T.P.

CRYPTOGRAPHIE

DEFINITIONS : CLASSE + PARAMETRE

- class Ulrich (Cedric) : Definit une classe Ulrich qui derive de la classe Cedric
- def __init__ (self): definit un constructeur de la classe Ulrich.
- super().__init__() appelle la classe Cedric pour init les attribut de base

self._key = None / initialis_key stocke la clé de chiffrement

self._salt : initialise une valeur en bytes utilisé dans la deriv de clé.

def _create_conect_window (self). Def de methode.

MDP + BOUTON

if field == "password":

dpg.add_input_text (default_value = "", tag = f"connection_{field}", pss=T)

else:

dpg.add_input_text(default)

↳ si le champ est password -> zone de texte masquée / si non défaut.

dpg.add_button. (label="...", callback= ___).

↳ crée un bouton et si cliqué -> ouvre methode.

hash = dpg.get_value -> récupère la donnée entrée par user.

DERIVATION DE LA CLÉ

Definir un objet de dérivat° de la clé (kdf) grace à PBKDF2HMAC

```
kdf = PBKDF2HMAC(
    algo = hashes.SHA256(),   - Ne pas stocker le mdp en clair mais son empreinte (hash)
    length = 16 (octets)      - taille de la clé
    salt = self._salt,        - sel constant pour la séc du hachage de la clé
    iterat = 1080000,
)
```

on convertit le mdp entré (chaine de caract) en byte

```
self._key = kdf.derive(password.encode())
```

On génère la clé à partir du mdp entré en bytes grâce à .encode

La clé de chiffrement crée grâce au mdp est dérivée avec un salt constant et stockée dans le self._key déclaré au début.

CHIFFRE UN MESSAGE

`def encrypt(self, message:str) -> tuple`

↳ cette meth retourne un tuple qui contient l'IV et le mess chiffré. (tuple = collection de 2 sieurs éléments.)

- IV = os.urandom(16) → IV aléatoire de 16 octets.
  c'est une clé secrette qui empêche le déchiffrement non autorisé du message.
- cipher = Cipher(algorithms.AES(self._key), modes.CTR(IV))
  ↳ création d'objet de chiffrement AES en mode CTR. (counter
- encryptor = cipher.encryptor() → chiffrer les données

* Il faut ensuite rendre le message compatible avec AES.
donc on le rembourre avec des octets en plus.
↳ padder = padding.PKCS7(128).padder()
padder_data = padder.update(message.encode()) + padder.finalize()

- On retourne enfin l'IV et le message chiffré.

ENVOYER LE MSG CHIFFRÉ

- Message d'erreur lorsque la clé de chiffrement n'a pas été générée
"iv/message" : base64.b64encode(iv/encrypted data).decode('utf8')
↳ Les données doivent être envoyées en base 64 (transmission + sure)
- On serialise les données (convertir en format transportable)
↳ on utilise "serpent.dumps"
- On convertit ces données sérialisées en bytes
↳ on utilise "serpent.tobytes"

Pour la reception:

- On désérialise les données reçues avec "serpent.frombytes" et "serpent loads"
- On décode l'IV et le message chiffré de base 64
- On déchiffre le message et on l'affiche.

FERNETGUI.PY

1) Génération de la clé

- Objet de hachage: -> hash_obj = hashes.Hash(hashes.SHA256())
  - ↳ SHA 256 -> un hash de 256 bits (32 octets)

haché grace à SHA 256

(.encode) Encoder le mdp en base 64 puis l'ajoute au hash
  - ↳ key.update(password.encode())

- la finalisation du hash permet d'obtenir les byte de la clé.
- la clé (en byte.) est encodé en base 64 et stockée dans fernet_key.
  - ↳ fernet_key = base64.b64encode(key_bytes)

2) Encrypt

- le message entré (normal) est pris et codé en bytes
- Chiffrement du message avec (self._fernet).
- On retourne le msg sous forme de bytes.

3) Décrypt.

- les données chiffrées sont pris en entrée et dechffrée avec self.fernet.decrypt