2]

client.subject_credentials(password, credentials_file)
```

```
alof@alof-IdeaPad-5-14IAL7:~/Desktop/SIO/sio-2425-project-115243_113480_112665/delivery2$ ./rep_subject_credentials "palavrapasse" chave1.pem
Strength: Weak (Score: 3)
Suggestions for improvement:
- Add at least one uppercase letter.
- Add at least one digit.
- Add at least one special character (e.g., !, @, #).
We strongly advise you choose a stronger password.
```

```
alof@alof-IdeaPad-5-14IAL7:~/Desktop/SIO/sio-2425-project-115243_113480_112665/delivery2$ ./rep_subject_credentials "palavrapasse!" chave1.pem
Strength: Medium (Score: 4)
Suggestions for improvement:
- Add at least one uppercase letter.
- Add at least one digit.
ECC key pair generated successfully.
Encrypted private key saved to chave1.pem
Public key saved to chave1_public.pem
```

- **2.1.9** - Verify that there are no password composition rules limiting the type of characters permitted. There should be no requirement for upper or lower case or numbers or special characters - Fulfilled

Evidence: we previously had no restrictions of any kind - however if we were to implement what we have now been doing here, then we'd only need to change one thing:

```
if result['strength'] == "Weak":
        print("We strongly advise you choose a stronger password.")
```

Instead of exiting the program when the password is weak, we simply suggest that the user picks a stronger password

- **2.1.10** - Verify that there are no periodic credential rotation or password history requirements - Fulfilled

Evidence: nowhere in our code is such a mechanism implemented.

- **2.1.11** - Verify that "paste" functionality, browser password helpers, and external password managers are permitted. - Fulfilled

Evidence: the password is pasted into the terminal. Even if we wanted to, we couldn't find a way to prevent users from pasting in the terminal. We don't operate on a browser so that part doesn't apply to our repository.

- **2.1.12** - Verify that the user can choose to either temporarily view the entire masked password, or temporarily view the last typed character of the password on platforms that do not have this as built-in functionality. - Fulfilled

Evidence: again, we have no control over what is typed in the terminal. The user can see the password fully when typing in the terminal.

- **2.2.1** - Verify that anti-automation controls are effective at mitigating breached credential testing, brute force, and account lockout attacks. Such controls include blocking the most common breached passwords, soft lockouts, rate limiting, CAPTCHA, ever increasing delays between attempts, IP address restrictions, or risk-based restrictions such as location, first login on a device, recent attempts to unlock the account, or similar. Verify that no more than 100 failed attempts per hour is possible on a single account - Not Fulfilled

How to fix:

Most of these don't apply, such as using CAPTCHA. We could however prevent some of these issues. We could slightly modify the DB schema in order to be able to, for example, check every failed attempt in the last hour:

```
CREATE TABLE Authentication_Attempt (
    id INTEGER PRIMARY KEY,
    subject_id INTEGER REFERENCES Subject(id) ON DELETE CASCADE,
    organization_id INTEGER REFERENCES Organization(id) ON DELETE CASCADE,
    timestamp TIMESTAMP NOT NULL
);
```

When creating a session (which is our login), we can check if there are more than 100 rows that reference the subject and organization that is being aimed at. In `./repository.py`:

```
#delete old authentication attempts
cur.execute(
    "DELETE FROM Authentication_Attempt WHERE timestamp < ?",
    (time.time() - 3600,)
)

#check last authentication attempts for this subject + organization
all_auths = query_db(
    "SELECT * FROM Authentication_Attempt WHERE subject_id = ? AND organization_id = ?", (subject_id, organization)
)

if all_auths:
    if len(all_auths) > 100:
        return jsonify(signed_payload({"error": "Too many authentication attempts in the past hour"})), 403
    last_auth = all_auths[-1]
    if time.time() - last_auth["timestamp"] < 0.5:
        return jsonify(signed_payload({"error": "Authentication attempts are too fast"})), 403

#add authentication attempt to this session
curtime = time.time()
cur.execute(
    "INSERT INTO Authentication_Attempt (subject_id, organization_id, timestamp) VALUES (?, ?, ?)",
    (subject_id, organization_data["id"], curtime)
)
```

We do this after verifying the subject_id and organization

```
conn.commit()
#attempt successful, remove attempt made by this session
cur.execute(
    "DELETE FROM Authentication_Attempt WHERE subject_id = ? AND organization_id = ? AND timestamp = ?",
    (subject_id, organization_id, curtime)
)
return jsonify(signed_payload({"message": "Session created successfully", "session_data": session_data})), 200
```

If the authentication is successful, we delete the attempt from the DB (we only want failed attempts)

- **2.2.2** - Verify that the use of weak authenticators (such as SMS and email) is limited to secondary verification and transaction approval and not as a replacement for more secure authentication methods. Verify that stronger methods are offered before weak methods, users are aware of the risks, or that proper measures are in place to limit the risks of account compromise. - N/A

- **2.2.3** - Verify that secure notifications are sent to users after updates to authentication details, such as credential resets, email or address changes, logging in from unknown or risky locations. The use of push notifications - rather than SMS or email - is preferred, but in the absence of push notifications, SMS or email is acceptable as long as no sensitive information is disclosed in the notification. - N/A

- **2.2.4** - Verify impersonation resistance against phishing, such as the use of multi-factor authentication, cryptographic devices with intent (such as connected keys with a push to authenticate), or at higher AAL levels, client-side certificates. - Fulfilled

Evidence:

The client signs every message sent when in the context of a session:

```
signed_payload_ = signed_payload(encrypted_payload, private_key)
```

Which is a form of protection against impersonation, as the repository only accepts messages signed by the client. People trying to impersonate a subject would need to forge this signature as well.

- **2.2.5** - Verify that where a Credential Service Provider (CSP) and the application verifying authentication are separated, mutually authenticated TLS is in place between the two endpoints. - N/A

- **2.2.6** - Verify replay resistance through the mandated use of One-time Passwords (OTP) devices, cryptographic authenticators, or lookup codes. - Fulfilled

Evidence:

Every message sent using a session includes a nonce:

```
payload = {
    "organization": organization,
    "subject": subject,
    "username": username,
    "name": name,
    "email": email,
    "public_key": public_key,
    "nonce": os.urandom(16).hex(),
    "seq_number": seq_number,
    "signature": signature
}
```

If the message doesn't have a nonce or it is repeated, the repository will generate an error:

```
nonce = data.get("nonce")
seq_number = data.get("seq_number")
signature = data.get("signature")

if not verify_signature(subject_id, org, signature):
    return jsonify(signed_payload({"error": "Authentication failed"})), 403

if not verify_nonce(subject_id, org, nonce) or not verify_sequential_number(subject_id, org, seq_number):
    return jsonify(signed_payload({"error": "Invalid nonce or sequence number. Woop woop that's the sound of the police!"})), 403
```

- **2.2.7** - Verify intent to authenticate by requiring the entry of an OTP token or user-initiated action such as a button press on a FIDO hardware key - Not fulfilled

How to fix:

We could implement a sort of "manual challenge" system where the repository would select a question from a list of questions with an easy answer (such as "What color is the sky?"), encrypted with the subject's public key, the user would then decrypt the message, reply with "blue", and then encrypt the message, sign it, and return it to the repository. Upon verifying the correct message the interaction would proceed as normal.

Regarding the security measures implemented towards the password, it should be noted that our authentication not only requires the correct password, but it also requires the correct private key, obtained when and only when creating credentials, using `./rep_subject_credentials` which greatly mitigates the impact caused by a password breach.

From 2.3 to 2.7, none of the controls apply, since we do not store passwords, we don't have look-up secrets and we don't have out band verifiers

- **2.8.1** - Verify that time-based OTPs have a defined lifetime before expiring - Fulfilled

Evidence:

```
if not challenge_data:
    challenge = os.urandom(32)
    print(challenge)
    query_db(
        "INSERT INTO Challenge (subject_id, organization_id, challenge, timestamp) VALUES (?, ?, ?, ?)",
        (subject_org["subject_id"], subject_org["organization_id"], challenge, time.time()),
        commit=True
    )

elif time.time() - challenge_data["timestamp"] > 5:
    challenge = os.urandom(32)
    query_db(
        "UPDATE Challenge SET challenge = ?, timestamp = ? WHERE subject_id = ? AND organization_id = ?",
        (challenge, time.time(), subject_org["subject_id"], subject_org["organization_id"]),
        commit=True
    )
else:
    challenge = challenge_data["challenge"]
payload = {"challenge": encrypt(challenge.hex(), subject_org["public_key"]), "rep_pub_key": PUBLICKEY}
```

You can see here that the challenge (which is our OTP) has a 5 second limit.

- **2.8.2** - Verify that symmetric keys used to verify submitted OTPs are highly protected, such as by using a hardware security module or secure operating system based key storage - N/A

- **2.8.3** - Verify that approved cryptographic algorithms are used in the generation, seeding, and verification of OTPs. - Fulfilled

Evidence:

Our challenge is generated according to the cryptography website recommendation:

> Therefore, it is our recommendation to always use your operating system's provided random number generator, which is available as `os.urandom()`.

```
challenge = os.urandom(32)
```

We then encrypt the challenge using the subject's public key.

```
payload = {"challenge": encrypt(challenge.hex(), subject_org["public_key"]), "rep_pub_key": PUBLICKEY}
```

- **2.8.4** - Verify that time-based OTP can be used only once within the validity period - Not fulfilled

How to fix:

Simply delete the challenge from the DB once it is successful:

```
challenge_data = query_db(
    "SELECT * FROM Challenge WHERE subject_id = ? AND organization_id = ?",
    (subject_id, organization_id),
    one=True
)

if not challenge_data:
    print("no challenge")
    return jsonify(signed_payload({"error": "Invalid credentials"})), 403

if challenge_data["timestamp"] - time.time() > 5:
    return jsonify(signed_payload({"error": "Challenge expired"})), 403

if challenge != challenge_data["challenge"].hex():
    print("challenge", challenge)
    print("challenge_data", challenge_data["challenge"].hex())
    return jsonify(signed_payload({"error": "Invalid credentials"})), 403

query_db(
    "DELETE FROM Challenge WHERE subject_id = ? AND organization_id = ?",
    (subject_id, organization_id),
    commit=True
)
```

- **2.8.5** - Verify that if a time-based multi-factor OTP token is re-used during the validity period, it is logged and rejected with secure notifications being sent to the holder of the device. - Not fulfilled

How to fix:

It is challenging to send a live notification in this situation but we could setup a system where whenever the user makes a valid login, they get notified about all the potential malicious login attempts. To do this, we should store previous challenges in a separate table, and instead of just deleting the challenge when successful, we should store that challenge in that separate table. Then, whenever a login attempt is unsuccessful due to the challenge, we (by creating another separate table) can store a notification associated with the subject in question containing the timestamp of the attempted login, and such notification is to be displayed whenever a user creates a session.

Our application does not support usage of cryptographic keys / FIDO keys, therefore none of the points in 2.9 apply

- **2.10.1** - Verify that intra-service secrets do not rely on unchanging credentials such as passwords, API keys or shared accounts with privileged access. - Fulfilled

Evidence: our secrets do not rely on keys - only the access to the repository's data by a client depends on it (therefore, not intra-service secrets)

- **2.10.2** - Verify that if passwords are required for service authentication, the service account used is not a default credential. (e.g. root/root or admin/admin are default in some services during installation). - Not fulfilled

How to fix: currently our masterkey password is “art” but we could have it be a more complicated password such as a random 64-character password.

Conclusion

This project allowed us to experience what it was like to create an application that required ensuring security within an organization and in communications. We employed several concepts as encryption of messages, encryption of contents and security within an organization by applying the principle of least privilege. It is impossible to create a perfect defense, but by employing the knowledge learned in theoretical classes (e.g. hybrid encryption of communications) we certainly made it much harder for potential attackers to perform all manners of attacks. It wasn't until we inspected the OWASP controls though that we realized how many breaches our application still had. It was even more evident to us that it is impossible to maintain perfect security, but we managed to improve security even further after making adjustments based on said controls. With this, we believe that this project covered many aspects of the subject, and we learned a lot about security by creating this repository.