CÉGEP MARIE-VICTORIN

TP : Chiffrement Symétrique et Sécurité des mots de passe

420-313-MV: Sécurité informatique

TP 1 partie 1 : Chiffrement Symétrique – Shift + Permutation

Objectif

L'objectif de ce TP est de créer un algorithme de chiffrement symétrique combinant :

1. Un **shift de type César**.
2. Un **mapping de permutation** aléatoire pour chaque lettre.

Vous utiliserez les **dictionnaires Python** (`dict`) pour représenter la permutation.

Instructions

1. Créer une permutation

Créez un dictionnaire Python qui mappe chaque lettre de l'alphabet vers une autre lettre. Exemple :

```
permutation = {'a': 'q', 'b': 'm', 'c': 'x', ...}
```

Vous pouvez générer aléatoirement cette permutation avec `random.shuffle()`.

2. Chiffrement

Pour chaque lettre du texte :

1. Appliquez un **shift** d'un certain nombre (`shift`) comme dans le chiffre de César.
2. Appliquez la **permutation** à l'aide du dictionnaire.

Conservez les espaces et la ponctuation.

3. Déchiffrement

1. Inversez le dictionnaire de permutation.
2. Appliquez l'inverse du shift.

4. Exemple de fonctionnement attendu

```
text = "HelloWorldThisismyplaintext"
shift = 3
permutation = create_permutation()  # fonction a implementer

cipher = encrypt(text, shift, permutation)
print(cipher)

plain = decrypt(cipher, shift, permutation)
print(plain)  # doit afficher "HelloWorldThisismyplaintext"
```

5. Réflexion

Répondez aux questions suivantes :

1. Pourquoi ce chiffrement est-il plus difficile à brute force qu'un simple César ?
2. Pourquoi l'analyse de fréquence pourrait-elle permettre de deviner certaines lettres ?
3. Comment l'utilisation d'un dictionnaire (`dict`) facilite-t-elle l'implémentation de la permutation ?

Conseil pédagogique

- Utilisez les dictionnaires pour mapper rapidement les lettres.
- Pour inverser un dictionnaire, vous pouvez utiliser :

  ```
  reverse_map = {v: k for k, v in permutation.items()}
  ```

TP 1 partie 2 : Sécurité des mots de passe et attaques

Objectif

L'objectif de cette partie du TP est de comprendre la sécurité des mots de passe, la vérification d'identifiants et les limites des attaques par force brute et rainbow table. Les étudiants travailleront sur trois fichiers CSV :

1. `passwords_plain.csv` : mots de passe en clair.
2. `passwords_hash.csv` : mots de passe hachés avec SHA-256.
3. `passwords_hash_salt.csv` : mots de passe hachés avec SHA-256 + sel unique par mot de passe.

Instructions

1. Vérification des mots de passe (fonction "login")

Un fichier Python est fourni avec une fonction `check_login(user, password, users_db)` qui simule la vérification d'un mot de passe pour un utilisateur.

Exemple d'utilisation :

```
from auth import check_login
users_db = load_csv("data/passwords_plain.csv")

if check_login("alice", "motdepasse123", users_db):
    print("Login reussi")
else:
    print("Login echoue")
```

2. Attaque brute force sur les mots de passe en clair

- Chargez `passwords_plain.csv` et récupérez les mots de passe.
- Écrivez une fonction qui teste toutes les combinaisons possibles pour retrouver les mots de passe à l'aide de `check_login`.

Observation : Cette attaque réussit facilement car les mots de passe sont en clair.

3. Rainbow table sur les mots de passe hachés

- Utiliser *passwords* pour charger des mots de passe fréquents.
- Chiffrez ces mots de passe avec SHA-256 (hashlib)
- Créez `passwords_hash.csv` en hachant les mots de passe du csv original.
- Testez chaque mot de passe deviné avec `check_login`.

Observation : Cela fonctionne pour les mots de passe hachés sans sel.

4. Rainbow table sur les mots de passe avec sel

- Créez `passwords_hash_salt.csv` en ajoutant un sel unique pour chaque user, conservé dans la base de données, avant d'appliquer le hash. Il y aura trois colonnes: username, salt, hashed_password
- Essayez d'utiliser votre rainbow table.

Observation : Cela **ne fonctionne pas** car chaque mot de passe a un sel unique, ce qui rend la rainbow table inefficace.

Questions de réflexion

1. Pourquoi les mots de passe hachés avec sel résistent mieux aux attaques par rainbow table ?

2. Pourquoi la force brute est inefficace sur des mots de passe hachés même sans sel ?

3. Que montre cet exercice sur l'importance du sel et des fonctions de hachage sécurisées dans les systèmes de login ?

Arborescence du projet

Pour ce TP, nous allons organiser le projet de manière structurée afin de séparer les différentes parties : données, fonctions utilitaires et code principal. L'arborescence proposée est la suivante :

```
project/
|
+-- data/
|   +-- passwords_plain.csv
|   +-- passwords_hash.csv
|   +-- passwords_hash_salt.csv
|
+-- utils/
|   +-- auth.py           # contient la fonction check_login
|   +-- crypto_utils.py   # fonctions pour le chiffrement shift + permutation
|
+-- src/
|   +-- tp1_chiffrement.py   # script pour le chiffrement
|   +-- tp2_passwords.py     # script pour les attaques sur mots de passe
```

Description des dossiers

- **data**/ : contient tous les fichiers CSV utilisés pour le TP.
- **utils**/ : contient les fonctions réutilisables telles que `check_login` et les fonctions de chiffrement.
- **src**/ : contient les scripts principaux que vous exécuterez pour chaque partie du TP.

Courage!