Cryptographie **I**ndustrielle **A**vancée // TP2

Sécurisation d'un Flux de Données de Capteur IIoT

Loïc Rouquette

I. Objectifs Pédagogiques

Ce TP vise à fournir une expérience pratique sur la sécurisation de communications dans un contexte d'Internet Industriel des Objets (IIoT). À l'issue de cette séance, vous devrez être capable de :

1. **Comprendre et justifier** le besoin de chiffrements authentifiés (AEAD).
2. **Mettre en œuvre** un schéma AEAD moderne (ASCON) en Python pour garantir la confidentialité et l'intégrité d'un flux de données.
3. **Manipuler** correctement les différents éléments d'un AEAD : clé, nonce, données associées, et ciphertext.
4. **Démontrer expérimentalement** les garanties d'intégrité d'un AEAD face à une corruption des données.
5. **Analyser** les limites de ce schéma de protection, notamment face à certains types d'attaques.

II. Contexte Théorique : L'Insuffisance du Chiffrement Seul

Dans de nombreux scénarios, et particulièrement en IIoT, les données transitent sur des canaux non fiables (MQTT sur TCP/IP, LoRaWAN, etc.). La première préoccupation est la **confidentialité** : s'assurer que seuls les destinataires autorisés peuvent lire les données. Des algorithmes de chiffrement symétrique comme AES le permettent.

Cependant, un attaquant peut intercepter un message chiffré et, même sans pouvoir le déchiffrer, le modifier avant de le ré-acheminer. Si le destinataire n'a aucun moyen de vérifier que le message a été altéré, il pourrait déchiffrer des données corrompues et les traiter comme valides, menant à des décisions critiques erronées (ex: ajuster une valve sur la base d'une fausse mesure de pression).

C'est ici qu'intervient le **chiffrement authentifié avec données associées (AEAD)**. Un AEAD fournit simultanément :

- **Confidentialité** : Le message est chiffré.
- **Intégrité** : Toute modification du message chiffré est détectable.
- **Authentification** : Le message est authentifié comme provenant d'une source possédant la clé secrète.

Un schéma AEAD combine un mode de chiffrement (ex: ChaCha20, AES-GCM) avec un code d'authentification de message (MAC, ex: Poly1305, GMAC). Le résultat de l'opération de chiffrement est un **ciphertext** qui inclut à la fois le contenu chiffré et une **étiquette d'authentification (tag)**. Au déchiffrement, le tag est recalculé et comparé. S'ils ne correspondent pas, le déchiffrement échoue, signalant une corruption.

Les **données associées (Associated Data - AD)** sont des informations qui ne sont pas chiffrées mais dont l'intégrité est protégée par le tag. Elles servent à lier un ciphertext à son contexte (ex: un ID de capteur, une version de protocole, une adresse IP).

Pour ce TP, nous utiliserons **ASCON**, le lauréat de la compétition du NIST pour la cryptographie légère (Lightweight Cryptography - LWC) et désormais le standard officiel. Sa légèreté le rend particulièrement adapté aux dispositifs contraints comme les capteurs IIoT.

III. Environnement de Travail

Vous utiliserez Python 3. Assurez-vous d'installer la bibliothèque **`pyascon`** qui fournit une implémentation simple et directe de l'algorithme. Vous pourrez également utiliser **`tabulate`** pour formater joliment vos résultats dans le compte rendu.

```
pip install pyascon tabulate
```

IV. Travail à Réaliser

1. Simulation du Flux de Données

Dans un premier temps, nous allons simuler un capteur qui génère des mesures.

1. **Créez une fonction `generate_sensor_data(sensor_id: int) → dict`.**
   - Cette fonction doit retourner un dictionnaire Python contenant :
     - **`sensor_id`** : l'identifiant passé en paramètre.
     - **`temperature`** : une valeur flottante aléatoire simulant une température (par exemple, entre 15.0 et 30.0 °C).
     - **`timestamp`** : le timestamp UTC actuel (utilisez **`datetime.datetime.utcnow().isoformat()`**).
2. **Manipulation des données :**
   - Le protocole de communication transporte des chaînes de caractères (JSON), et les bibliothèques de cryptographie opèrent sur des **octets (bytes)**.
   - Dans votre script principal, appelez votre fonction pour générer un paquet de données.
   - Sérialisez le dictionnaire en une chaîne JSON.
   - Encodez cette chaîne JSON en un objet **`bytes`** en utilisant l'encodage **`UTF-8`**. C'est cet objet **`bytes`** que nous appellerons le *plaintext*.

```python
import json
import random
import datetime
import os

def generate_sensor_data(sensor_id: int) → dict:
    """
    Génère un dictionnaire de données simulées pour un capteur.

    TODO: Complétez cette fonction pour qu'elle retourne les données
    conformément à la spécification ci-dessus.
    """
    # Votre code ici
    pass

# --- Dans votre script principal ---
SENSOR_ID = 123
data_packet = generate_sensor_data(SENSOR_ID)
plaintext_json = json.dumps(data_packet, sort_keys=True)
plaintext_bytes = plaintext_json.encode('utf-8')

print(f"Données brutes : {data_packet}")
print(f"Plaintext (bytes) : {plaintext_bytes}")
```

> Pourquoi est-il souvent recommandé d'utiliser une représentation canonique du JSON (comme **`sort_keys=True`**) avant de le chiffrer, même si le chiffrement opère au niveau des octets ?

2. Protection des Données (Chiffrement)

Nous allons maintenant écrire la fonction qui sécurise notre *plaintext*.

1. **Définissez la fonction** `protect_data` avec la signature suivante : `protect_data(key: bytes, nonce: bytes, associated_data: bytes, plaintext: bytes) → bytes`
2. À l'intérieur de cette fonction, utilisez `ascon.encrypt`.
   - La variante recommandée pour notre cas d'usage est `'Ascon-128a'`.
   - La fonction retournera le ciphertext (qui contient le tag d'authentification).
3. **Préparation des éléments :**
   - **Clé (key) :** Une clé secrète partagée. Pour ce TP, générez une clé aléatoire de 16 octets (128 bits) avec `os.urandom(16)`.
   - **Nonce :** Un "Number used once". Il doit être unique pour chaque opération de chiffrement avec une même clé. Générez un nonce aléatoire de 16 octets avec `os.urandom(16)`.
   - **Associated Data (AD) :** Nous utiliserons l'ID du capteur. Comme l'ID est un entier, il doit être converti en `bytes`. Une méthode simple est `SENSOR_ID.to_bytes(4, 'big')`.

```python
import ascon

def protect_data(key: bytes, nonce: bytes, associated_data: bytes, plaintext: bytes) → bytes:
    """
    Chiffre et authentifie le plaintext en utilisant ASCON.

    TODO: Implémentez le chiffrement en utilisant la bibliothèque ascon.
    """
    # Votre code ici
    pass

# --- Dans votre script principal ---
KEY = os.urandom(16)
NONCE = os.urandom(16)
ad_bytes = SENSOR_ID.to_bytes(4, 'big') # Associated Data

ciphertext = protect_data(KEY, NONCE, ad_bytes, plaintext_bytes)

print(f"Ciphertext (bytes) : {ciphertext.hex()}")
print(f"Taille Plaintext : {len(plaintext_bytes)} octets")
print(f"Taille Ciphertext: {len(ciphertext)} octets")
```

> Observez la différence de taille entre le `plaintext` et le `ciphertext`. Quelle est la taille de cette différence en octets ? À quoi correspond-elle dans le schéma ASCON et pourquoi sa taille est-elle fixe quelle que soit la taille du message ?

> Expliquez la criticité de l'unicité du couple (Clé, Nonce). Que se passerait-il si vous chiffriez deux messages différents avec la même clé et le même nonce en utilisant un AEAD comme ASCON (ou AES-GCM) ?

> Le capteur peut redémarrer (panne de courant). Comment garantir que le compteur ne sera jamais réinitialisé à une valeur déjà utilisée ?

3. Déchiffrement et Vérification

Écrivons maintenant la fonction miroir pour déchiffrer et, surtout, vérifier l'intégrité.

1. **Définissez la fonction** `unprotect_data` avec la signature : `unprotect_data(key: bytes, nonce: bytes, associated_data: bytes, ciphertext: bytes) → bytes | None`
2. À l'intérieur, utilisez `ascon.decrypt`. Cette fonction lève une exception de type `ascon.AsconError` si la vérification du tag d'authentification échoue.
3. **Votre implémentation doit impérativement** utiliser un bloc `try ... except ascon.AsconError`.
   - En cas de succès, la fonction retourne le plaintext déchiffré en `bytes`.

- En cas d'échec (exception levée), elle doit retourner `None` pour indiquer que le message n'est pas authentique.

```
def unprotect_data(key: bytes, nonce: bytes, associated_data: bytes, ciphertext: bytes) → bytes | Non
    """
    Vérifie et déchiffre le ciphertext en utilisant ASCON.
    Retourne le plaintext en bytes si l'opération réussit, sinon None.

    TODO: Implémentez le déchiffrement et la gestion d'erreur.
    """
    # Votre code ici
    pass

# --- Dans votre script principal ---
decrypted_bytes = unprotect_data(KEY, NONCE, ad_bytes, ciphertext)

if decrypted_bytes:
    decrypted_json = decrypted_bytes.decode('utf-8')
    decrypted_data = json.loads(decrypted_json)
    print(f"Succès du déchiffrement : {decrypted_data}")
    assert decrypted_data == data_packet
else:
    print("ÉCHEC : Le message n'a pas pu être authentifié/déchiffré.")
```

4. Expérimentation et Analyse

Cette dernière étape est cruciale pour observer le comportement du système face à des attaques.

a. Corruption du Ciphertext

Simulons un attaquant qui modifie un seul bit du message en transit.

1. Prenez le `ciphertext` valide généré à l'étape 2.
2. Convertissez-le en un objet mutable, un `bytearray`.
3. Modifiez un octet. Par exemple, "flippez" les bits d'un octet au milieu du message en utilisant l'opérateur XOR : `tampered_ciphertext_array[20] ^= 0b00000001`.
4. Appelez `unprotect_data` avec ce ciphertext modifié.
5. Vérifiez et documentez le résultat.

b. Corruption des Données Associées

Simulons maintenant un attaquant qui intercepte un message destiné au capteur `123` et tente de le faire passer pour un message du capteur `456`.

1. Utilisez le `ciphertext` original et valide.
2. Créez de nouvelles données associées : `wrong_ad_bytes = (456).to_bytes(4, 'big')`.
3. Appelez `unprotect_data` en utilisant la clé, le nonce et le ciphertext corrects, mais avec ces `wrong_ad_bytes`.
4. Vérifiez et documentez le résultat.

> Sur la base de l'expérience 2, décrivez avec vos mots le rôle de sécurité rempli par les Données Associées (AD). Donnez un autre exemple concret où les AD seraient indispensables pour prévenir une attaque logique.

> Le schéma AEAD, tel qu'implémenté ici, protège-t-il contre les **attaques par rejeu (replay attacks)**, où un attaquant capture un message valide et le renvoie plus tard ? Si non, quelle information présente dans notre message simulé (`data_packet`) pourrait être utilisée, en combinaison avec une logique de maintien d'état côté serveur, pour détecter et rejeter de tels rejeux ? Justifiez votre réponse.

V. Compte Rendu à Rendre

Votre compte rendu, au format PDF, devra contenir :

1. **Vos réponses complètes et argumentées** aux questions 1, 2, 3, 4 et 5.
2. **Le code Python final, complet et commenté** de votre script. Le code doit être propre et suivre les bonnes pratiques.
3. **Un tableau récapitulatif** de vos expériences, présentant clairement les entrées et les sorties observées. Vous pouvez utiliser la bibliothèque `tabulate` pour un rendu professionnel.

Exemple de tableau de résultats :

Expérience	Plaintext Size (bytes)	Ciphertext Size (bytes)	Données d'Entrée pour `unprotect`	Résultat de `unprotect`
Cas Nominal	78	94	(KEY, NONCE, ad_bytes, ciphertext)	Succès (Plaintext retourné)
Corruption Ctxt	78	94	(KEY, NONCE, ad_bytes, tampered_ciphertext)	Échec (None)
Corruption AD	78	94	(KEY, NONCE, wrong_ad, ciphertext)	Échec (None)

Votre évaluation portera autant sur la qualité de votre code et de vos expérimentations que sur la profondeur et la clarté de vos justifications théoriques.