TP – Signatures numériques et Autorité de Certification simulée

🎯 Objectif général

Mettre en œuvre une application de système simple de **signatures électroniques** en Python avec interface en utilisant Django.
Vous allez :

- Générer des paires de clés RSA
- Simuler une « Autorité » qui enregistre les clés publiques
- Signer des fichiers
- Vérifier les signatures
- Comprendre l'intérêt d'une infrastructure de confiance

📌 Contexte

Dans ce TP, vous jouerez deux rôles :

✅ **Utilisateur** : génère ses clés et signe des documents.
✅ **Autorité (CA simplifiée)** : stocke les clés publiques des utilisateurs, distribue ces clés et vérifie les signatures.

✅ Étape 1 – Génération des clés RSA

🎯 Objectif :

- Générer une paire de clés RSA pour un utilisateur (par ex. user1)
- Sauvegarder les clés dans des fichiers PEM

📌 Consignes :

- Générer une clé privée RSA (2048 bits minimum).
- La génération se fait en déhors de l'application principale à faire
- Extraire la clé publique.
- Sauvegarder :
  - `user1_private.pem`
  - `user1_public.pem`

✅ Question bonus :

> Pourquoi la clé privée doit-elle rester secrète ?

✅ Étape 2 – Simuler l'Autorité de Certification

🎯 Objectif :

- Créer un registre des utilisateurs et de leurs clés publiques.
- une page d'inscription pour les utilisateurs en important leurs fichiers de clé publique

📌 Consignes :

- Créer un « registre » sous forme de dictionnaire ou fichier JSON :

```
{
  "user1": "clé_publique_pem",
  "user2": "clé_publique_pem"
}
```

- Implémenter une fonction :

```
def register_user(username, public_key):
    # Ajoute l'utilisateur et sa clé au registre
```

- Sauvegarder et charger le registre.

✅ Question bonus :

> Quelles seraient les bonnes pratiques pour ce registre dans un système réel?

✅ Étape 3 – Signature d'un fichier

🎯 Objectif :

- l'utilisateur peut visualiser les fichiers qu'il doit signer (uniquement des fichiers .txt)
- Il peut alors digner le contenu d'un fichier texte avec RSA.
- dater la signature

📌 Consignes détaillées :

1. Lire le fichier `document.txt`.
2. Calculer son hash SHA-256.
3. Signer ce hash avec la clé privée RSA (utiliser PSS ou PKCS1v15).
4. Sauvegarder la signature dans un fichier `.sig`.

✅ Code-type attendu :

```
signature = private_key.sign(
    hash_value,
    padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH
    ),
```

```
    hashes.SHA256()
)
```

✅ Consigne de sauvegarde :

- Créez un fichier `document.sig` contenant la signature binaire.
- Sauvegardez signature + métadonnées dans un JSON :

```
{
  "user": "user1",
  "timestamp": "2025-07-07T12:00:00Z",
  "signature": "base64..."
}
```

✅ Question bonus :

> Pourquoi signer le **hash** et pas tout le fichier directement?

✅ Étape 4 – Vérification de la signature

🎯 Objectif :

- Vérifier la signature d'un fichier signé.

📌 Consignes :

- Charger le fichier `document.txt`.
- Calculer son hash SHA-256.
- Charger la clé publique du registre.
- Vérifier la signature.
- Afficher « Signature VALIDE » ou « Signature INVALIDE ».

✅ Code-type attendu :

```
public_key.verify(
    signature,
    hash_value,
    padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH
    ),
    hashes.SHA256()
)
```

✅ Consignes pédagogiques :

- Implémentez une fonction :

```
def verify_signature(username, file_path, signature_path):
    # Charge la clé publique du registre et vérifie la signature
```

✅ Question bonus :

> Quelles informations doit-on vérifier en plus du hash ? (timestamp, user...)

✅ Étape 5 – Simulation d'attaque MITM

🎯 Objectif :

- Comprendre la menace du remplacement de clé publique.

📌 Consignes :

- Montrez qu'un attaquant peut remplacer la clé publique de `user1` dans le registre.
- Signez un fichier en se faisant passer pour `user1`.
- Vérifiez la signature (qui sera « valide » si le registre est corrompu).

✅ Discussion attendue :

> Que faudrait-il ajouter pour éviter cette attaque ? Réponse attendue : « certificats numériques signés par une autorité de confiance »

✅ BONUS – Génération d'un certificat rudimentaire

🎯 Objectif :

- Simuler un « certificat » signé par l'autorité.

📌 Consignes :

- Structure du certificat :

```
{
  "username": "user1",
  "public_key": "...",
  "CA_signature": "..."
}
```

- Signature CA sur :

```
hash(username + public_key)
```

- Vérification :

- L’autorité vérifie sa propre signature sur le certificat avant d’ajouter la clé au registre.

✅ Question bonus :

> Quelle différence avec le registre naïf ?

✅ 💾 Remise attendue

- Fichiers PEM des clés générées.
- Registre des clés publiques (JSON).
- Une application coté client pour signer les documents
- Une application web pour uploader les sigantures et pour vérifier des signatures sur un document
- Simulation d’attaque MITM
- Explications aux questions

✅ 💡 Compétences visées

- RSA : génération, signature, vérification
- SHA-256 : hachage sécurisé
- Gestion des clés publiques
- Notion d’infrastructure à clé publique (PKI)
- Compréhension des attaques et des mesures de protection

Bonne pratique et amusez-vous bien !