Cryptographie **I**ndustrielle **A**vancée // TP1

PKI et TLS

Loïc Rouquette

I. Objectifs

Ce TP vise à dépasser la simple configuration de services pour aborder l'implémentation et l'analyse critique de la sécurité au niveau applicatif. Les étudiants devront :

1. **Mettre en œuvre une mini-PKI (Public Key Infrastructure)** pour gérer le cycle de vie des certificats.
2. **Développer un canal de communication sécurisé en TLS 1.3** entre deux applications (client/serveur) en Python, en implémentant l'authentification mutuelle (mTLS).
3. **Analyser quantitativement l'impact des suites de chiffrement (cipher suites)** sur la latence de la connexion et le débit, afin de justifier un choix éclairé selon des contraintes spécifiques (performance vs. sécurité).
4. **Protéger un flux de données applicatif en utilisant un schéma de chiffrement authentifié léger (AEAD)**, simulant les besoins d'un environnement à ressources contraintes comme l'IIoT (Industrial Internet of Things).

II. Pré-requis

- Maîtrise de l'environnement Linux et des scripts shell.
- Programmation Python, notamment en réseau (`socket`).
- Compréhension des concepts de cryptographie asymétrique, des certificats X.509 et des principes de TLS.
- OpenSSL et Python 3 (avec la bibliothèque `cryptography`) installés.

III. Mise en place d'une PKI

L'objectif est de simuler une autorité de certification (CA) interne qui signera les certificats pour les serveurs et les clients (capteurs IIoT).

1. Création de l'Autorité de Certification (CA)

Générez la clé privée et le certificat auto-signé de votre CA. Ce certificat racine sera la base de confiance pour tous les autres certificats.

1. **Clé privée de la CA** : Créez une clé RSA 4096 bits protégée par une passphrase.
2. **Certificat racine auto-signé** : Créez un certificat X.509 auto-signé valide pour 10 ans.
3. **Vérification du certificat**:
    1. **Affichez le certificat**: Vous pouvez maintenant afficher votre certificat et regarder les informations qu'il contient.
    2. **Vérifiez que la clé privée correspond au certificat**: Utilisez OpenSSL pour vérifier que la clé privée correspond au certificat.

> **Discussion :** Pourquoi est-il crucial de protéger la clé `ca.key` avec une passphrase (`-aes256`) dans un environnement de production ?

2. Émission des Certificats Serveur et Client

Votre CA va maintenant émettre des certificats pour le serveur collecteur de données et pour un client simulant un capteur IIoT.

1. **Clé privée du serveur** : Générez une clé RSA 2048 bits pour le serveur.
2. **Demande de signature de certificat (CSR) pour le serveur** : Créez une signature de certificat (CSR) pour le serveur.
3. **Signature du CSR avec la CA** : Utilisez votre CA pour signer le CSR du serveur et générer un certificat X.509 valide pour 1 an.
4. **Répétez les étapes 1-3 pour le client (capteur IIoT)**. Attention à utiliser des chiffrements adaptés pour un capteur IIoT.

À la fin de cette étape, vous disposez d'une chaîne de confiance complète.

> **Discussion :** Quels sont les risques associés à l'utilisation de certificats auto-signés dans un environnement de production ?

IV. Implémentation d'un canal sécurisé avec mTLS

L'objectif est de créer une application client-serveur basique et de la sécuriser avec TLS 1.3 et une authentification mutuelle (le serveur vérifie le client, et vice-versa).

Serveur Sécurisé (Python)

Le serveur écoute les connexions, exige un certificat client valide signé par votre CA et affiche les données reçues. Une fois que la connexion est établie, le serveur et le client affichent la suite de chiffrement négociée.

V. Analyse des performances des cipher suites

L'objectif est de mesurer et comparer l'impact du choix de la cipher suite sur les performances, une contrainte essentielle dans l'IIoT.

Exercice :

1. Modifiez le contexte SSL du serveur et du client pour forcer l'utilisation de cipher suites spécifiques en utilisant context.set_ciphers(...).
2. Mettez en place un micro-benchmark pour mesurer deux métriques :
   - Latence du Handshake : Le temps nécessaire pour établir la connexion sécurisée.
   - Débit : Le temps nécessaire pour transférer un volume de données important (ex: 10 Mo).
3. Comparez au minimum les deux cipher suites TLS 1.3 suivantes :
   - `TLS_AES_256_GCM_SHA384` (Très haute sécurité, basée sur AES-GCM)
   - `TLS_CHACHA20_POLY1305_SHA256` (Haute sécurité, souvent plus performante sur les plateformes sans accélération matérielle AES)
4. Présentez vos résultats dans un tableau et justifiez le choix d'une suite pour un scénario où la latence est critique et un autre où la puissance de calcul du capteur est très limitée.

> Il se peut que vous ayez des difficultés à forcer certaines suites de chiffrement en fonction de votre version d'OpenSSL et de Python. Si la sélection des cipher suites ne fonctionne pas correctement avec TLS 1.3 vous pouvez basculer sur TLS 1.2.

VI. Chiffrement authentifié léger au niveau applicatif

Dans l'IIoT, on peut avoir besoin d'une sécurité de bout en bout au niveau du message (end-to-end), même si les relais intermédiaires terminent la connexion TLS. On utilise alors un chiffrement léger directement sur la donnée.

L'objectif est d'utiliser ChaCha20-Poly1305, un AEAD (Authenticated Encryption with Associated Data) performant, pour protéger les messages envoyés via le canal TLS.

Exercice :

1. Installez la bibliothèque `cryptography` dans votre environnement Python (si vous utilisez le Dockerfile, `cryptography` est déjà inclus).
2. Intégrez le chiffrement/déchiffrement AEAD dans votre application client/serveur.

> **Discussion :** Quels sont les avantages de cette double couche de sécurité (TLS + AEAD applicatif) dans un contexte IIoT ?

> **Discussion :** Quelles sont les contraintes liées à la gestion des nonces ?