

Projet de cryptographie

FÉVRIER 2025 - AVRIL 2025

Projet réalisé par :

CATALA Alexandre
VERNANCHET Louis

<https://github.com/AviMcCartney/Cryptographie-Projet---CATALA-Alexandre-VERNANCHET-Louis.git>

Table des matières

I.	Introduction	1
II.	Prérequis	2
III.	Réalisation du projet	3
A.	Validation d'un certificat d'autorité racine	3
1.	Lire le contenu d'un certificat	3
2.	Extraire la clef publique du certificat auto-signé et la vérification de la signature	5
3.	Affichage du sujet et de l'émetteur du certificat	7
4.	Vérifier l'extension Key Usage	7
5.	Vérifier la période de validité	9
6.	Extraire l'algorithme de signature du certificat ainsi que la signature	11
B.	Validation d'une chaîne de certificats	13
1.	Validation de la chaîne de certificats récursive	13
2.	Signature RSA en utilisant une API de calcul sur les grands nombres	15
3.	Effectuer la vérification de signature ECDSA en utilisant une librairie de calcul sur courbe elliptique	17
4.	Ajuster la vérification de l'extension KeyUsage suivant le niveau de certificat	21
5.	Effectuer la vérification de l'extension BasicConstraints	23
C.	Vérification du statut de révocation	27
1.	Vérification du Statut de Révocation des Certificats via les CRL	27
2.	Ajouter la vérification du statut de révocation en utilisant le protocole OCSP s'il est disponible pour un certificat donné	30
3.	Ajouter un mécanisme de cache pour ne pas télécharger une CRL s'il elle n'a pas été mise à jour ...	31
4.	Question Bonus	33
IV.	Conclusion	35
V.	Bibliographie	37

Table des figures

Figure 1 : Code implémenté pour le format DER	3
Figure 2 : Code implémenté pour le format PEM	4
Figure 3 : Affichage dans le terminal des informations des certificats de test	5
Figure 4: Extraction de la clé publique.....	5
Figure 5: Méthode pour la vérification de la signature	6
Figure 6 : Vérification de la signature sur un CA root.....	6
Figure 7: Vérification de la signature sur un sous CA	6
Figure 8 : Affichage des informations sur le sujet et l'émetteur.....	7
Figure 9 : Documentation officielle et commentaires	7
Figure 10 : Méthodes utilisées pour afficher les informations de sujet/émetteur.....	7
Figure 11 : Documentation pour la méthode getKeyUsage	8
Figure 12 : Fonction pour vérifier les key usage dans un certificat.....	8
Figure 13 : Vérification de la fonction sur un CA root.....	9
Figure 14 : Fonction pour vérifier la date de validité.....	9
Figure 15 : Test de la fonction dans le main.java	9
Figure 16 : Validation de la date de validité	10
Figure 17 : Certificat invalide	10
Figure 18 : Erreur lors de la vérification d'un certificat expiré	10
Figure 19 : Exemple de sortie pour un certificat valide	12
Figure 20 : Fonction pour vérifier l'algorithme et la signature	12
Figure 21 : Commande insérée comme demandé dans le sujet	12
Figure 22 : Résultat obtenu	12
Figure 23 : Étapes de debug	14
Figure 24 : Extrait de la fonction récursive avec la condition d'arrêt	14
Figure 25 : Implémentation de la signature avec BigInteger dans la fonction signatureRSA avec BigInteger	16
Figure 26 : Implémentation de la comparaison des hahs dans la fonction signatureRSA avec BigInteger	16
Figure 27 : Implémentation des solutions pour le PKCS#1 et la structure correcte	16
Figure 28 : Test avec un mauvais certificat	17
Figure 29 : Import de bouncyCastle	17
Figure 30 : Vérification avec un certificat invalide	18
Figure 31 : Vérification avec un certificat valide	18
Figure 32 : Vérification de la chaîne de certificat	18
Figure 33 : Ajout de Bouncy Castle et parcours de la chaîne de certificats	19
Figure 34 : Vérification de la clé publique du signataire	19
Figure 35 : Identification de la courbe elliptique	20
Figure 36 : Vérification de la signature	20
Figure 37: Vérification du point	21
Figure 38 : Détection du rôle de certificat	22
Figure 39 : Vérification des Key Usage	22
Figure 40 : Affichage avec messages de debug dans le terminal	23
Figure 41: Affichage avec la fonction finale	23
Figure 42 : Documentation pour basicconstraints.....	23
Figure 43 : vérification du rôle du certificat.....	24
Figure 44 : Affichage du test pour un CA root	25
Figure 45 : Affichage pour la vérification d'une chaîne de certificats.....	25

Figure 46 : Modifications apportés aux précédentes fonctions pour optimiser le main.....	26
Figure 47 : Implémentation plus claire dans le main.....	26
Figure 48 : Extraction de la CRL	27
Figure 49 Détails sur la méthode URL.....	28
Figure 50 : méthode pour télécharger la CRL	28
Figure 51 : Méthode pour la vérification des révocations	29
Figure 52 : Test sur une chaîne de certificats	29
Figure 53 : Test sur un CA root	29
Figure 54 : Vérification avec OCSP si disponible ou CRL sinon.....	30
Figure 55 : Méthode de vérification avec OCSP.....	30
Figure 56 : Affichage du test avec vérification OCSP	30
Figure 57 : Vérification de la présence de la CRL sur disque.....	31
Figure 58 : Charger la CRL depuis le disque si elle est présente	32
Figure 59 : Télécharger la CRL si non présente sur le disque	32
Figure 60 : Affichage du téléchargement des CRL.....	33
Figure 61 : Vérification de la persistence des CRL sur disque	33
Figure 62 : Architecture du projet.....	36
Figure 63 : Extrait de la Javadoc accessible depuis le index.html.....	36

I. Introduction

Ce projet consiste à développer un logiciel d'analyse d'une chaîne de certificats X.509 afin d'évaluer sa validité et sa sécurité. Ce module pourrait être intégré à une pile TLS et servira à vérifier la confiance accordée aux certificats émis.

L'implémentation doit prendre en charge les algorithmes RSA et ECDSA pour la vérification des signatures de certificats. Les certificats analysés seront stockés individuellement sous forme de fichiers et devront être compatibles avec les formats DER ainsi que DER encodé en base 64 (PEM).

Il est crucial d'assurer la correspondance entre le sujet et l'émetteur de chaque certificat afin de garantir l'intégrité de la chaîne. Un certificat ne peut être considéré comme valide que si l'ensemble des certificats formant la chaîne sont correctement reliés et signés par une autorité reconnue.

Les spécifications du programme seront détaillées de manière progressive. Pour garantir la robustesse du module, des tests seront effectués sur des certificats valides et invalides. Le TLS reposant sur ce système, s'il est faillible alors la communication TLS ne sera pas viable car pas sécurisé dès le début.

L'objectif final du projet est de valider la chaîne de trois certificats du site [TBS Certificates](#) et de [Le Monde](#) pour s'assurer de leur intégrité et de leur conformité aux standards de sécurité.

II. Prérequis

Avant de démarrer le projet, certaines étapes préliminaires sont essentielles :

- Télécharger les certificats de chaque site et chaque étape de la chaîne de certificats aux formats DER et PEM, afin de les utiliser ultérieurement dans le projet. Nous avons des exemples de mauvais certificats sur le site [badssl.com](#), et des certificats valides sur le site de [Le Monde](#) et [TBS Certificates](#).
- Créer un repository GitHub pour faciliter le travail en mode incrémental et collaboratif en binôme.
- Le projet étant axé sur l'analyse et la validation de **certificats X.509**, le **Java** a été choisi comme langage principal, conformément aux spécifications du sujet. Java est particulièrement adapté à ce type de projet grâce à sa **puissante API de gestion des certificats et des clés cryptographiques** intégrée dans le package `java.security`.

III. Réalisation du projet

A. Validation d'un certificat d'autorité racine

1. Lire le contenu d'un certificat

L'objectif était de permettre la lecture de certificats X.509 en deux formats courants :

- **DER** (Distinguished Encoding Rules) : Format binaire standardisé.
- **PEM** (Privacy-Enhanced Mail) : Format encodé en Base64, souvent utilisé dans les fichiers .pem, .crt ou .cer.

Nous avons implémenté deux méthodes distinctes pour traiter ces formats dans la classe *ValidateCert*.

Lecture d'un certificat au format DER :

Le format DER étant binaire, il doit être lu en mode fichier brut avant d'être transformé en objet X509Certificate. On utilise alors FileInputStream pour lire le fichier et CertificateFactory pour le parser.

```
public class ValidateCert {  
    1 usage  ↗ CATALA Alexandre  
    public static X509Certificate affichage_DER(String filePath) throws Exception {  
        CertificateFactory certFactory = CertificateFactory.getInstance( type: "X.509");  
        try (InputStream inStream = new FileInputStream(filePath)) {  
            return (X509Certificate) certFactory.generateCertificate(inStream);  
        }  
    }  
}
```

Figure 1 : Code implémenté pour le format DER

- ***CertificateFactory.getInstance("X.509")*** : Crée un analyseur de certificats X.509.
- ***FileInputStream(filePath)*** : Ouvre le fichier en mode binaire.
- ***generateCertificate(inStream)*** : Convertit le fichier en un objet X509Certificate.

Lecture d'un certificat au format PEM :

Le format PEM contient des données encodées en Base64 entourées de balises (-
----BEGIN CERTIFICATE---- et ----END CERTIFICATE----).

Ces balises doivent être supprimées avant la conversion. Nous avons donc lu le fichier en tant que texte, extrait les données encodées et les avons décodés en binaire avant de les parser avec CertificateFactory.

```
1 usage  ▲ CATALA Alexandre
public static X509Certificate affichage_PEM(String filePath) throws Exception {
    CertificateFactory certFactory = CertificateFactory.getInstance( type: "X.509");
    String pemContent = new String(Files.readAllBytes(Paths.get(filePath)));
    String base64Cert = pemContent.replaceAll( regex: "----BEGIN CERTIFICATE----", replacement: "")
        .replaceAll( regex: "----END CERTIFICATE----", replacement: "")
        .replaceAll( regex: "\\s", replacement: "");
    byte[] decoded = Base64.getDecoder().decode(base64Cert);
    try (InputStream inStream = new ByteArrayInputStream(decoded)) {
        return (X509Certificate) certFactory.generateCertificate(inStream);
    }
}
```

Figure 2 : Code implémenté pour le format PEM

- Files.readAllBytes(Paths.get(filePath)) : Lit le fichier PEM sous forme de texte.
- Suppression des balises ----BEGIN CERTIFICATE---- et ----END CERTIFICATE---- et des espaces blancs.
- Décodage en Base64 pour obtenir un fichier binaire équivalent au format DER.
- Conversion en X509Certificate via generateCertificate().

Nous avons rencontré plusieurs problèmes lors de l'élaboration de ces premières fonctions tel que : chemin du fichier introuvable, format incorrect, erreurs de lectures ... Mais nous avons réglé ces erreurs avant de continuer le développement des autres fonctions et avons réussi à obtenir le résultat attendu.

```
Test du certificat DER :  
-----  
Sujet : CN=*.lemonde.fr  
Émetteur : CN=GlobalSign Atlas R3 DV TLS CA 2024 Q4, O=GlobalSign nv-sa, C=BE  
Valide du : Tue Jan 07 21:55:17 CET 2025  
Valide jusqu'à : Sun Feb 08 21:55:16 CET 2026  
Numéro de série : 2109305907229186487112996407600886329  
Algorithme de signature : SHA256withRSA  
-----  
  
Test du certificat PEM :  
-----  
Sujet : CN=*.lemonde.fr  
Émetteur : CN=GlobalSign Atlas R3 DV TLS CA 2024 Q4, O=GlobalSign nv-sa, C=BE  
Valide du : Tue Jan 07 21:55:17 CET 2025  
Valide jusqu'à : Sun Feb 08 21:55:16 CET 2026  
Numéro de série : 2109305907229186487112996407600886329  
Algorithme de signature : SHA256withRSA  
-----  
  
Process finished with exit code 0
```

Figure 3 : Affichage dans le terminal des informations des certificats de test

2. Extraire la clef publique du certificat auto-signé et la vérification de la signature

Nous avons mis en place une méthode permettant d'extraire la clé publique d'un certificat X.509 et de vérifier la validité de sa signature.

Nous utilisons la méthode `getPublicKey()` pour afficher la clé publique contenue dans le certificat :

```
System.out.println("Clé publique : " + cert.getPublicKey());
```

Figure 4: Extraction de la clé publique

Nous utilisons ensuite la méthode verify() pour confirmer la validité de la signature :

```
1 usage  ↗ Sanqyh
public static boolean verifierSignature(X509Certificate cert) {
    try {
        PublicKey publicKey = cert.getPublicKey();
        cert.verify(publicKey);
        return true;
    } catch (Exception e) {
        System.err.println("Échec de la vérification de la signature: " + e.getMessage());
        return false;
    }
}
```

Figure 5: Méthode pour la vérification de la signature

Cette vérification fonctionne correctement sur les certificats auto-signés (root CA), car ils sont signés avec leur propre clé privée et peuvent être validés avec leur clé publique. En revanche, pour les certificats émis par une autorité de certification (CA), la vérification échoue si on utilise leur propre clé publique. Dans ce cas, il faut utiliser la clé publique du certificat de la CA qui a signé le certificat.

```
Test du certificat PEM :
-----
Sujet : CN=GlobalSign, O=GlobalSign, OU=GlobalSign Root CA - R3
Émetteur : CN=GlobalSign, O=GlobalSign, OU=GlobalSign Root CA - R3
Validé du : Wed Mar 18 11:00:00 CET 2009
Validé jusqu'à : Sun Mar 18 11:00:00 CET 2029
Numéro de série : 4835703278459759426209954
Algorithme de signature : SHA256withRSA
Clé publique : Sun RSA public key, 2048 bits
  params: null
  modulus: 25771087976912555723460212693216601925287656672099574380533157801820567
  public exponent: 65537
-----
 Signature valide.
```

Figure 6 : Vérification de la signature sur un CA root

```
-----
Sujet : CN=*.lemonde.fr
Émetteur : CN=GlobalSign Atlas R3 DV TLS CA 2024 Q4, O=GlobalSign nv-sa, C=BE
Validé du : Tue Jan 07 21:55:17 CET 2025
Validé jusqu'à : Sun Feb 08 21:55:16 CET 2026
Numéro de série : 2109305907229186487112996407600886329
Algorithme de signature : SHA256withRSA
Clé publique : Sun RSA public key, 2048 bits
  params: null
  modulus: 274622009481450931172122226062943018479937072335655967390799427466
  public exponent: 65537
-----
✖ Signature invalide.
Échec de la vérification de la signature: Signature does not match.
```

Figure 7 : Vérification de la signature sur un sous CA

Lors des tests, nous avons constaté que :

- La vérification verify() fonctionne avec les certificats auto-signés.
- Une erreur se produit lors de la vérification d'un certificat intermédiaire (sous CA) si sa propre clé publique est utilisée.
- La vérification nécessite la clé publique du certificat émetteur pour les certificats non-auto-signés.

3. Affichage du sujet et de l'émetteur du certificat

Nous avons mis en place une méthode permettant d'afficher le sujet ainsi que l'émetteur du certificat.

```
Sujet : CN=GlobalSign, O=GlobalSign, OU=GlobalSign Root CA - R3  
Émetteur : CN=GlobalSign, O=GlobalSign, OU=GlobalSign Root CA - R3
```

Figure 8 : Affichage des informations sur le sujet et l'émetteur

Les méthodes `getSubjectDN()` et `getIssuerDN()` étant dépréciées, nous avons utilisé `getSubjectX500Principal()` et `getIssuerX500Principal()` comme recommandé par [la documentation officielle](#) :

```
getSubjectDN()  
Denigrated, replaced by getSubjectX500Principal().  
  
getIssuerDN()  
Denigrated, replaced by getIssuerX500Principal().
```

Figure 9 : Documentation officielle et commentaires

Nous avons donc implémenté l'affichage des informations avec les méthodes appropriées :

```
System.out.println("Sujet : " + cert.getSubjectX500Principal());  
System.out.println("Émetteur : " + cert.getIssuerX500Principal());
```

Figure 10 : Méthodes utilisées pour afficher les informations de sujet/émetteur

4. Vérifier l'extension Key Usage

Nous avons mis en place une méthode permettant d'examiner l'extension `KeyUsage`, qui définit les usages autorisés pour une clé contenue dans un certificat.

D'abord nous avons cherché dans la documentation quelle méthode était disponible pour pouvoir effectuer cette tâche, et nous avons trouvé cette ressource :

getKeyUsage

```
public abstract boolean[] getKeyUsage()
```

Gets a boolean array representing bits of the KeyUsage extension, (OID = 2.5.29.15). The key usage extension defines the purpose (e.g., encipherment, signature, certificate signing) of the key contained in the certificate. The ASN.1 definition for this is:

```
KeyUsage ::= BIT STRING {
    digitalSignature      (0),
    nonRepudiation       (1),
    keyEncipherment      (2),
    dataEncipherment     (3),
    keyAgreement         (4),
    keyCertSign          (5),
    cRLSign              (6),
    encipherOnly         (7),
    decipherOnly         (8) }
```

RFC 3280 recommends that when used, this be marked as a critical extension.

Returns:

the KeyUsage extension of this certificate, represented as an array of booleans. The order of KeyUsage values in the array is the same as in the above ASN.1 definition. The array will contain a value for each KeyUsage defined above. If the KeyUsage list encoded in the certificate is longer than the above list, it will not be truncated. Returns null if this certificate does not contain a KeyUsage extension.

Figure 11 : Documentation pour la méthode getKeyUsage

Nous pouvons donc voir que getKeyUsage() est une liste de booléen représentant si une fonctionnalité est présente (bit = 1) ou absente (bit = 0), la position du bit dans la liste représentant les paramètres ci-dessus.

Nous avons donc élaborer notre fonction en utilisant la méthode getKeyUsage() et en parcourant ce tableau de booléen :

```
1 usage ~ CATALA Alexandre
public static void verifierKeyUsage(X509Certificate cert) {
    boolean[] keyUsage = cert.getKeyUsage();
    if (keyUsage != null) {
        System.out.println("Key Usage:");
        String[] usages = {"Digital Signature", "Non Repudiation", "Key Encipherment", "Data Encipherment", "Key Agreement",
        for (int i = 0; i < keyUsage.length; i++) {
            if (keyUsage[i]) {
                System.out.println("✓ " + usages[i]);
            }
        }
    } else {
        System.out.println("Key Usage non spécifié dans le certificat.");
    }
}
```

Figure 12 : Fonction pour vérifier les key usage dans un certificat

Lors des tests, nous avons confirmé que la fonction retourne bien les usages corrects des certificats. Par exemple, pour un certificat d'autorité racine (CA root), les usages Certificate Signing et CRL Signing sont activés, ce qui est cohérent avec son rôle.

```
Vérification de l'extension KeyUsage :  
Key Usage:  
✓ Certificate Signing  
✓ CRL Signing
```

Figure 13 : Vérification de la fonction sur un CA root

5. Vérifier la période de validité

Nous utilisons la méthode `checkValidity()` qui permet de vérifier si un certificat est toujours valide en fonction de sa date de création et d'expiration.

Cette méthode prend seule en compte la date de création du certificat et la date d'expiration pour déterminer si le certificat est valide.

```
public static boolean verifierDate(X509Certificate cert){ 1 usage  ± Sanqyh  
    try{  
        cert.checkValidity();  
        return true;  
    } catch (Exception e){  
        System.err.println("Échec de la vérification de la date" + e.getMessage());  
        return false;  
    }  
}
```

Figure 14 : Fonction pour vérifier la date de validité

```
System.out.println("\n==== Vérification de la validité ===");  
if (ValidateCert.verifierDate(cert)) {  
    System.out.println("Le certificat est valide en termes de date.");  
} else {  
    System.out.println("Le certificat est expiré ou non valide.");  
}
```

Figure 15 : Test de la fonction dans le main.java

Dans le cas où le certificat testé est valide on aura :

```
==> Vérification de la validité ==>
Le certificat est valide en termes de date.
```

Figure 16 : Validation de la date de validité

Faisons le test avec un certificat qui lui est invalide :

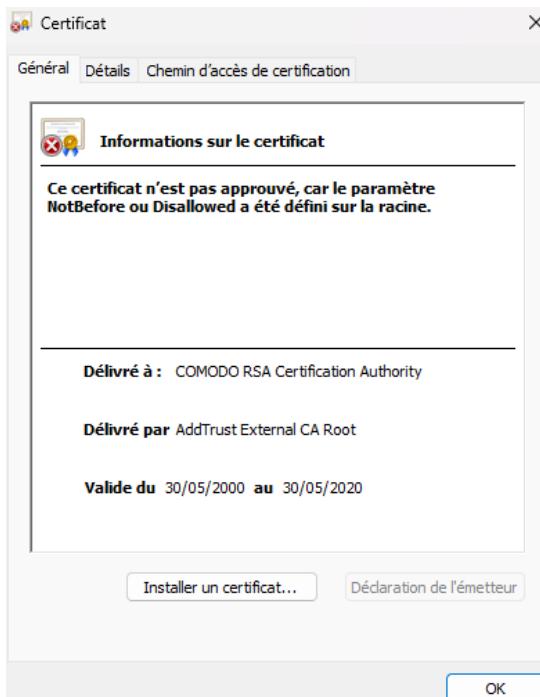


Figure 17 : Certificat invalide

Nous pouvons bien constater que celui-ci est expiré :

```
Sujet : CN=COMODO RSA Certification Authority, O=COMODO CA Limited, L=Salford, ST=Greater Manchester, C=GB
Emetteur : CN=AddTrust External CA Root, OU=AddTrust External TTP Network, O=AddTrust AB, C=SE
Valide du : Tue May 30 11:48:38 GMT+01:00 2000
Valide jusqu'à : Sat May 30 11:48:38 GMT+01:00 2020
```

Et notre fonction l'a bien détecté :

```
==> Vérification de la validité ==>
Le certificat est expiré ou non valide.
```

Figure 18 : Erreur lors de la vérification d'un certificat expiré

6. Extraire l'algorithme de signature du certificat ainsi que la signature

Nous avons mis en place une méthode permettant d'extraire l'algorithme de signature d'un certificat et de vérifier la validité de cette signature avec l'API cryptographique.

L'algorithme utilisé pour signer le certificat est extrait grâce à la méthode :

```
String algo = cert.getSigAlgName();
```

Cet algorithme peut être, par exemple, SHA256withRSA, ou ECDSA, selon la signature utilisée.

Nous créons une instance de l'objet Signature correspondant à l'algorithme extrait :

```
Signature sig = Signature.getInstance(algo);
```

Cette instance est utilisée pour vérifier la signature numérique. Nous fournissons ensuite les données originales signées (TBS Certificate) à l'objet Signature :

```
sig.update(cert.getTBSCertificate());
```

La partie TBS Certificate contient toutes les informations du certificat à l'exception de la signature elle-même. Enfin, nous vérifions la signature extraite du certificat :

```
boolean verified = sig.verify(signature);
```

Si la signature est correcte, verified sera true, sinon false. Nous affichons le résultat comme suit :

```
System.out.println("Algorithme de signature: " + algo);
System.out.println("Signature vérifiée: " + (verified ? "Valide" : "Invalide"));
```

Lors des tests, nous avons confirmé que l'algorithme et la signature sont validés correctement pour les certificats valides. En revanche, si la signature est corrompue ou l'algorithme incorrect, la vérification échoue avec un message d'erreur clair.

```
==> Vérification de la signature ==>
La signature du certificat est valide.
==> Vérification de la validité ==>
Le certificat est valide en termes de date.
==> Vérification de l'usage des clés ==>
Key Usage:
✓ Certificate Signing
✓ CRL Signing
==> Vérification de l'algorithme et de la signature ==>
Algorithme de signature: SHA256withRSA
Signature vérifiée: Valide
```

Figure 19 : Exemple de sortie pour un certificat valide

```
public static void vérifierAlgorithmeEtSignature(X509Certificate cert) { no usages à CATALA Alexandre
    try {
        String algo = cert.getSigAlgName();
        byte[] signature = cert.getSignature();
        PublicKey publicKey = cert.getPublicKey();
        Signature sig = Signature.getInstance(algo);
        sig.initVerify(publicKey);
        sig.update(cert.getTBSCertificate());
        boolean verified = sig.verify(signature);
        System.out.println("Algorithme de signature: " + algo);
        System.out.println("Signature vérifiée: " + (verified ? "Valide" : "Invalide"));
    } catch (Exception e) {
        System.err.println("Échec de la vérification de l'algorithme et de la signature: " + e.getMessage());
    }
}
```

Figure 20 : Fonction pour vérifier l'algorithme et la signature

```
validate-cert -format DER "C:\Users\Alexandre\OneDrive\Bureau\Cryptographie-Projet-
```

Figure 21 : Commande insérée comme demandé dans le sujet

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program F
==> Informations du Certificat ==>
Sujet : CN=GlobalSign, O=GlobalSign, OU=GlobalSign Root CA - R3
Émetteur : CN=GlobalSign, O=GlobalSign, OU=GlobalSign Root CA - R3
Date de début de validité : Wed Mar 18 11:00:00 CET 2009
Date de fin de validité : Sun Mar 18 11:00:00 CET 2029
Numéro de série : 4835703278459759426209954
==> Vérification de la signature ==>
La signature du certificat est valide.
==> Vérification de la validité ==>
Le certificat est valide en termes de date.
==> Vérification de l'usage des clés ==>
Key Usage:
✓ Certificate Signing
✓ CRL Signing
==> Vérification de l'algorithme et de la signature ==>
Algorithme de signature: SHA256withRSA
Signature vérifiée: Valide
```

Figure 22 : Résultat obtenu

B. Validation d'une chaîne de certificats

1. Validation de la chaîne de certificats récursive

Implémentation Initiale Sans Récursion

Dans un premier temps, nous avons développé une version non récursive de la validation de la chaîne de certificats. Cette approche utilisait une boucle itérative pour vérifier chaque certificat de la chaîne en le comparant au certificat supérieur (l'émetteur).

Toutefois, cette méthode présentait certaines limitations :

- Le Root CA était parfois traité comme un certificat classique, ce qui entraînait des erreurs lors de la validation.
- Il était difficile de gérer proprement l'arrêt des vérifications au bon moment.
- L'affichage des certificats intermédiaires n'était pas structuré.

Debugging et Analyse du Problème

Après avoir identifié ces limites, nous avons effectué un debugging approfondi en affichant les informations des certificats à chaque étape de la validation. Nous avons notamment observé :

- Que la boucle ne permettait pas toujours de vérifier la cohérence entre le sujet et l'émetteur correctement.
- Que la validation s'arrêtait parfois au mauvais endroit, notamment lorsque le certificat final était mal interprété comme auto-signé.

Ces observations nous ont amenés à repenser notre approche pour mieux respecter la hiérarchie des certificats.

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.4\lib\idea_rt.jar=54884" -Dfile.encoding=UTF-8
== Debug: Début de l'exécution ==
Nombre d'arguments reçus: 6
Argument [0]: validate-cert-chain
Argument [1]: -format
Argument [2]: PEM
Argument [3]: C:\Users\Alexandre\OneDrive\Bureau\Cryptographie-Projet---CATALA-Alexandre-VERNANCHET-Louis\projet crypto\Lemonde\PEM\GlobalSign_root_lemonde.pem.crt
Argument [4]: C:\Users\Alexandre\OneDrive\Bureau\Cryptographie-Projet---CATALA-Alexandre-VERNANCHET-Louis\projet crypto\Lemonde\PEM\Sousca_lemonde.pem.crt
Argument [5]: C:\Users\Alexandre\OneDrive\Bureau\Cryptographie-Projet---CATALA-Alexandre-VERNANCHET-Louis\projet crypto\Lemonde\PEM\lemonde.pem.crt

== Debug: Liste des certificats chargés ==
Certificat [0]: CN=GlobalSign, O=GlobalSign, OU=GlobalSign Root CA - R3
Certificat [1]: CN=GlobalSign Atlas R3 DV TLS CA 2024 Q4, O=GlobalSign nv-sa, C=BE
Certificat [2]: CN=*.lemonde.fr
```

Figure 23 : Étapes de debug

Passage à une Version Récursive

Pour corriger ces problèmes, nous avons remplacé l'itération par une approche récursive. La nouvelle version de la fonction suit le principe suivant :

1. Démarrer la vérification à partir du certificat feuille (Leaf Cert).
2. Remonter la chaîne en s'assurant que chaque certificat est signé par son émetteur direct.
3. S'arrêter uniquement lorsque le Root CA est atteint et vérifier qu'il est auto-signé.
4. Afficher les certificats intermédiaires de manière claire lors de la validation.

```
private static boolean verifierRecursive(List<X509Certificate> chain, int index) { 2 usages  ↳ CATALA Alexandre +1
    if (index == 0) {
        X509Certificate rootCert = chain.getFirst();
        try {
            rootCert.verify(rootCert.getPublicKey());
            System.out.println("Le certificat racine " + rootCert.getSubjectX500Principal() + " est auto-signé et valide.");
            return true;
        }
    }
}
```

Figure 24 : Extrait de la fonction récursive avec la condition d'arrêt

Grâce à cette implémentation récursive :

- La validation respecte mieux la hiérarchie des certificats.
- La gestion du Root CA est plus robuste, car il est traité différemment des autres certificats.
- La récursion permet de structurer proprement l'affichage et l'arrêt des vérifications.

2. Signature RSA en utilisant une API de calcul sur les grands nombres

L'algorithme RSA repose sur les opérations suivantes :

- Une signature numérique est obtenue en chiffrant un hash avec la clé privée :

$$S = H(M)^d \bmod N$$

Où :

- S est la signature chiffrée,
 - $H(M)$ est le hash du message (dans notre cas, le certificat),
 - d est l'exposant privé,
 - N est le module de la clé RSA.
-
- La vérification consiste à déchiffrer la signature avec la clé publique :

$$M = S^e \bmod N$$

Où :

- e est l'exposant public,
- M doit correspondre à $H(M)$ si la signature est valide.

L'API Java classique effectue ce calcul en interne via `java.security.Signature`, mais nous avons implémenté ce processus manuellement à l'aide de `BigInteger`.

Nous avons remplacé l'utilisation de l'API de cryptographie par une approche utilisant **les opérations mathématiques sur grands nombres** avec `BigInteger` :

1. Récupération des paramètres RSA de la clé publique :

- **Modulus (N) et Exposant public (e)** extraits de la clé publique du certificat.

2. Déchiffrement de la signature avec la formule RSA :

- La signature, représentée en `BigInteger`, est élevée à la puissance e , puis réduite modulo N .

$$M = S^e \bmod N$$

```
// Récupérer la signature chiffrée
byte[] signatureBytes = cert.getSignature();
BigInteger signature = new BigInteger( signum: 1, signatureBytes); // S (signature chiffrée)

// Effectuer le calcul de la signature RSA manuellement : M = S^e mod N
BigInteger decryptedMessage = signature.modPow(exponent, modulus);
```

Figure 25 : Implémentation de la signature avec BigInteger dans la fonction signatureRSA avec BigInteger

3. Comparaison du résultat avec le hash attendu :

- Nous avons extrait la partie finale du message déchiffré pour récupérer le **hash du certificat**.
- Nous avons ensuite comparé cet **hash avec le hash du certificat calculé via SHA-256**.

```
if (!Arrays.equals(extractedHash, expectedHash)) {
    System.err.println("Échec de la vérification de signature RSA pour " + cert.getSubjectX500Principal());
    return false;
}

System.out.println("Vérification de signature RSA réussie pour " + cert.getSubjectX500Principal());
```

Figure 26 : Implémentation de la comparaison des hahs dans la fonction signatureRSA avec BigInteger

L'un des défis rencontrés était le format des signatures RSA. Ces signatures suivent le standard PKCS#1 v1.5, ce qui signifie que le message déchiffré contient un padding en plus du hash.

Nous avons extrait uniquement les derniers octets du message déchiffré correspondant au hash. Une autre difficulté concernait la comparaison du hash extrait du message RSA avec celui du certificat. Nous avons utilisé SHA-256 pour recalculer le hash du certificat et comparer les valeurs.

```
// Récupérer le hash attendu du certificat
MessageDigest digest = MessageDigest.getInstance( algorithm: "SHA-256"); // Algorithme SHA-256
byte[] tbsCertificate = cert.getTBSCertificate(); // Structure signée du certificat
byte[] expectedHash = digest.digest(tbsCertificate); // H(M) attendu

// Extraire les derniers octets de decryptedMessage (car il contient un padding PKCS#1 v1.5)
byte[] decryptedBytes = decryptedMessage.toByteArray();
byte[] extractedHash = Arrays.copyOfRange(decryptedBytes, from: decryptedBytes.length - expectedHash.length, decryptedBytes.length);
```

Figure 27 : Implémentation des solutions pour le PKCS#1 et la structure correcte

Nous avons aussi réalisé des tests avec des certificats incorrects pour éprouver notre solution :

```
== Vérification de l'algorithme et de la signature ==  
== Vérification de la signature RSA avec BigInteger ==  
? La signature RSA est invalide.  
  
BUILD SUCCESSFUL in 312ms  
2 actionable tasks: 1 executed, 1 up-to-date  
◆ check de la vérification de la signature: Bad signature length: got 256 but was expecting 512  
◆ check de la vérification de la date: NotAfter: Sat May 30 11:48:38 GMT+01:00 2020  
◆ check de la vérification de l'algorithme et de la signature: Bad signature length: got 256 but was expecting 512  
? ◆ check de la vérification de signature RSA avec BigInteger.  
13:58:11: Execution finished ':org.example.Main.main()'.
```

Figure 28 : Test avec un mauvais certificat

3. Effectuer la vérification de signature ECDSA en utilisant une librairie de calcul sur courbe elliptique

Pour exploiter la bibliothèque Bouncy Castle, nous avons choisi de recréer le projet en utilisant Gradle. Cela permet d'ajouter facilement les dépendances nécessaires dans le fichier build.gradle, simplifiant ainsi la gestion des bibliothèques.

Cependant, nous avons rencontré des problèmes d'importation dans le fichier ValidateCert. Les dépendances ajoutées ne fonctionnaient pas à cause du cache d'IntelliJ IDEA, même après avoir reconstruit (build) le projet. Ce problème a été résolu en forçant la mise à jour des dépendances.

```
implementation ("org.bouncycastle:bcprov-jdk18on:1.75")  
implementation("org.bouncycastle:bcpkix-jdk18on:1.75")
```

Figure 29 : Import de bouncyCastle

Nous avons effectué des tests sur un certificat RSA valide en utilisant la méthode verify(), par la suite nous allons tester de vérifier la signature avec notre fonction ECDSA pour vérifier que la fonction détecte correctement que le format n'est pas celui attendu:

```
==> Véritation de l'algorithme et de la signature ==  
Algorithme de signature: SHA256withRSA  
Signature vérifiée: Valide  
  
==> Véritation de la signature ECDSA avec ECPoint ==  
? La signature ECDSA est invalide.  
  
BUILD SUCCESSFUL in 1s  
2 actionable tasks: 2 executed  
? Erreur : La clé publique du certificat n'est pas ECDSA.
```

Figure 30 : Vérification avec un certificat invalide

Quand on teste avec une signature ECDSA valide on obtient :

```
==> Véritation de la signature ECDSA avec ECPoint ==  
? La signature ECDSA est valide (calcul manuel avec ECPoint).
```

Figure 31 : Vérification avec un certificat valide

Dans un premier temps, nous avons fait une fonction qui permettait de valider un seul certificat. Ce qui nous a permis de faire un premier code fonctionnel de manière plus simple (en nous concentrant sur les calculs et l'utilisation de la librairie). Une fois cette première version faite et fonctionnelle, nous avons fait évoluer notre fonction pour qu'elle puisse valider à la fois un seul certificat mais aussi une chaîne de certificat.

```
==> Véritation des signatures ECDSA dans la chaîne de certificats ==  
? Véritation de signature ECDSA réussie pour CN=www.tbs-certificats.com, OID.2.5.4.97=NTRFR-440 443 810 00021, O=TBS CERTIFICATS, ST=Calvados, C=FR, OID.2.5.4.15=Private Organization  
? Véritation de signature ECDSA réussie pour CN=Sectigo Qualified Website Authentication CA E35, O=Sectigo (Europe) SL, C=ES  
? Véritation de signature ECDSA réussie pour CN=USERTrust ECC Certification Authority, O=The USERTRUST Network, L=Jersey City, ST>New Jersey, C=US  
Toutes les signatures ECDSA dans la chaîne sont valides !
```

Figure 32 : Vérification de la chaîne de certificat

Explication détaillée de la fonction *verifierSignatureECDSA* :

Afin de garantir l'authenticité des certificats de la chaîne, nous avons implémenté une fonction permettant de **vérifier les signatures ECDSA** à l'aide de la bibliothèque **Bouncy Castle**. Cette fonction assure que chaque certificat est correctement signé par l'entité émettrice.

1. Initialisation et préparation

Nous avons ajouté **Bouncy Castle** comme fournisseur de sécurité pour permettre la gestion des courbes elliptiques et des signatures ECDSA. La chaîne de certificats est ensuite **inversée** pour commencer la validation depuis le certificat **feuille (leaf)** jusqu'à l'autorité racine.

```
try {
    //Ajoute BouncyCastle comme fournisseur de sécurité
    Security.addProvider(new BouncyCastleProvider());

    if (IsChainNull(certChain)) {
        return false;
    }

    // Inverser la liste pour que la validation commence par le certificat leaf
    Collections.reverse(certChain);
```

Figure 33 : Ajout de Bouncy Castle et parcours de la chaîne de certificats

2. Vérification de la clé publique du signataire

Pour chaque certificat, nous identifions la clé publique qui doit être utilisée pour la validation :

- Si ce n'est pas le certificat racine, on utilise la **clé publique du certificat suivant** dans la chaîne.
- Si c'est le certificat racine, on utilise sa **propre clé publique**.
Nous nous assurons ensuite que cette clé publique est bien une **clé ECDSA**, et si ce n'est pas le cas, nous tentons une conversion via **Bouncy Castle**.

```
for (int i = 0; i < certChain.size(); i++) {
    X509Certificate cert = certChain.get(i);
    PublicKey issuerPublicKey;

    if (i < certChain.size() - 1) {
        //Si ce n'est pas le certificat racine, utiliser la clé publique du certificat suivant
        issuerPublicKey = certChain.get(i + 1).getPublicKey();
    } else {
        //Si c'est le certificat racine, utiliser sa propre clé publique
        issuerPublicKey = cert.getPublicKey();
    }
```

Figure 34 : Vérification de la clé publique du signataire

3. Identification de la courbe elliptique

L'algorithme de signature du certificat est analysé afin de déterminer l'**algorithme de hachage** utilisé (**SHA-256, SHA-384, SHA-512**). Ensuite, la **courbe elliptique** utilisée est identifiée parmi les courbes reconnues par **Bouncy Castle**. Si la courbe est inconnue, la validation échoue.

```
//Vérifier si la clé publique est bien ECDSA
if (!(issuerPublicKey instanceof ECPublicKey)) {
    try {
        KeyFactory keyFactory = KeyFactory.getInstance(algorithm: "EC", provider: "BC");
        issuerPublicKey = keyFactory.generatePublic(new X509EncodedKeySpec(issuerPublicKey.getEncoded()));
    } catch (Exception ex) {
        System.err.println("Échec de la conversion de la clé en ECPublicKey : " + ex.getMessage());
        return false;
    }
}
```

Figure 35 : Identification de la courbe elliptique

4. Extraction et validation de la signature

La signature ECDSA extraite du certificat est **décomposée en deux valeurs (r et s)**. Ensuite, le certificat est **haché** en fonction de l'algorithme déterminé précédemment, et la validation est effectuée en **calculant des valeurs intermédiaires** basées sur la courbe elliptique et les paramètres de la signature.

```
//Extraire la signature du certificat
byte[] signatureBytes = cert.getSignature();
ASN1InputStream asn1InputStream = new ASN1InputStream(signatureBytes);
ASN1Sequence asn1Sequence = (ASN1Sequence) asn1InputStream.readObject();
asn1InputStream.close();
BigInteger r = ((ASN1Integer) asn1Sequence.getObjectAt(i: 0)).getValue();
BigInteger s = ((ASN1Integer) asn1Sequence.getObjectAt(i: 1)).getValue();

//Calculer le hachage du certificat avec l'algorithme correspondant
MessageDigest digest = MessageDigest.getInstance(hashAlgorithm);
byte[] hash = digest.digest(cert.getTBSCertificate());
BigInteger e = new BigInteger(signum: 1, hash);
```

Figure 36 : Vérification de la signature

5. Vérification finale

Le point résultant du calcul elliptique est comparé avec la valeur r de la signature originale. Si les valeurs correspondent, la signature est valide. Sinon, la validation échoue et le certificat est considéré comme non fiable.

```
//Calculer P = u1 * G + u2 * Q
ECPPoint P = domainParams.getG().multiply(u1).add(Q.multiply(u2)).normalize();

//Vérification du point résultant
if (P.isInfinity()) {
    System.err.println("Échec de la vérification : Point à l'infini pour " + cert.getSubjectX500Principal());
    return false;
}

//Comparaison de r avec la coordonnée x de P modulo n
if (!P.getXCoord().toBigInteger().mod(domainParams.getN()).equals(r)) {
    System.err.println("Échec de la vérification de signature ECDSA pour " + cert.getSubjectX500Principal());
    return false;
}
```

Figure 37: Vérification du point

4. Ajuster la vérification de l'extension KeyUsage suivant le niveau de certificat

Dans un premier temps, notre fonction se contentait d'afficher les usages des clés disponibles dans un certificat. L'utilisateur devait alors analyser manuellement si les usages étaient conformes à son rôle (certificat racine, intermédiaire ou feuille).

Cette approche n'était pas optimale car :

- Elle demandait une vérification manuelle.
- Il était facile de rater une incohérence, surtout avec plusieurs certificats.
- Elle ne prenait en charge que des certificats individuels, sans gestion de chaînes.

Nous avons donc modifié la fonction pour :

- Gérer à la fois un certificat unique (Root CA) et une chaîne complète.
- Vérifier automatiquement si les usages clés sont corrects en fonction du rôle du certificat.
- Retourner une erreur si un certificat ne respecte pas les exigences.

Un certificat X.509 peut être de trois types :

Rôle	Explication	Key Usage requis
Certificat Feuille (Leaf)	Utilisé pour le chiffrement TLS ou l'authentification	Digital Signature (Index 0)
Certificat Intermédiaire	Délivre des certificats pour d'autres entités	Certificate Signing (Index 5)
Certificat Racine (Root CA)	Auto-signé, autorité de confiance	Certificate Signing (Index 5)

Notre nouvelle fonction identifie automatiquement le rôle de chaque certificat et s'assure qu'il possède les Key Usages adéquats.

Nous avons défini la logique suivante :

- **Si un seul certificat est présent**, il est considéré comme un **Root CA**.
- **Le dernier certificat de la chaîne est toujours un Leaf Cert.**
- **Tous les certificats intermédiaires doivent avoir l'usage "Certificate Signing".**

```
boolean isSingleRoot = (certChain.size() == 1);
```

Figure 38 : Détection du rôle de certificat

Pour chaque certificat :

- Si c'est un Leaf Cert (dernier de la chaîne) → Doit avoir "Digital Signature".
- Si c'est un Root ou un Intermédiaire → Doit avoir "Certificate Signing".

```
//Vérification finale en fonction du type de certificat
if (!isSingleRoot && i == 0 && !hasRequiredUsage) {
    System.err.println("Le certificat Leaf doit avoir 'Digital Signature'.");
    return false;
}
if ((isSingleRoot || i > 0) && !hasRequiredUsage) {
    System.err.println("Le certificat Intermédiaire/Root doit avoir 'Certificate Signing'.");
    return false;
}
```

Figure 39 : Vérification des Key Usage

```
== Vérification des KeyUsage dans la chaîne ==

Vérification du KeyUsage pour Certificat Leaf (site) : CN=*.lemonde.fr
Usages autorisés : Digital Signature | Key Encipherment |
KeyUsage validé

Vérification du KeyUsage pour Certificat Intermédiaire : CN=GlobalSign Atlas R3 DV TLS CA 2024 Q4, O=GlobalSign nv-sa, C=BE
Usages autorisés : Digital Signature | Certificate Signing | CRL Signing |
KeyUsage validé

Vérification du KeyUsage pour Certificat Root (racine) : CN=GlobalSign, O=GlobalSign, OU=GlobalSign Root CA - R3
Usages autorisés : Certificate Signing | CRL Signing |
KeyUsage validé

Tous les certificats ont un KeyUsage correct !
Tous les certificats de la chaîne ont des KeyUsage valides !
```

Figure 40 : Affichage avec messages de debug dans le terminal

```
== Vérification des KeyUsage dans la chaîne ==
Tous les certificats de la chaîne ont des KeyUsage valides !
```

Figure 41: Affichage avec la fonction finale

5. Effectuer la vérification de l'extension BasicConstraints

L'extension BasicConstraint est complémentaire à key usage. En effet, elle permet grâce à deux méthodes de savoir si le certificat a le droit ou non de signer d'autres certificat.

Methods	Modifier and Type	Method and Description
	boolean	<code>getCA()</code> Gets the CA flag.
	<code>java.math.BigInteger</code>	<code>getPathLen()</code> Gets the path length constraint.

Figure 42 : Documentation pour basicconstraints

Cas d'un Certificat Unique

Si un seul certificat est fourni, on suppose qu'il s'agit d'un Root CA. Dans ce cas :

- Aucune vérification supplémentaire n'est nécessaire pour ce certificat car il est auto-signé.
- Nous effectuons cependant un message de débogage pour informer l'utilisateur que le certificat est traité comme un Root CA.

Cas d'une Chaîne de Certificats

Dans le cas d'une chaîne de certificats :

1. Certificat Leaf (dernier certificat) :

- o getCA() doit être false.
- o getPathLen() doit être égal à 0 car un Leaf ne doit pas pouvoir signer d'autres certificats.
- o Si ces conditions ne sont pas remplies, l'utilisateur reçoit un message d'erreur.

2. Certificat Intermédiaire :

- o Le certificat intermédiaire doit pouvoir signer d'autres certificats. Ainsi :
 - getCA() doit être true.
 - getPathLen() doit être suffisamment élevé pour permettre la signature des certificats suivants.

3. Certificat Root (s'il existe) :

- o Aucune vérification n'est effectuée sur le certificat racine, car il est supposé être auto-signé et est déjà validé par sa position dans la chaîne.

```
// Déterminer le rôle du certificat
boolean isRoot = isSingleRoot || i == certChain.size() - 1;
boolean isLeaf = !isSingleRoot && i == 0;
boolean isLastInterm = (i == certChain.size() - 2);

int basicConstraints = cert.getBasicConstraints();

if (isLeaf) {
    // Vérification que le certificat Leaf n'est pas un CA
    if (basicConstraints != -1) {
        System.err.println("Erreur : Le certificat Leaf ne doit pas être un CA.");
        return false;
    }
} else {
    // Vérification que le certificat est un CA
    if (basicConstraints == -1) {
        System.err.println("Erreur : Le certificat " + cert.getSubjectX500Principal() +
                           " n'est pas un CA, mais il est dans la chaîne de certification.");
        return false;
    }
}
```

Figure 43 : vérification du rôle du certificat

On teste notre solution sur un certificat unique dans un premier temps, on considéreras qu'un certificat unique sera toujours un root CA.

```
== Vérification de Basic Constraints ==

? Vérification de Basic Constraints pour Certificat Root : CN=GlobalSign, O=GlobalSign, OU=GlobalSign Root CA - R3
? Basic Constraints validé ?

? Tous les certificats ont des Basic Constraints corrects !
Le certificat respecte les Basic Constraints !
```

Figure 44 : Affichage du test pour un CA root

On voit bien que le test se révèle positif, on test alors aussi sur une chaîne de certificats et on obtient :

```
== Vérification des Basic Constraints dans la chaîne ==

Vérification de Basic Constraints pour Certificat Leaf : CN=*.lemonde.fr
Basic Constraints validé

Vérification de Basic Constraints pour Certificat Intermediaire : CN=GlobalSign Atlas R3 DV TLS CA 2024 Q4, O=GlobalSign nv-sa, C=BE
Basic Constraints validé

Vérification de Basic Constraints pour Certificat Root : CN=GlobalSign, O=GlobalSign, OU=GlobalSign Root CA - R3
Basic Constraints validé

Tous les certificats ont des Basic Constraints corrects !
Tous les certificats de la chaîne respectent les Basic Constraints !
```

Figure 45 : Affichage pour la vérification d'une chaîne de certificats

Pour améliorer l'efficacité et la lisibilité du code, nous avons effectué les modifications suivantes :

Suppression des répétitions

Avant, nous créions une liste `List<X509Certificate>` à chaque fois que nous traitions les certificats, ce qui entraînait de la duplication de code. Nous avons désormais centralisé cette gestion dans une fonction.

Centralisation des Vérifications

Les vérifications des Key Usage et des Basic Constraints sont désormais gérées dans une méthode unique, `verifierProprietesCertificat`, qui permet de centraliser toutes les vérifications des certificats.

Meilleure gestion des erreurs

Avant, chaque erreur était gérée de manière dispersée dans main(). Désormais, nous utilisons des fonctions dédiées pour gérer proprement les erreurs et afficher des messages explicites.

En restructurant le code, nous avons rendu main() plus clair et concis. Les différentes étapes du traitement sont maintenant encapsulées dans des méthodes bien définies, ce qui améliore la lisibilité et la maintenance du code.

```
private static void verifierSignature(List<X509Certificate> certs) { 2 usages new *
    String sigAlg = certs.getFirst().getSigAlgName().toUpperCase();

    if (sigAlg.contains("RSA")) {
        System.out.println("\n==== Vérification de la signature RSA ===");
        if (ValidateCert.verifierSignatureRSA_BigInteger(certs)) {
            System.out.println("La signature RSA est valide.");
        } else {
            System.err.println("La signature RSA est invalide.");
        }
    } else if (sigAlg.contains("ECDSA")) {
        System.out.println("\n==== Vérification de la signature ECDSA ===");
        if (ValidateCert.verifierSignatureECDSA(certs)) {
            System.out.println("La signature ECDSA est valide.");
        } else {
            System.err.println("La signature ECDSA est invalide.");
        }
    } else {
        System.err.println("Algorithm de signature non supporté : " + sigAlg);
    }
}
```

```
private static void verifierProprietesCertificat(List<X509Certificate> certs) {
    System.out.println("\n==== Vérification des KeyUsage ===");
    if (ValidateCert.verifierKeyUsage(certs)) {
        System.out.println("Tous les certificats ont des KeyUsage valides !");
    } else {
        System.err.println("Erreur : Un ou plusieurs certificats ont un KeyUsage invalide.");
    }

    System.out.println("\n==== Vérification des Basic Constraints ===");
    if (ValidateCert.verifierBasicConstraints(certs)) {
        System.out.println("Tous les certificats respectent les Basic Constraints.");
    } else {
        System.err.println("Erreur : Un ou plusieurs certificats ont des Basic Constraints invalides.");
    }
}
```

Figure 46 : Modifications apportées aux précédentes fonctions pour optimiser le main

```
//Vérification de la signature
verifierSignature(certChain);

//Vérification des propriétés des certificats
verifierProprietesCertificat(certChain);
```

Figure 47 : Implémentation plus claire dans le main

C. Vérification du statut de révocation

1. Vérification du Statut de Révocation des Certificats via les CRL

Nous avons mis en place une solution en trois étapes :

1. Extraction de l'URL de la CRL à partir du certificat (via l'extension CRL Distribution Points)
2. Téléchargement de la CRL à partir de cette URL
3. Vérification de la révocation du certificat en consultant la CRL téléchargée

Les Root CA ne publient généralement pas de CRL. Par conséquent :

- Aucune tentative de récupération de CRL n'est effectuée pour ces certificats
- Une vérification spécifique est faite pour détecter si un certificat est un Root CA

La première étape consiste à extraire l'URL de la CRL à partir du certificat. Cette URL se trouve dans l'extension CRL Distribution Points, qui est encodée en ASN.1.

Logique de l'extraction

- On récupère l'extension **CRL Distribution Points** du certificat.
- On la **décode en ASN.1** pour en extraire l'URL.
- Si aucune extension CRL n'est trouvée, un message d'erreur est affiché.

```
public static String extraireCRLDistributionPoint(X509Certificate cert) { 1 usage new *
    try {
        byte[] crlBytes = cert.getExtensionValue(Extension.cRLDistributionPoints.getId());
        if (crlBytes == null) {
            System.err.println("Aucune extension CRL trouvée");
            return null;
        }

        // Décodage ASN.1
        try (ASN1InputStream asn1InputStream = new ASN1InputStream(new ByteArrayInputStream(crlBytes))) {
            ASN1OctetString octetString = ASN1OctetString.getInstance(asn1InputStream.readObject());
            try (ASN1InputStream asn1Stream2 = new ASN1InputStream(new ByteArrayInputStream(octetString.getOctets()))) {
                CRLDistPoint crlDistPoint = CRLDistPoint.getInstance(asn1Stream2.readObject());

                for (DistributionPoint dp : crlDistPoint.getDistributionPoints()) {
                    DistributionPointName dpn = dp.getDistributionPoint();
                    if (dpn != null && dpn.getType() == DistributionPointName.FULL_NAME) {
                        for (GeneralName gn : GeneralNames.getInstance(dpn.getName()).getNames()) {
                            if (gn.getTagNo() == GeneralName.uniformResourceIdentifier) {
                                return gn.getName().toString();
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Figure 48 : Extraction de la CRL

Téléchargement de la CRL

Une fois l'URL extraite, la CRL est téléchargée et convertie en un objet X509CRL.

- L'ancienne méthode utilisait URL, mais cette classe est dépréciée depuis Java 20.

'URL(java.lang.String)' is deprecated since version 20

Constructor	Description
URL(String spec)	Deprecated. Use <code>URI.toURL()</code> to construct an instance of URL.

Figure 49 Détails sur la méthode URL

Nous utilisons désormais `URI.toURL().openStream()` pour éviter les avertissements de dépréciation.

```
public static X509CRL telechargerCRL(X509Certificate cert) { 1 usage  new *
    try {
        String crlUrl = extraireCRLDistributionPoint(cert);
        if (crlUrl == null) {
            System.err.println("Aucune URL CRL trouvée pour le certificat : " + cert.getSubjectX500Principal());
            return null;
        }

        try (InputStream crlStream = new URI(crlUrl).toURL().openStream()) {
            CertificateFactory cf = CertificateFactory.getInstance( type: "X.509");
            return (X509CRL) cf.generateCRL(crlStream);
        }
    } catch (Exception e) {
        System.err.println("Erreur lors du téléchargement de la CRL : " + e.getMessage());
        return null;
    }
}
```

Figure 50 : méthode pour télécharger la CRL

Vérification de la Révocation avec la CRL

1. On ignore la vérification pour les certificats Root CA.
2. On télécharge la CRL associée au certificat.
3. On vérifie que la CRL est bien signée par l'émetteur du certificat.
4. On cherche dans la CRL si le certificat est révoqué.

```
public static boolean verifierRevocationAvecCRL(X509Certificate cert, List<X509Certificate> possibleIssuers) { 1 usage new *
try {
    // Vérifier si c'est un Root CA
    if (cert.getSubjectX500Principal().equals(cert.getIssuerX500Principal())) {
        System.out.println("Le certificat " + cert.getSubjectX500Principal() + " est un Root CA. Vérification CRL ignorée.");
        return false;
    }

    X509CRL crl = telechargerCRL(cert);
    if (crl == null) {
        System.err.println("Impossible de récupérer la CRL pour la vérification.");
        return false;
    }

    X509Certificate crlIssuerCert = null;
    for (X509Certificate issuer : possibleIssuers) {
        if (crl.getIssuerX500Principal().equals(issuer.getSubjectX500Principal())) {
            crlIssuerCert = issuer;
            break;
        }
    }

    if (crlIssuerCert == null) {
        System.err.println("Aucun certificat trouvé correspondant à l'émetteur de la CRL");
        return false;
    }

    // Vérification de la signature de la CRL
    try {
        crl.verify(crlIssuerCert.getPublicKey());
    } catch (Exception e) {
        System.err.println("Erreur de vérification de la signature de la CRL : " + e.getMessage());
        return false;
    }

    // Vérification de la révocation du certificat
    X509CRLEntry crlEntry = crl.getRevokedCertificate(cert.getSerialNumber());
    return crlEntry != null;
}
```

Figure 51 : Méthode pour la vérification des révocations

Nous pouvons alors éprouver notre solution en la testant :

```
== Vérification de la révocation via CRL ==
Téléchargement de la CRL depuis : http://crl.sectigo.com/SectigoQualifiedWebsiteAuthenticationCAE35.crl
* émetteur de la CRL téléchargée : CN=Sectigo Qualified Website Authentication CA E35, O=Sectigo (Europe) SL, C=ES
Signature de la CRL vérifiée avec succès.
Le certificat CN=www.tbs-certificats.com, OID.2.5.4.97=NTRFR-440 443 810 00021, 0=TBS CERTIFICATS, ST=Calvados, C=FR, OID.2.5.4.15=Private Organization, OID.1.3.6.1.4.1.311.60.2.1.3=
Le certificat CN=www.tbs-certificats.com, OID.2.5.4.97=NTRFR-440 443 810 00021, 0=TBS CERTIFICATS, ST=Calvados, C=FR, OID.2.5.4.15=Private Organization, OID.1.3.6.1.4.1.311.60.2.1.3=
Téléchargement de la CRL depuis : http://crl.usertrust.com/USERTrustECCCertificationAuthority.crl
* émetteur de la CRL téléchargée : CN=USERTrust ECC Certification Authority, O=The USERTRUST Network, L=Jersey City, ST=New Jersey, C=US
Signature de la CRL vérifiée avec succès.
Le certificat CN=Sectigo Qualified Website Authentication CA E35, O=Sectigo (Europe) SL, C=ES n'est pas révoqué.
Le certificat CN=Sectigo Qualified Website Authentication CA E35, O=Sectigo (Europe) SL, C=ES n'est pas révoqué.
```

Figure 52 : Test sur une chaîne de certificats

Cependant, ça ne fonctionne pas pour le CA Root. Ce qui est normal car ils n'ont pas de CRL.

```
Aucune extension CRL trouvée.
Aucune URL CRL trouvée pour le certificat : CN=GlobalSign, O=GlobalSign, OU=GlobalSign Root CA - R3
Impossible de récupérer la CRL pour la vérification.
```

Figure 53 : Test sur un CA root

2. Ajouter la vérification du statut de révocation en utilisant le protocole OCSP s'il est disponible pour un certificat donné

Nous avons ajouté la vérification de la révocation via **OCSP** (Online Certificate Status Protocol). Ce protocole interroge un serveur OCSP pour savoir si un certificat a été révoqué, sans nécessiter le téléchargement d'une CRL complète.

```
// Vérification via OCSP si possible
boolean estRevokedOCSP = ValidateCert.verifierRevocationOCSP(certToCheck, issuerCert);
if (estRevokedOCSP) {
    System.err.println("Le certificat " + certToCheck.getSubjectX500Principal() + " est révoqué selon OCSP !");
    return;
}

// Si OCSP n'est pas disponible, bascule sur CRL
boolean estRevokedCRL = ValidateCert.verifierRevocationAvecCRL(certToCheck, List.of(issuerCert));
if (estRevokedCRL) {
    System.err.println("Le certificat " + certToCheck.getSubjectX500Principal() + " est révoqué selon la CRL !");
    return;
}
```

Figure 54 : Vérification avec OCSP si disponible ou CRL sinon

Notre méthode principale est **verifierRevocationOCSP**, on extrait l'URL OCSP à partir du certificat (via **extraireOCSPUrl**), ensuite on créer une requête OCSP (avec **creerRequeteOCSP**), et pour finir on envoie la requête et analyse la réponse (avec **envoyerRequeteOCSP**).

```
public static boolean verifierRevocationOCSP(X509Certificate cert, X509Certificate issuerCert) { 1 usage  ↗ Sanqyh
try {
    //Récupérer l'URL OCSP depuis le certificat
    Optional<String> ocspUrlOpt = extraireOCSPUrl(cert);
    if (ocspUrlOpt.isEmpty()) {
        System.err.println("Aucune URL OCSP trouvée pour " + cert.getSubjectX500Principal());
        return false;
    }
    String ocspUrl = ocspUrlOpt.get();

    //Construire et envoyer la requête OCSP
    return Optional.ofNullable(creerRequeteOCSP(cert, issuerCert)) Optional<byte[]>
        .map( byte[] req -> envoyerRequeteOCSP(ocspUrl, req)) Optional<Boolean>
        .orElse( other: false);

} catch (Exception e) {
    System.err.println("Erreur OCSP : " + e.getMessage());
    return false;
}
}
```

Figure 55 : Méthode de vérification avec OCSP

```
== Vérification de la révocation via OCSP ==
Le certificat CN=*.lemonde.fr n'est pas révoqué.
Le certificat CN=GlobalSign Atlas R3 DV TLS CA 2024 Q4, O=GlobalSign nv-sa, C=BE n'est pas révoqué.
```

Figure 56 : Affichage du test avec vérification OCSP

3. Ajouter un mécanisme de cache pour ne pas télécharger une CRL s'il elle n'a pas été mise à jour

Deux approches possibles :

1. Cache en mémoire
 - o Le cache est perdu lorsque le programme s'arrête.
2. Cache sur disque
 - o Chaque CRL est sauvegardée dans un dossier sur le disque, afin de persister entre les exécutions du programme.
 - o Au prochain lancement, on vérifie si le fichier est toujours valide (date d'expiration). Si oui, on le charge au lieu de télécharger la CRL à nouveau.

Ici la version stockage en mémoire n'est pas intéressante pour notre cas d'utilisation car les sauvegardes ne seront pas persistantes sur le disque.

Vérifier si la CRL est déjà en mémoire sur le disque

1. Lorsqu'on appelle telechargerCRL(cert), on extrait l'URL CRL du certificat.
2. On regarde si l'URL est dans la map crlCache.
3. Si la CRL s'y trouve et que **sa date de mise à jour** (getNextUpdate()) n'est pas dépassée, on **réutilise** cette CRL.

```
//Vérifier d'abord si une CRL valide est en mémoire
if (crlCache.containsKey(crlUrl)) {
    X509CRL cachedCRL = crlCache.get(crlUrl);
    if (cachedCRL.getNextUpdate().after(new Date())) {
        System.out.println("Utilisation de la CRL en mémoire pour : " + crlUrl);
        return cachedCRL;
    }
}
```

Figure 57 : Vérification de la présence de la CRL sur disque

Charger depuis le Disque si non Présente en Mémoire

1. Si la CRL n'est pas en mémoire sur le disque (ou a expiré), on essaie de la charger depuis le disque (méthode chargerCRLDepuisDisque(crlUrl)).
2. Si le fichier existe et est encore valide, on le charge et on l'ajoute en mémoire (crlCache.put(crlUrl, crl)) puis on le renvoie.

```
//Si non trouvée en mémoire, essayer de charger depuis le disque
X509CRL crlFromDisk = chargerCRLDepuisDisque(crlUrl);
if (crlFromDisk != null) {
    crlCache.put(crlUrl, crlFromDisk);
    return crlFromDisk;
}
```

Figure 58 : Charger la CRL depuis le disque si elle est présente

Télécharger et Sauvegarder si le Cache est Vide ou Expiré

1. Si la CRL n'est ni en mémoire, ni disponible sur le disque, on la télécharge depuis l'URL.
2. On la place en cache mémoire et on la sauvegarde sur le disque.

```
//Si la CRL n'est pas en cache, la télécharger
System.out.println("Téléchargement de la CRL depuis : " + crlUrl);
try (InputStream crlStream = new URI(crlUrl).toURL().openStream()) {
    CertificateFactory cf = CertificateFactory.getInstance(type: "X.509");
    X509CRL crl = (X509CRL) cf.generateCRL(crlStream);

    // Sauvegarde en mémoire et sur disque
    crlCache.put(crlUrl, crl);
    sauvegarderCRLSurDisque(crlUrl, crl);

    System.out.println("CRL téléchargée et mise en cache : " + crlUrl);
    return crl;
}
```

Figure 59 : Télécharger la CRL si non présente sur le disque

Au premier lancement, nous voyons bien que l'on télécharge la CRL et qu'on la met en cache :

```
== Vérification de la révocation via OCSP et CRL ==
Téléchargement de la CRL depuis : http://crl.globalsign.com/ca/gsatlasr3dvtlsca2024q4.crl
CRL sauvegardé sur disque : C:\Users\louis\OneDrive\Documents\GitHub\Cryptographie-Projet---CATALA-Alexandre-VERNANCHET-Louis\ProjetcryptoGradle\cache_crl\198469553.crl
CRL téléchargée et mise en cache : http://crl.globalsign.com/ca/gsatlasr3dvtlsca2024q4.crl
Le certificat CN=*.lemonde.fr n'est pas révoqué.
Téléchargement de la CRL depuis : http://crl.globalsign.com/root-r3.crl
CRL sauvegardé sur disque : C:\Users\louis\OneDrive\Documents\GitHub\Cryptographie-Projet---CATALA-Alexandre-VERNANCHET-Louis\ProjetcryptoGradle\cache_crl\1583460194.crl
CRL téléchargée et mise en cache : http://crl.globalsign.com/root-r3.crl
Le certificat CN=GlobalSign Atlas R3 DV TLS CA 2024 Q4, O=GlobalSign nv-sa, C=BE n'est pas révoqué.
```

Figure 60 : Affichage du téléchargement des CRL

Nous faisons un deuxième lancement, nous voyons bien que cette fois ci nous utilisons le disque pour vérifier la CRL :

```
== Vérification de la révocation via OCSP et CRL ==
CRL chargée depuis le disque : C:\Users\louis\OneDrive\Documents\GitHub\Cryptographie-Projet---CATALA-Alexandre-VERNANCHET-Louis\ProjetcryptoGradle\cache_crl\198469553.crl
Le certificat CN=*.lemonde.fr n'est pas révoqué.
CRL chargée depuis le disque : C:\Users\louis\OneDrive\Documents\GitHub\Cryptographie-Projet---CATALA-Alexandre-VERNANCHET-Louis\ProjetcryptoGradle\cache_crl\1583460194.crl
Le certificat CN=GlobalSign Atlas R3 DV TLS CA 2024 Q4, O=GlobalSign nv-sa, C=BE n'est pas révoqué.
```

Figure 61 : Vérification de la persistance des CRL sur disque

4. Question Bonus

La loi applicable et la juridiction compétente pour un éventuel procès entre Le Monde (déteinteur d'un certificat) et son fournisseur de certificat (une Autorité de Certification, AC) dépendent principalement :

1. Du contrat ou des conditions générales conclues entre Le Monde et l'AC.
 - o Souvent, ces documents prévoient une loi applicable et une clause attributive de juridiction (ex. tribunaux de Paris, tribunaux californiens, etc.).
 - o Exemple : si l'AC est basé aux États-Unis et que le contrat stipule la compétence des tribunaux du Delaware, c'est cette clause qui s'applique, sauf contrariété à l'ordre public français ou européen.

2. Du lieu d'établissement des parties et de la réglementation en vigueur (droit national, droit de l'Union européenne, conventions internationales).
 - Si l'AC est situé dans l'UE et fournit un service de confiance qualifié, le règlement eIDAS (n° 910/2014) peut s'appliquer.
 - En France, les dispositions du Code civil, du Code de commerce et éventuellement du Code de la consommation (s'il s'agit d'une relation assimilable à du B2C, hypothèse peu probable pour Le Monde) peuvent intervenir.
 - En cas de service transfrontalier, on se réfère également aux conventions et règlements applicables en droit international privé.
3. Des règles du droit international privé et des règlements européens si aucune clause contractuelle claire n'existe.
 - Le règlement Bruxelles I (refondu) (n° 1215/2012) détermine la compétence judiciaire en matière civile et commerciale dans l'UE.
 - Le règlement Rome I (n° 593/2008) fixe la loi applicable aux obligations contractuelles au sein de l'UE.

IV. Conclusion

Ce projet nous a permis d'explorer en profondeur les mécanismes de validation des certificats X.509, en intégrant des techniques avancées de cryptographie et en assurant la vérification rigoureuse des signatures RSA et ECDSA. Nous avons progressivement construit une solution capable de valider une chaîne complète de certificats, en contrôlant non seulement la signature de chaque certificat mais aussi la cohérence des sujets et des émetteurs.

Tout au long du projet, plusieurs défis techniques ont été rencontrés :

- Problèmes d'importation de bibliothèques : L'intégration de Bouncy Castle via Gradle a nécessité des ajustements, notamment en raison du cache d'IntelliJ IDEA.
- Compréhension et manipulation des signatures ECDSA : L'implémentation des vérifications ECDSA a été plus complexe que prévu, nécessitant une compréhension approfondie des courbes elliptiques et de leurs paramètres.
- Gestion de la chaîne de certificats : Assurer une validation récursive et sécurisée de la chaîne a exigé un travail important sur la factorisation et l'optimisation du code.

Bien que notre implémentation soit fonctionnelle et robuste, plusieurs pistes d'amélioration sont envisageables :

- Ajout du support d'autres algorithmes de signature, comme DSA ou ECDSA, pour rendre notre solution plus complète.
- Amélioration des performances, notamment en optimisant la gestion des certificats et en réduisant le nombre d'opérations cryptographiques redondantes.
- Intégration d'une meilleure gestion des erreurs, avec des messages plus précis et des logs détaillés pour faciliter le débogage et l'analyse des échecs de validation.
- Mise en place de tests unitaires plus poussés, en couvrant davantage de cas, y compris des certificats corrompus ou mal formés.

Pour structurer notre travail et faciliter la compréhension du code, nous avons mis en place une JavaDoc complète, permettant une meilleure lisibilité et réutilisabilité du code. De plus, nous avons factorisé au maximum notre implémentation, en évitant la redondance et en améliorant la modularité des différentes fonctions.

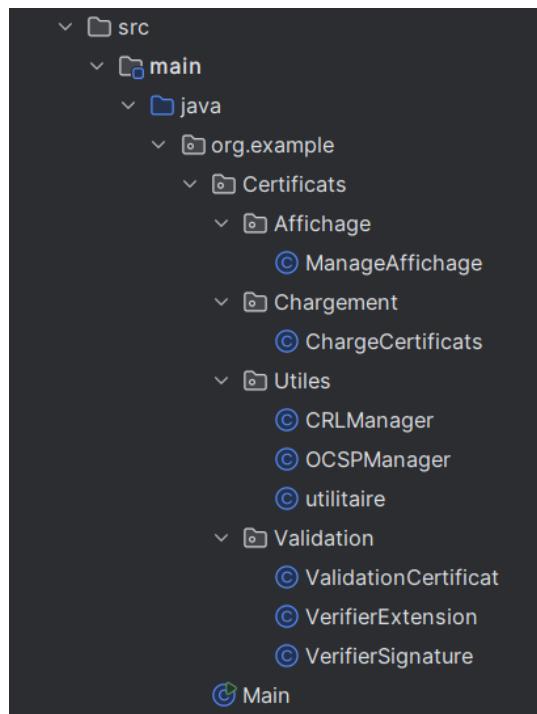


Figure 62 : Architecture du projet

Class ManageAffichage

```

java.lang.Object
  org.example.Certificats.Affichage.ManageAffichage
  
```

```

public class ManageAffichage
extends Object
  
```

Constructor Summary

Constructors	Description
ManageAffichage()	

Method Summary

All Methods	Static Methods	Concrete Methods	Description
Modifier and Type	Method		
static void	afficherAide()		
static void	afficherInfos(X509Certificate cert)		Affiche les informations détaillées d'un certificat unique

Figure 63 : Extrait de la Javadoc accessible depuis le index.html

Grâce à cette approche, notre solution offre une validation efficace et sécurisée des certificats X.509, avec une base solide pour d'éventuelles évolutions futures.

V. Bibliographie

- ANSSI. (s.d.). *Documentation des autorités de certification publiques (ex. ANSSI en France, ENISA au niveau de l'UE)*. Récupéré sur www.ssi.gouv.fr
- D. Cooper, S. S. (2008, May). *RFC 5280 : Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. Récupéré sur Network Working Group: datatracker.ietf.org/doc/html/rfc5280
- Inc., T. L. (s.d.). *The Bouncy Castle Crypto APIs for Java*. Récupéré sur bouncycastle.org
- OPEN AI. (2025). *ChatGPT: Modèle de traitement du langage naturel*. Récupéré sur <https://openai.com/chatgpt>
- Oracle. (s.d.). *Java Platform, Standard Edition (Java SE) 8+ Documentation*. Récupéré sur docs.oracle.com/en/java/
- Oracle. (s.d.). *KeyStore & Certificat Management*. Récupéré sur docs.oracle.com/javase/8/docs/technotes/tools/unix/keytool.html
- Oracle. (s.d.). *Package java.security et sous-packages (Java Security Architecture)*. Récupéré sur docs.oracle.com/javase/8/docs/technotes/guides/security/
- Parlement européen et Conseil de l'Union européenne. (2008). *Règlement Rome I (n° 593/2008)*. Récupéré sur eur-lex.europa.eu/legal-content/FR/TXT/?uri=CELEX:32008R0593
- Parlement européen et Conseil de l'Union européenne. (2012). *Règlement Bruxelles I (refondu) (n° 1215/2012)*. Récupéré sur eur-lex.europa.eu/legal-content/FR/TXT/?uri=CELEX:32012R1215
- Parlement européen et Conseil de l'Union européenne. (2014). *Règlement eIDAS (n° 910/2014) : Identification électronique et services de confiance*. Récupéré sur eur-lex.europa.eu/legal-content/FR/TXT/?uri=CELEX:32014R0910
- République française. (s.d.). *Code civil, Code de commerce, Code de la consommation (France)*. Récupéré sur www.legifrance.gouv.fr
- S. Santesson, M. M. (2013, June). *RFC 6960 : X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP*. Récupéré sur Internet Engineering Task Force (IETF): datatracker.ietf.org/doc/html/rfc6960
- Union), I. (. (2025, January 16). *Spécifications ITU-T X.509*. Récupéré sur ITU (International Telecommunication Union): www.itu.int/rec/T-REC-X.509