

Lab 4

3.

Question 1

L'inscription a été autorisée malgré le mot de passe faible. C'est un problème car l'utilisateur inscrit est vulnérable à une attaque par « brute force ». Il faut pour y remédier par exemple mettre en place une politique de mot de passe forte. C'est-à-dire obliger l'utilisateur à utiliser un mot de passe d'au minimum 12 signes avec au moins une majuscule, un chiffre et un caractère spécial.

```
33 # Route pour s'inscrire
32 @app.route("/register", methods=["POST"])
31 def register():
30     username = request.json["username"]
29     password = request.json["password"]
28
27     if len(password) < 12:
26         return jsonify({"message": "Password too short (min length: 12)"}), 400
25     elif not check_password(password):
24         return (
23             jsonify(
22                 {
21                     "message": "Password not valid !\n\
20                     You need at least:\n\
19                     - 1 Uppercase letter\n\
18                     - 1 Number\n\
17                     - 1 Special Character(#!$%^&*()@!...) "
16                 }
15             ),
14             400,
13         )
12
2 # fonction pour tester le mot de passe
1 def check_password(password):
15     has_num = False
1     has_upper = False
2     has_special = False
3     for s in password:
4         if s.isdigit():
5             has_num = True
6         if s.isupper():
7             has_upper = True
8         if not s.isdigit() and not s.isalpha():
9             has_special = True
10     return has_num and has_upper and has_special
11
12
```

Pour ce qui est de l'implémentation de mon idée j'ai simplement ajouté différents contrôles sur le mot de passe proposé par l'utilisateur afin qu'il respecte la politique de mot de passe voulue. (longueur, majuscules, chiffre, caractère spécial)

4.

Question 2

```
student@vm04:~/lab4/lab4$ sqlite3 local.db "SELECT * FROM users"
1|borito|oui
2|bor|Borisverdecia2@
```

On voit que les mots de passes sont stockés sous forme de texte clair. Donc toute personne ayant accès à la base de données peut connaître le mot de passe de chaque utilisateur.

Question 3

La meilleure solution niveau sécurité est argon2. Cependant, son exécution demande plus de ressources que Bcrypt. Bcrypt est présent depuis plus longtemps, il est donc plus testé et plus utilisé. SHA-256 quant à lui n'est pas fait pour être utilisé lors du hachage de mot de passe, car il est vulnérable aux attaques par GPU. Dans notre cas au vu de la simplicité de la « web app » j'ai opté pour l'algorithme le plus sécurisé c'est-à-dire argon2.

```
(venv) student@vm04:~/lab4/lab4$ sqlite3 local.db "SELECT * FROM users"
1|boris|$argon2id$v=19$m=65536,t=3,p=4$jRmpyRrLz1w0Jm$1V4+UKQ$GNY0kWvLZhfsLcFIM5/FgwAz3xd2BBzYvfbKvbqcDe8
2|vebo|$argon2id$v=19$m=65536,t=3,p=4$tjc3K8s6SVpVyyy7BVhbDA$lrC7K+8Pi6ZxoW/eP5lNynorEUzD1weB12D1mFEoyEs
3|bove|$argon2id$v=19$m=65536,t=3,p=4$f6TeBK6XKK0eN4Jds/ZSbQ$XUixh/TjRHRBt0/IdJh/gpvl26QPILTL5qMOCBZsmU
(venv) student@vm04:~/lab4/lab4$
```

À noter que les utilisateurs « vebo » et « bove » ont les mêmes mots de passes mais les hash sont différents !

```
2 ph = PasswordHasher()
```

Pour ce qui est de l'implémentation. On utilise la librairie argon2 et plus précisément l'objet PasswordHasher() ici contenu dans la variable globale ph.

```
# Route pour se inscrire
@app.route("/register", methods=["POST"])
def register():
    username = request.json["username"]
    password = request.json["password"]

    if len(password) < 12:
        return jsonify({"message": "Password too short (min length: 12)"}), 400
    elif not check_password(password):
        return (
            jsonify(
                {
                    "message": "Password not valid !\n\n
                    You need at least:\n\n
                    - 1 Uppercase letter\n\n
                    - 1 Number\n\n
                    - 1 Special Character(#!$%^&*()@!...) "
                }
            ),
            400,
        )

    password = ph.hash(password)
    conn = get_db_connection()
    try:
        conn.execute(
            "INSERT INTO users (username, password) VALUES (?, ?)", (username, password)
        )
        conn.commit()
    except sqlite3.IntegrityError:
        return jsonify({"message": "Username already exists!"}), 400
    finally:
        conn.close()

    return jsonify({"message": "User registered successfully!"}), 201
```

On crée un hash avec le mot de passe, qu'on vient ensuite stocker dans la base donnée. On aura donc un hash dans la base de données plutôt que le texte en clair.

```
# Route pour se connecter
@app.route("/login", methods=["POST"])
def login():
    username = request.json["username"]
    password = request.json["password"]

    conn = get_db_connection()
    user = conn.execute(
        "SELECT * FROM users WHERE username = ?", (username,)
    ).fetchone()
    conn.close()

    try:
        ph.verify(user["password"], password)
        return jsonify({"message": "OK. Welcome user " + user["username"]}), 200
    except:
        return jsonify({"message": "Invalid credentials!"}), 401
```

Puis lors du login, on vérifie la correspondance du mot de passe fourni par l'utilisateur avec le hash stocké dans la base de données pour ce même utilisateur, grâce à la méthode verify() de notre objet.

5.

Question 4


```
# Route pour la page d'inscription
@app.route("/web/register")
def register_page():
    totp_secret = pyotp.random_base32()
    return render_template("register_otp.html", totp_secret=totp_secret)

# Route pour la page de connexion
@app.route("/web/login")
def login_page():
    totp_secret = pyotp.random_base32()
    return render_template("login_otp.html")
```

Premièrement, on change les pages d'inscription et de login. Sur la page d'inscription on ajoute le secret totp généré.

```
3 # Créer la table si elle n'existe pas
2 def init_db():
1     conn = get_db_connection()
     conn.execute(
         """
         CREATE TABLE IF NOT EXISTS users (
             id INTEGER PRIMARY KEY AUTOINCREMENT,
             username TEXT NOT NULL UNIQUE,
             password TEXT NOT NULL,
             secret_totp TEXT NOT NULL
         )
         """
     )
     conn.commit()
     conn.close()
```

On n'oublie pas d'ajouter la nouvelle colonne avec le secret totp dans la base de données.


 Ajouter un compte

AUTRE COMPTE


Nom de compte

Clé secrète

Ensuite, grâce au secret affiché sur la page d'inscription, on ajoute un compte sur notre application d'authentification préférée.

 bortio
bortio

>

047 743 

On obtient un code de vérification qui se rafraîchi automatiquement toutes les 30 secondes.

Inscription

bortio

.....

DJ7HKVGZQIZLHJ7KWU3GZ

344721

S'inscrire

Puis nous complétons l'inscription grâce au code de vérification.

Connexion

bortio

.....

256851

Se connecter

OK. Welcome user bortio

On utilise le même procédé pour la connexion au compte.

```

# Route pour s'inscrire
@app.route("/register", methods=["POST"])
def register():
    username = request.json["username"] # L'objet de type "Aucun" n'est pas inscriptible
    password = request.json["password"] # L'objet de type "Aucun" n'est pas inscriptible
    secret_totp = request.json["totp_secret"] # L'objet de type "Aucun" n'est pas inscriptible
    code = request.json["totp"] # L'objet de type "Aucun" n'est pas inscriptible
    totp = pyotp.TOTP(secret_totp)

    if len(password) < 12:
        return jsonify({"message": "Password too short (min Length: 12)"}), 400
    elif not check_password(password):
        return (
            jsonify(
                {
                    "message": "Password not valid !\n\
                    You need at least:\n\
                    - 1 Uppercase letter\n\
                    - 1 Number\n\
                    - 1 Special Character(#!$%^&*()@!...) "
                }
            ),
            400,
        )
    elif not totp.verify(code):
        return jsonify({"message": "Le code ne correspond pas !"}), 400
    password = ph.hash(password)
    conn = get_db_connection()

```

Au niveau du code pour l'inscription on ajoute simplement 3 variables, une contenant le secret, une le code et la dernière un objet de la librairie pyotp. Ensuite, il nous reste simplement qu'à vérifier si le code d'authentification fourni correspond bien au code actuel pour le secret généré correspondant.

```

conn = get_db_connection()
try:
    conn.execute(
        "INSERT INTO users (username, password, secret_totp) VALUES (?, ?, ?)",
        (username, password, secret_totp),
    )
    conn.commit()

```

Et finalement on n'oublie pas d'ajouter le secret totp du nouvel utilisateur dans la base de données.

```

# Route pour se connecter
@app.route("/login", methods=["POST"])
def login():
    username = request.json["username"] # L'objet de type "Aucun" n'est pas inscriptible
    password = request.json["password"] # L'objet de type "Aucun" n'est pas inscriptible
    code = request.json["totp"] # L'objet de type "Aucun" n'est pas inscriptible

    conn = get_db_connection()
    user = conn.execute(
        "SELECT * FROM users WHERE username = ?", (username,)
    ).fetchone()
    conn.close()
    if not user:
        return jsonify({"message": "Invalid credentials!"}), 401
    try:
        ph.verify(user["password"], password)
    except:
        return jsonify({"message": "Invalid credentials!"}), 401
    finally:
        totp = pyotp.TOTP(user["secret_totp"])
        if not totp.verify(code):
            return jsonify({"message": "Invalid credentials!"}), 401
        return jsonify({"message": "OK. Welcome user " + user["username"]}), 200

```

Pour le login on utilise le même procédé avec quelques ajustements au niveau de l'accès au secret qui se trouve dans la base donnée.