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.

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. (longeur, 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$lV4+UKQ$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/gpvl26QPIlTlL5qM0CBZsmU
(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 librarie argon2 et plus précisemment l'objet PasswordHasher() ici contenu dans la variable globale ph.

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.

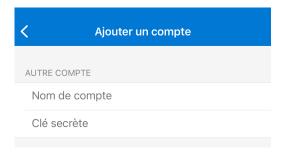
Puis lors du login, on vérifie la correspondance du mot de passe fourni par l'utilisateur avec le hash stocké dans la base de données pour ce même utilisateur, grâce à la méthode verify() de notre objet.

5.

Question 4

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

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

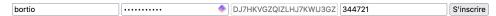


Ensuite, grâce au secret affiché sur la page d'inscription, on ajoute un compte sur notre application d'authentification préférée.



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

Inscription



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

Connexion



On utilise le même procédé pour la connexion au compte.

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.

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

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.