

Signature électronique Horodatage BouncyCastle

Cet article à pour objectif de fournir l'information nécessaire à la compréhension de la signature électronique et l'horodatage ainsi que la mise en application de ces concepts et normes en java grace à l'API de BouncyCastle.

1. rappel sur le jeton d'horodatage et serveur de temps
2. rappel sur la signature électronique
3. Créer un keystore contenant les clefs privés et public
4. Création d'un jeton d'horodatage
5. Signature électronique d'un document (PKCS#7)
6. Vérification d'une signature

1 rappel sur le jeton d'horodatage et serveur de temps

Le tiers horodateur construit une réponse contenant les données de la requête et en particulier l'empreinte, et y rajoute une marque de temps ainsi que des données additionnelles dont :

l'identité du tiers horodateur et la politique sous laquelle il a produit le jeton. Le jeton est contenu dans la réponse sous la forme d'une structure CMS (Cryptographic Message Syntax) signée. Le jeton d'horodatage permet de fournir la preuve d'existence de données à un instant dans le temps. Le but est de faire le lien entre une chaîne de caractères et une marque de temps :

- la chaîne de caractères : c'est l'empreinte numérique (20 ou 16 octets) d'une chaîne quelconque de données obtenue par une fonction de hachage (SHA-1, MD5, RIPEMD-160) ;
- La marque de temps : c'est la valeur de temps obtenue d'un serveur de temps fiable, sous la forme YYYYMMDDhhmmss. (La norme permet d'introduire des fractions de seconde si nécessaire) ;

- La référence du temps : il est basé sur l'U.T.C. qui est le temps donné par Greenwich.

Appartenant au domaine de la certification électronique, la fonction d'horodatage met en Suivre un certificat électronique d'un type particulier, appelé "jeton", en anglais "token". Ainsi, la RFC 3161 (Internet X509 Public Key Infrastructure Time Stamp Protocol (TSP)), publiée par IETF définit un jeton d'horodatage comme suit :

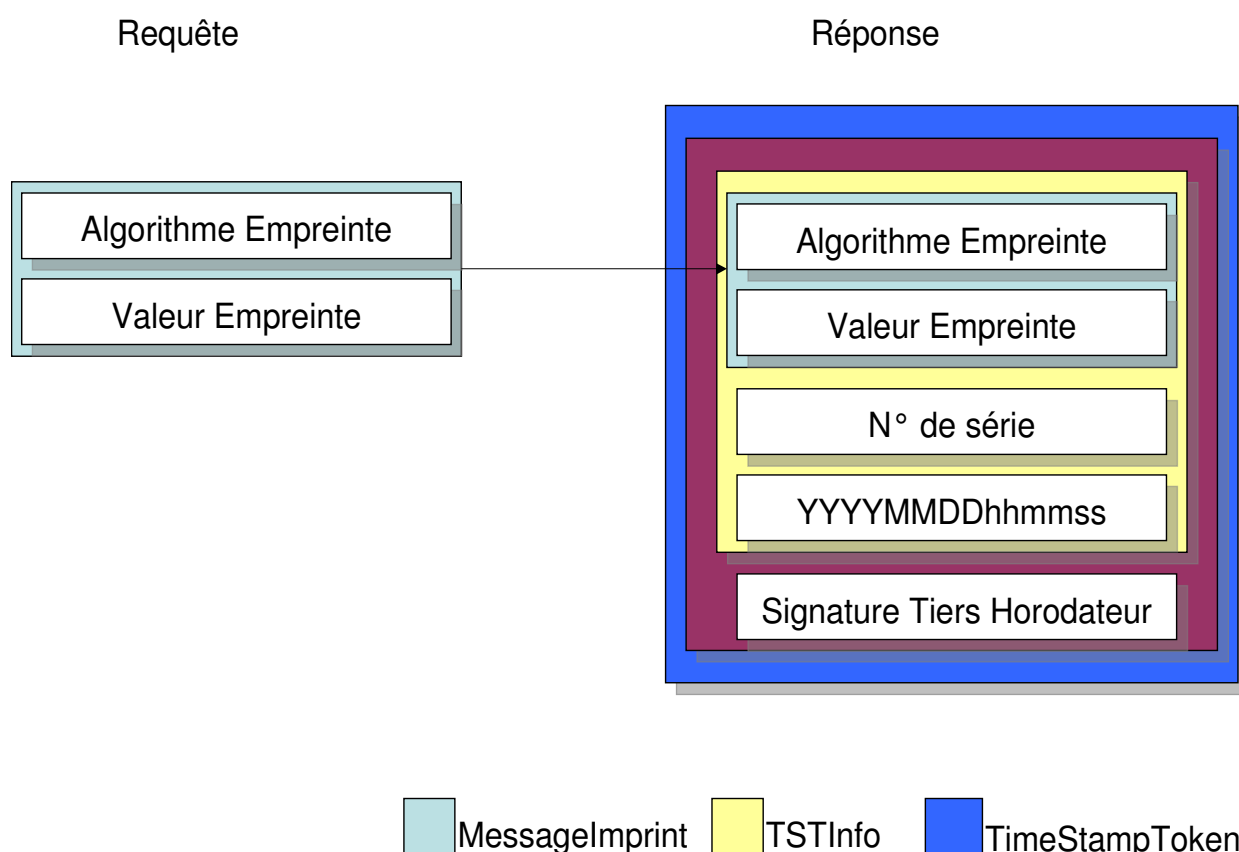
```
TimeStampToken ::= ContentInfo
-- contentType est id-signedData ([CMS])
```

Tutoriels & co

-- content est SignedData ([CMS])

Il doit encapsuler une information signée et être une valeur codée DER de TSTInfo.

```
TSTInfo ::= SEQUENCE {
    version          INTEGER { v1(1) },
    policy           TSAPolicyId,
    messageImprint   MessageImprint,
    -- DOIT avoir la même valeur que le champ similaire TimeStampReq
    serialNumber     INTEGER,
    genTime          GeneralizedTime,
    accuracy         Accuracy          OPTIONAL,
    ordering         BOOLEAN           DEFAULT FALSE,
    nonce           INTEGER           OPTIONAL,
    -- DOIT être présent si un champ similaire était présent dans TimeStampReq. Dans ce
    -- cas, il DOIT avoir la même valeur.
    tsa             [0] GeneralName    OPTIONAL,
    extensions      [1] IMPLICIT Extensions OPTIONAL
}
MessageImprint ::= SEQUENCE {
    hashAlgorithm    AlgorithmIdentifier,
    hashedMessage    OCTET STRING
}
```



2 rappel sur la signature électronique

La **signature numérique** est un mécanisme permettant d'authentifier l'auteur d'un document électronique et de garantir son intégrité, par analogie avec la signature manuscrite d'un document papier. Un mécanisme de signature numérique doit présenter les propriétés suivantes :

- Il doit permettre au lecteur d'un document d'identifier la personne ou l'organisme qui a apposé sa signature.
- Il doit garantir que le document n'a pas été altéré entre l'instant où l'auteur l'a signé et le moment où le lecteur le consulte.

Pour cela, les conditions suivantes doivent être réunies :

- **Authentique** : L'identité du signataire doit pouvoir être retrouvée de manière certaine.
- **Infalsifiable** : La signature ne peut pas être falsifiée. Quelqu'un d'autre ne peut se faire passer pour un autre.
- **Non réutilisable**: La signature n'est pas réutilisable. Elle fait partie du document signé et ne peut être déplacée sur un autre document.
- **Inaltérable** : Un document signé est inaltérable. Une fois qu'il est signé, on ne peut plus le modifier.

Tutoriels & co

venez de créer puis dans le cadre target sélectionner SHA1withRSA pour l algo .

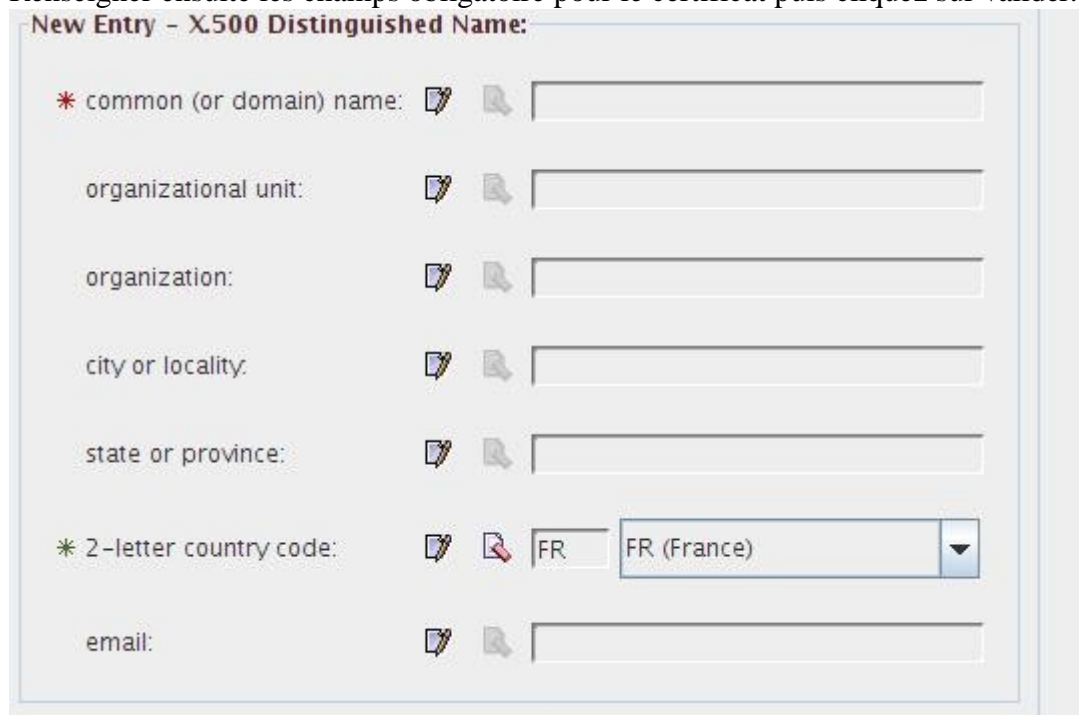


Target:

New Entry - Private Key:

- * key size (bits): 2048
- * certificate version: 3
- * cert. signature algo.: SHA1withRSA
- * validity (days): 1095

Renseigner ensuite les champs obligatoire pour le certificat puis cliquez sur valider.



New Entry - X.500 Distinguished Name:

- * common (or domain) name:
- organizational unit:
- organization:
- city or locality:
- state or province:
- * 2-letter country code: FR (France)
- email:

a partir de la vous êtes l heureux propriétaire d un keystore avec une pair de clef privé/public.
nous pouvons donc passer a la création du code.

En premier lieu les quelques constante nécessaire à adapter selon vos besoins.

```
private final static String FILE_TO_SIGNED =  
"/home/ubuntu/Documents/articles  
karlverger.com/tutorielBouncyCastle/lib/fichierasigner.txt";  
private final static String KEYSTORE_FILENAME =  
"/home/ubuntu/Documents/articles  
karlverger.com/tutorielBouncyCastle/lib/keystore.p12";  
private final static String KEYSTORE_PASSWORD = "poipoi"; //password  
utiliser lors de la création du keystore  
private final static String CERTIFICAT_NAME = "keypairTest"; //nom de
```

Tutoriels & co

l'alias que vous avez fournis lors de la création des clefs

```
private final static String CERTIFICAT_PASSWORD = "poipoi";
private final static String TSA_POLICY_ID = "0.0";
```

et le code de chargement du keystore

```
/**
 * Charge le keystore
 * @param fileName
 * @param pwd
 * @return
 * @throws java.lang.Exception
 */
public static KeyStore getKeystore(String fileName, String pwd) throws
Exception {
    KeyStore keyStore = KeyStore.getInstance("PKCS12", "BC");
    keyStore.load(new ByteArrayInputStream(Util.getBytesFromFile(new
File(fileName))), pwd.toCharArray());
    return keyStore;
}
```

4 Création d un jeton d horodatage

```
private final static String KEYSTORE_PASSWORD = "poipoi";
private final static String CERTIFICAT_NAME = "keypairTest";
private final static String CERTIFICAT_PASSWORD = "poipoi";
private final static String TSA_POLICY_ID = "0.0";

/**
 * Création d'un jeton d'horodatage via la date du systeme
 * pour nos test c'est tout a fait valable nous avons bien un jeton
 * conforme à la norme si ce n'est que la date n'est pas probante.
 * et accessoirement nous utilison aussi un certificat autosigné
 * pour obtenir un jeton ayant valeur légale vous devez obtenir le temps
 * par un serveur NTP et utiliser une pair de clef fournie aupres d'une PKI
 * @param empreinte
 * @param idJeton
 * @return
 * @throws java.lang.Exception
 */
public static byte[] getTimestamp(byte[] empreinte, String idJeton) throws
Exception {
    KeyStore ks = getKeystore(KEYSTORE_FILENAME, KEYSTORE_PASSWORD);
    try {
        // recup keystore, certifs et clefs
        X509Certificate cert = (X509Certificate)
ks.getCertificate(CERTIFICAT_NAME);
        PrivateKey pk = (PrivateKey) ks.getKey(CERTIFICAT_NAME,
CERTIFICAT_PASSWORD.toCharArray());
        ArrayList certList = new ArrayList();
        certList.add(cert);
        CertStore certs = CertStore.getInstance("Collection", new
```

Tutoriels & co

```
CollectionCertStoreParameters(certList), "BC");

    String algorithme;
    algorithme = TSPAlgorithms.SHA1;
    TimeStampTokenGenerator tokenGen = new TimeStampTokenGenerator(pk,
cert, algorithme, TSA_POLICY_ID);
    tokenGen.setCertificatesAndCRLs(certs);

    TimeStampRequestGenerator gen = new TimeStampRequestGenerator();
    gen.setCertReq(true);

    TimeStampRequest req = gen.generate(algorithme, empreinte, new
BigInteger(80, SecureRandom.getInstance("SHA1PRNG")));
    TimeStampToken token = tokenGen.generate(req, new
BigInteger(idJeton), new Date(), "BC");
    return token.getEncoded();
} catch (Exception e) {
    throw new Exception("Internal error while generating timestamp", e);
}
}
```

5 Signature électronique d'un document (PKCS#7)

```
/**
 * Génère une signature PKCS#7 embarquant l'empreinte
 * du document à signer
 * et la renvoie sous forme de tableau de byte
 * @param empreinteDocASigner
 * @return
 * @throws java.lang.Exception
 */
public static byte[] getSignature(byte[] empreinteDocASigner) throws
Exception {
    System.out.println("Réalisation de la signature ....");
    KeyStore ks = getKeystore(KEYSTORE_FILENAME, KEYSTORE_PASSWORD);
    try {
        /**Récupération du certificat et de la clef**/
        X509Certificate cert = (X509Certificate)
ks.getCertificate(CERTIFICAT_NAME);
        PrivateKey pk = (PrivateKey) ks.getKey(CERTIFICAT_NAME,
CERTIFICAT_PASSWORD.toCharArray());
        ArrayList certList = new ArrayList();
        certList.add(cert);
        CertStore certs = CertStore.getInstance("Collection", new
CollectionCertStoreParameters(certList), "BC");

        /**création des information concernant le signataire**/
        CMSSignedDataGenerator signGen = new CMSSignedDataGenerator();
        signGen.addSigner(pk, cert, CMSSignedDataGenerator.DIGEST_SHA1);
        signGen.addCertificatesAndCRLs(certs);

        /**génération de la signature**/
        CMSProcessable content = new
```

Tutoriels & co

```
CMSPprocessableByteArray(empreinteDocASigner);
    CMSSignedData signedData = signGen.generate(content, true, "BC");

    byte[] signeddata = signedData.getEncoded();
    System.out.println("signature à été réalisée....");
    return signeddata;
} catch (Exception e) {
    throw new Exception("Erreur lors du processus de signature", e);
}
}
```

6 Vérification d'une signature

```
/**
 * vérifie la signature
 * @param signaturePKCS7
 */
public static void verifieSignature(byte[] signaturePKCS7){

    try {

        System.out.println("Vérification de la signature ....");

        /**lecture de l'enveloppe PKCS7**/

        CMSSignedData signature = new CMSSignedData(signaturePKCS7);

        /**récupération des informations sur le signataire**/

        SignerInformation signer = (SignerInformation)
signature.getSignerInfos().getSigners().iterator().next();

        /**récupération du certificat du signataire**/

        CertStore cs = signature.getCertificatesAndCRLs("Collection", "BC");

        Iterator iter = cs.getCertificates(signer.getSID()).iterator();

        X509Certificate certificate = (X509Certificate) iter.next();

        /**récupération de l'empreinte qui à été signé**/

        CMSPprocessable sc = signature.getSignedContent();

        byte[] data = (byte[]) sc.getContent();
```

Tutoriels & co

```
        System.out.println("Empreinte contenu dans la signature :  
"+Util.asHex(data));  
  
        // Verifie la signature  
  
        System.out.println("Validité de la signature :  
"+signer.verify(certificate, "BC"));  
  
        } catch (CertStoreException ex) {  
  
            Logger.getLogger(SignHorodate.class.getName()).log(Level.SEVERE,  
null, ex);  
  
        } catch (NoSuchAlgorithmException ex) {  
  
            Logger.getLogger(SignHorodate.class.getName()).log(Level.SEVERE,  
null, ex);  
  
        } catch (NoSuchProviderException ex) {  
  
            Logger.getLogger(SignHorodate.class.getName()).log(Level.SEVERE,  
null, ex);  
  
        } catch (CertificateExpiredException ex) {  
  
            Logger.getLogger(SignHorodate.class.getName()).log(Level.SEVERE,  
null, ex);  
  
        } catch (CertificateNotYetValidException ex) {  
  
            Logger.getLogger(SignHorodate.class.getName()).log(Level.SEVERE,  
null, ex);  
  
        } catch (CMSException ex) {  
  
            Logger.getLogger(SignHorodate.class.getName()).log(Level.SEVERE,  
null, ex);  
  
        }  
  
    }
```

Tutoriels & co

www.karlverger.com