TP : Signature électronique en JAVA (4 heures)

Anthony JULOU, Cyril THIRION

Le but de ce TP est d'implémenter la signature électronique à l'aide des API cryptographiques en JAVA dont JCA (*Java Cryptography Architecture*)

Exercice 1 : Stocker un mot de passe avec un algorithme de hachage

Exercice 2 : Générer et valider la signature (scellement) d'un message à l'aider d'un bi clé RSA et d'un algorithme de hachage

Exercice 3 : Utiliser une API spécialisée avec un certificat X509 pour réaliser une signature électronique d'un document PDF

Préquis :

L'environnement de développement utilisé nécessite

- **Eclipse** *(Dernière version de préférence)* ou un autre IDE JAVA de votre choix (NetBeans, IntelliJ…)
  - http://www.eclipse.org
- **Java** (JRE *Java Runtime Environnement* ou SE *Standard Edition*) version supérieure à 1.8
  - https://www.oracle.com/technetwork/java/index.html
- **Acrobat Reader** *(Dernière version DC)* ou tout autre lecteur PDF reconnaissant la signature électronique
  - https://get.adobe.com/fr/reader

Fournitures pour l'exercice 3 :

- **Un certificat numérique X509** fourni dans un fichier PKCS12 d'extension « .p12 » ou « .pfx » contenant le biclé RSA, son certificat associé et la chaine de confiance ou bien à télécharger gratuitement sur un site.
- Les JAR de l'API de signature PDF fournis dans le fichier **API.zip** à intégrer dans le classpath Java.

Exercice 1 : Stocker un mot de passe avec un algorithme de hachage

✓ Le but de cet exercice est de stocker de manière sécurisée un mot de passe afin qu'il ne soit ni **visible en clair et ni trouvable** facilement sur Internet ou par un tier (Attaque par force brute ou rejeu).

A) Stockage du mot de passe en clair

Dans un premier temps, nous allons stocker (en mémoire dans un attribut) le mot de passe en clair.

1. Créer une classe `MyPassword` avec pour attribut `password` de type `private String` qui sera le mot de passe à stocker. Créer le constructeur avec son paramètre associé. Surchargez la méthode `toString()` qui doit retourner la chaine suivante : `"Mot de passe stocké : " + this.password ;`
2. Créer l'accesseur en lecture `getPassword()`
3. Dans la méthode `main()`, utiliser ce code pour faire saisir une chaine par l'utilisateur :

```
System.out.println("Veuillez définir un mot de passe : ");
Scanner scanner = new Scanner( System.in );
String passString = scanner.nextLine();
```

4. Instancier un objet `myPassword` de la classe `MyPassword` avec en paramètre le mot de passe *« toto »* saisi au clavier ; faire afficher sur la console le mot de passe par l'instruction `System.out.println(myPassword)`. Exécuter le programme (« Bouton vert 'run' »).

B) Stockage du mot de passe en sha256

> ✓ Un mot de passe ne devant pas être stocké en clair, nous allons le chiffrer avec l'empreinte d'un algorithme de hachage.

1. A partir de la documentation javadoc en ligne de la classe `java.security.MessageDigest`, ajouter une méthode de classe (static) « `public static String hacheSha256(String pMessage)` » qui retourne le haché en SHA256 de `pMessage` ; ceci en utilisant successivement les méthodes documentées « `getInstance()` » et « `digest()` ».

2. Maintenant, modifier le corps du constructeur pour initialiser l'attribut `password` en appelant la méthode `hacheSha256()`.

Indication : L'affichage n'étant pas très lisible en raison du nombre important de caractères non affichages dans le jeu de caractères Unicode (16 bits), utiliser ce bout de code à la fin du corps de la méthode `hacheSha256()` pour transformer le tableau de bytes du « digest() » en hexadécimal plus visibile.

> ✓ Ce format est généralement la manière la plus simple de transporter un identifiant codé sans qu'il ne subisse de translation de jeux de caractères dans son transport (http notamment).

```
StringBuffer sb = new StringBuffer();
byte[] lMessageUTF8;
try {
      // On convertit de le message de UNICODE vers UTF-8
      lMessageUTF8 = pMessage.getBytes("UTF-8");
} catch (UnsupportedEncodingException e) {
      return "Erreur encodage";
}

for(byte b : lMessageDigest.digest( lMessageUTF8 ) ) {
  sb.append(String.format("%02x", b));
}
```

3. Exécuter le programme en saisissant « `toto` » comme mot de passe.

Copier l'empreinte du mot de passe obtenu et coller dans un moteur de recherche sur Internet. Avez-vous trouvé le mot de passe déchiffré ? Si oui, pourquoi ?

C) Stockage du mot de passe en sha256 + sel

> ✓ Un mot de passe est retrouvable facilement grâce un dictionnaire de mots de passe hachés. Il faut donc ajouter du sel et ainsi augmenter sa longueur et complexité. Ajouter du sel consiste, par exemple, à concaténer une chaine définie au mot de passe avant de le hacher.

1. Créer un attribut `private static final String SEL = '#G4D!éR$*+JD%Z'`
2. Créer une méthode `public static String saler (String pPass)` qui retourne la concaténation du `pPass` avec le SEL.
3. Modifier le constructeur pour saler avant de hacher le mot de passe.
4. Lancer le programme avec comme mot de passe « toto ». L'empreinte est-elle toujours trouvable sur Internet ?

> ✓ Il est recommandé d'utiliser des mots de passe de 12 caractères minimum et de types différents (majuscules, minuscules, chiffres, caractères spéciaux) et de les changer régulièrement (En raison des vols de données)

D) Contrôle d'accès par mot de passe

> ✓ Le mot de passe n'est donc pas stocké en clair. L'application ne connaît donc pas le mot de passe de l'utilisateur. Pour l'authentifier, elle va donc devoir lui demander la confirmation de son mot de passe et refaire le même calcul de hash afin de comparer ce résultat obtenu avec le hash mémorisé. Si les 2 hachés sont identiques, l'authentification est acceptée.

1. Surcharger la méthode `public boolean equals(Object pPass) { return this.password.equals( ((MyPassword) pPass).getPassword() ); }` qui retourne vrai si les 2 objets - et donc - les 2 mots de passe sont identiques.

2. Créer une méthode `public boolean controleAcces(String pPass)` qui
   a. Instancie un nouvel objet `myPasswordControleAcces` de type `MyPassword`
   b. Retourne le résultat du test l'égalité des 2 objets donc l'égalité des 2 mots de passe.
   c. Dans la méthode `main()`, à la suite du code existant, tester le contrôle d'accès comme cela :
      i. Demander une confirmation de saisie du mot de passe à l'utilisateur (en utilisant le scanner précédent) « *Veuillez saisir votre mot de passe pour vous connecter* : »
      ii. Appeler la méthode `controleAccess()` en affichant « *Echec* » ou « *Succès* » suivant le résultat du contrôle retourné.

Exercice 2 : Création manuelle d'une signature électronique

> ✓ Nous allons maintenant mettre en œuvre une signature électronique d'un message texte et sa vérification à l'aide des algorithmes cryptographiques et de JCA.
>
> ✓ Rappel : une signature électronique consiste à hacher le message, puis à chiffrer l'empreinte obtenue au moyen de la clé privée d'un algorithme asymétrique. Le résultat est appelé le condensat signé. La vérification consiste à déchiffrer avec la clé publique et de comparer le hash obtenu avec le hash du message brut. Si les 2 sont identiques la signature est valable.

A) Création de la classe et du haché

Créer une classe `MyFirstSignature` avec un attribut privé `myMessage` de type `String` qui sera le message à faire signer électroniquement. Créer le constructeur à un paramètre associé.

1. Dans la méthode `main()`, instancier un objet `MyFirstSignature` avec la phrase *« Je signe un message électroniquement »*;
2. Mémoriser dans une variable locale le résultat de l'appel à la méthode de classe `hacheSha256()` de l'exercice *(Attention : retirer le sel si vous l'aviez ajouté dans cette méthode)*.
3. Afficher les 2 lignes suivantes sur la console :

```
L'haché sha256 de [Je signe un message électroniquement] est
399ace8c2b57ddcd9a3708eba6c2f4665215427172373255 44a80a1ba13bb6db
```

B) Génération d'un bi-clé RSA pour réaliser le chiffrement

1) Génération du bi clé RSA

A l'aide de la documentation de la classe `java.security.KeyPairGenerator` réaliser une méthode d'instance « `private void genereBiCleRsa()` » qui génère un bi-clé RSA de 2048 bits et mémorise dans 2 nouveaux attributs d'instance la clé privée et la clé publique ainsi obtenues.

```
/** Clé privée RSA */
private PrivateKey privateKey;
/** Clé publique RSA */
```

```
private PublicKey publicKey;
```

Cette méthode sera appelée dans le constructeur de la classe « `MyFirstSignature` » dans lequel on affichera les clés obtenues sur la console.

2) Chiffrement avec la clé privée

A partir de la documentation de la classe `javax.crypto.Cipher`, réaliser une méthode d'instance `private byte[] chiffrer( String pMessage )` qui chiffre avec la clé privée le message passé en paramètre et retourne le condensat obtenu sous la forme d'un tableau de `bytes`.

Indication : Récupérer une instance de `Cipher` de type « RSA », initialiser en mode chiffrement avec la clé privée `(init)`, puis finaliser l'opération `(doFinal)` pour chiffrer et retourner le résultat.

3) Signature du message

- ✓ La signature électronique étant le résultat des 2 opérations cryptographiques réalisées précédemment :
- ✓ Le haché d'un message à signer
- ✓ Le chiffrement asymétrique du haché avec la clé privée du signataire.

Réaliser une méthode d'instance « `public byte[] signe()` » qui appelle les 2 précédentes méthodes créées.

Faire afficher le résultat de la signature produite dans la console *(transformer le tableau de byte en chaine)***:**

Indication : On construit une chaine en appelant le constructeur new `String( byte[] pTableauBytes)` du tableau de bytes déchiffrés

`La signature du message: [Je signe un message électroniquement] est :`

_1ÍJuÈW_7É_-¸¶,Ê]¸[_UC¡Ç_ºíQÏªUË_Öü"o×j_êß½Öá©ì_É„ô__ë¶ó:Z*Þ

Vérification de la signature

- ✓ Pour vérifier la signature d'un message, on procède ainsi :
- ✓ Réaliser un haché du message original, ce qui donne H
- ✓ Déchiffrer le message « signé » (le condensat chiffré) avec la clé publique du signataire.
- ✓ Si le résultat obtenu est identique à H alors la signature est valide.

Réaliser une méthode d'instance qui déchiffre un message avec la clé publique du signataire (*jumelle de « chiffrer()»)* , affiche la chaine obtenue avant de la retourner.

```
private String dechiffrer( byte pCondensat[] )
```

Le résultat affiché dans la console est :

```
Le déchiffrement est
399ace8c2b57ddcd9a3708eba6c2f4665215427172373255 44a80a1ba13bb6db
```

Créer la méthode `public boolean verifierSignature( byte pCondensat[] )` qui réalise le haché du message et le compare avec le résultat du déchiffrement obtenu à l'aide de la méthode *equals() de `String`*.
Appeler la méthode `verifierSignature()` depuis la méthode main() .

Le résultat dans la console est : `La signature est valide`

Exercice 3 : Signature d'un PDF avec un certificat X509 en Java

✓ Le but de cet exercice est de générer un PDF puis de le signer numériquement avec une signature visible réalisée par un certificat X509 au format logiciel

Avant de programmer … : Démonstration d'une signature avec Adobe Reader

A partir de la **version 11.0.9 du logiciel gratuit Adobe Reader**, il est possible de signer numériquement un document PDF si l'on dispose d'un certificat de signature électronique. Nous allons voir comment.

1. Importer le certificat fourni *« fichier .p12 »* en double-cliquant sur le fichier afin qu'Adobe le trouve dans le magasin de certificat (sous Windows) ou dans le trousseau d'accès (Macintosh)
2. Ouvrir un fichier PDF (par exemple le sujet de ce TP)
   a. Cliquez sur le bouton *« Remplir et Signer »* ;
   b. Puis sur *« Apposer une signature »* ;
   c. Cocher *« Utiliser un certificat »*, puis cliquez sur *« Suivant »* ;
   d. Il est demandé de tracer un rectangle ou apparaitra la signature. Tracez-le ;
   e. Sélectionnez le certificat pour signer ;
   f. Enregistrer le nouveau document signé qui sera une copie du fichier original non signé.
3. Ouvrez le fichier signé. Que se passe-t-il ?

Pré-requis :
Les librairies supplémentaires suivantes seront utilisées et fournies :

- iText, http://itextpdf.com/ permettant de générer des PDF
- Bouncy Caslte http://bouncycastle.org/ librairie cryptographique pour apposer les signatures sur les PDF

Les fichiers Jar de ces API sont fournis dans un zip par mail (API.zip) Les extraire et l'archive et les copier dans un répertoire « `lib` » du projet Eclipse, puis modifier le « build path » du projet pour intégrer ces libraires au classpath.

Copier le fichier certificat « .pfx » dans le répertoire racine du projet.

1. Génération d'un fichier PDF avec un texte simple

La javadoc de iText est disponible ici http://api.itextpdf.com/itext/

Ecrire une classe « `MySignaturePdf` » et une méthode d'instance « `public void generatePdf()` » qui génère un fichier PDF « `text.pdf` » avec un paragraphe « *`Je vais signer un fichier PDF`* ». Pour ce faire, examiner la javavoc de la classe « `com.itextpdf.text.Document` »

Appeler la méthode depuis le `main()`

Ouvrir le fichier « text.pdf » généré. (*F5 pour rafraichir l'affichage des fichiers de l'arborescence projet*)

2. Manipuler un keystore JAVA pour accéder au certificat

Lien de la Javadoc : `java.security.KeyStore`

Dans le constructeur, charger le `keystore` à partir du nom de fichier de certificat p12 transmis en paramètre (String) du constructeur ;

```
public MySignaturePdf( String pP12File, String pMotDePasse )
{
	// Le manageur crypto BouncyCastle est ajouté comme fournisseur de sécurité
	Security.addProvider(new BouncyCastleProvider());
	try {
		keystore = KeyStore.getInstance("pkcs12", "BC");
		// Chargement du fichier avec le mot de passe en 2ème paramètre
		keystore.load(new FileInputStream(pP12File), pMotDePasse.toCharArray());
	…
```

Chaque élément dans le keystore est accessible par un alias. Faire une boucle sur tous les alias (*aliases()*) et tester chaque entrée (entry) par `isKeyEntry( lAlias )`, ainsi si le test est ok, il s'agit du certificat personnel associé à la clé privée, l'afficher en faisant `getCertificate()`. A cette étape, mémoriser dans un attribut la clé privée et dans un tableau de `Certificate` la chaine de certification `(getCertificateChain())`, ils vont être nécessaires pour signer le PDF.

- ✓ Lorsque l'on signe un PDD avec un certificat, il faut y ajouter aussi la chaine de certification complète du certificat (Autorité(s) intermédiaire(s) et autorité racine)

3. Signature du PDF avec le certificat logiciel

Créer la méthode suivante qui ajoute une signature électronique visible sur le PDF en signant avec la clé privée. Appeler la méthode depuis le `main()`

Les paramètres d'entrée sont les noms des fichiers PDF source et destination (Le PDF signé)

```
public void apposerSignature( String pSourceFile, String pDestFile )
{
   try
   {
       // Lecteur du fichier source depuis le système de fichiers
       PdfReader reader = new PdfReader(pSourceFile);
       FileOutputStream os = new FileOutputStream(pDestFile);

       // Initialisation du tampon de signature
       PdfStamper stamper = PdfStamper.createSignature(reader, os, '\0');
       // Apparence de la signature (on aurait pu ajouter une image)
       PdfSignatureAppearance appearance = stamper .getSignatureAppearance();
       // Raison de la signature du document
       appearance.setReason("Je signe ce document.");
       // Lieu de la signature
       appearance.setLocation("Mon école ma ville");
       // Positionnement dans un rectangle de la signature visible demandée
       appearance.setVisibleSignature(new Rectangle(52, 692, 234, 760), 1,    "first");
       // Initialisation de l'algorithme de hash sha256, de la clé privée du signataire
       // et du fournisseur de sécurité, ici Bouncy Castle
       ExternalSignature es = new PrivateKeySignature(privateKey, "SHA-256", "BC");
       ExternalDigest digest = new BouncyCastleDigest();
       // Application de la signature au PDF cible avec la chaine de certification au format CADES
       MakeSignature.signDetached(appearance, digest, es, certificateChain, null, null, null, 0,
CryptoStandard.CMS);
   } catch (Exception e )    {
       e.printStackTrace();
   }
}
```