

Java Avancé - Cours 5

Plan

1	Signature de fichier	1
1.1	Notions de base	1
1.2	Génération d'une clé	2
1.3	Sauvegarde des clés	2
1.4	Signature du fichier	2
2	Bac à sable	2
2.1	Motivation	2
2.2	Le fichier <code>java.policy</code>	3
2.3	Des informations sur son bac à sable	3
2.4	Fixer la taille du bac à sable	3
3	Fichier jar signé	4
3.1	Keystore	4
3.2	Génération d'une clé dans le keystore	4
3.3	Signature du fichier jar	4
3.4	Création du certificat public	4
3.5	Politique de sécurité	4
3.6	Charger les bons certificats	4
4	Javadoc	5
4.1	Documentation d'une méthode	5
4.2	Documentation d'une classe	5
4.3	Exécuter javadoc	6
4.4	Création d'une cible ant	6
4.5	Faire le test	6
5	Exercices	6
5.1	Exercice 5.1	6
5.2	Exercice 5.2	7

1 Signature de fichier

1.1 Notions de base

Voici les notions cryptographiques de base utilisées dans ce cours :

- **clé** : une clé est constituée d'un couple clé publique, clé privée. L'interface pour manipuler les clés est la classe `java.security.KeyPair`.
- **clé privée** : la clé privée est une donnée qui doit rester confidentielle. Dans le cas de RSA il s'agit par exemple d'un couple d'entiers a et b . L'interface pour manipuler une clé privée est `java.security.PrivateKey`.
- **clé publique** : la clé publique est une donnée correspondante à une clé privée et qui peut être diffusée sans problèmes, par exemple sur un site web. Il est *a priori* extrêmement difficile de retrouver une clé privée à partir d'une clé publique. Dans le cas de RSA, pour une clé privée constituée de deux entiers a et b , la clé publique va être le produit ab . L'interface pour manipuler une clé publique est `java.security.PublicKey`.
- **signature** : une signature est une donnée associée à un fichier par exemple et qui ne peut être générée à partir du fichier, qu'en connaissant la clé privée. En revanche d'avoir la clé publique suffit à vérifier que la signature est bonne. L'implémentation se trouve dans la classe `java.security.Signature`.

1.2 Génération d'une clé

On va se servir de la classe `KeyPairGenerator` pour créer une clé RSA :

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
keyGen.initialize(1024, new SecureRandom());
KeyPair pair = keyGen.generateKeyPair();
```

Les clés publiques et privées sont alors :

```
PublicKey pub = pair.getPublic();
PrivateKey priv = pair.getPrivate();
```

1.3 Sauvegarde des clés

Pour sauver une clé (ou tout autre objet java sérialisable) on va utiliser un `ObjectOutputStream` dans le package `java.io` :

```
PublicKey pub = ... ;
FileOutputStream fout = new FileOutputStream("public.key");
ObjectOutputStream oos = new ObjectOutputStream(fout);
oos.writeObject(pub);
oos.close();
```

Pour la manipulation inverse où l'on créé l'objet `pub` à partir du fichier `public.key` il suffit d'utiliser un `ObjectInputStream` dans le package `java.io` :

```
FileInputStream fis = new FileInputStream("public.key");
ObjectInputStream ois = new ObjectInputStream(fis);
PublicKey pub = (PublicKey) ois.readObject();
ois.close();
```

1.4 Signature du fichier

Maintenant que nous avons les clés il faut créer un objet signature basé sur cette clé :

```
Signature rsa = Signature.getInstance("MD5withRSA");
rsa.initSign(priv);
```

L'initialisation à l'aide de la méthode `initSign` est réalisée car on va signer le fichier. Si cela avait été pour vérifier la signature on aurait appelé `initVerify(pub)` où `pub` est la clé publique correspondant à la clé privée générée.

Il faut alors faire "passer" tout le fichier à signer dans l'instance `rsa` :

```
FileInputStream fis = new FileInputStream(new File("nomdufichier"));
while (fis.available() != 0) {
    rsa.update(fis.read());
}
fis.close();
```

Le résultat de la signature est alors obtenu par la commande :

```
byte [] sig = rsa.sign();
```

2 Bac à sable

2.1 Motivation

Java est pratique : on peut facilement télécharger du bytecode. Cependant dès lors que l'on télécharge du code que l'on ne connais pas, il est pratique de pouvoir restreindre les droits de ce dernier.

2.2 Le fichier `java.policy`

Considérons le code suivant d'une classe `OpenFile` qui affiche le contenu du fichier `build.xml` :

```
FileInputStream f = new FileInputStream ("build.xml");
int i=0;
byte [] b = new byte[1024];
while ((i=f.read(b))!=-1) {
    System.out.println(new String(b));
}
```

Une fois compilé, ce code s'exécute sans problème avec par exemple la commande : `java OpenFile`.

Rajoutons maintenant un fichier `java.policy` vide et lançons la ligne de commande :

```
java -Djava.security.manager -Djava.security.policy=java.policy OpenFile
```

On obtient alors l'erreur suivante :

```
java.security.AccessControlException: access denied
    (java.io.FilePermission build.xml read)
```

Explication : le fichier `java.policy` vide ne donne aucun droit au code qui s'exécute. On dit que le code est contraint de jouer dans son bac à sable (ou *sand box* en anglais).

Le code réalisant ce test ainsi que le fichier ant correspondant sont téléchargeables à l'url :

http://www.derepas.com/java/cours5_exemple_2.jar

2.3 Des informations sur son bac à sable

Le code qui s'exécute peut obtenir des informations sur le `SecurityManager` (*i.e.*, le bac à sable) dans lequel il s'exécute :

```
if (System.getSecurityManager()!=null) {
    // alors il y a un security manager
    java.security.AccessController.checkPermission
        (new java.io.FilePermission("build.xml", "write"));
}
```

Le code suivant génère une exception de type `java.security.AccessControlException` si le code n'est pas autorisé à écrire dans le fichier `build.xml`.

2.4 Fixer la taille du bac à sable

Si par exemple dans l'exemple précédent on veut donner seulement le droit de lire le fichier `build.xml` voici le contenu du fichier `java.policy` :

```
grant {
    permission java.io.FilePermission "build.xml", "read";
};
```

Si on a vraiment confiance dans des classes situées dans un répertoire donné on peut leur donner tous les privilèges :

```
grant codeBase "file:/chemin/vers/mes/classes/*" {
    permission java.security.AllPermission;
};
```

Remarque : sous windows les chemins ne sont pas spécifiés sous la forme `file:/mon/chemin` mais `file:/C:/mon/chemin` pour `c:\mon\chemin`.

Pour plus de détails sur le fichier `java.policy` se reporter à l'url :

<http://java.sun.com/j2se/1.5.0/docs/guide/security/PolicyFiles.html>

3 Fichier jar signé

3.1 Keystore

Plutôt que de stocker les clés dans des fichiers comme nous avons fait précédemment, java propose une interface manipulable en ligne de commande ou par un programme.

Cet endroit ou nous allons stocker les clés se nomme le keystore.

3.2 Génération d'une clé dans le keystore

Il faut pour cela utiliser l'application `keytool` qui vient avec le JDK. L'option à rajouter sur la ligne de commande est `-genkey`, on peut en outre préciser le nom de clé générée (ici `mykey`), le type de clé (ici `RSA`), la taille (ici `1024`) et le fichier de stockage (ici `keystore`):

```
keytool -genkey -alias mykey -keyalg RSA -keysize 1024 -keystore keystore
```

Pour voir les clés qui sont stockées il suffit de taper :

```
keytool -list -keystore keystore
```

3.3 Signature du fichier jar

Si dans la liste on a une clé nommée `mykey` il suffit alors pour signer un fichier jar de taper :

```
jarsigner -keystore keystore monfichier.jar mykey
```

Les fichiers `META-INF/MYKEY.SF` et `META-INF/MYKEY.DSA` ont alors été rajoutés dans l'archive jar. Le fichier `MANIFEST.MF` a également été modifié, et contient en plus :

```
Name: ex1/ToDownload.class  
SHA1-Digest: Q4wliYWpbRKBTzUICVWjuo0+6fc=
```

c'est à dire la liste des classes ainsi que leur signature par la clé `mykey`.

3.4 Création du certificat public

Le certificat public peut être créé à l'aide de la commande :

```
keytool -export -keystore keystore -alias mykey -file derepas.cer
```

3.5 Politique de sécurité

Si l'on dispose d'une source de confiance nommée par exemple `mykey`, on a envie d'écrire un fichier `java.policy` de la forme :

```
grant signedBy "mykey" {  
    permission java.security.AllPermission;  
};
```

dont la sémantique est la suivante : fait entièrement confiance au code situé dans un fichier jar qui a été signé par la clé privée correspondant au certificat de la clé publique dans le registre de clé pour l'entrée nommée `mykey`.

3.6 Charger les bons certificats

Il faut charger le bon certificat. Pour cela récupérer le certificat public, puis l'importer dans le `KeyStore` :

```
keytool -import -alias mykey -file derepas.cer
```

Pour plus d'informations :

<http://java.sun.com/docs/books/tutorial/security1.2/>

4 Javadoc

Javadoc est un système de documentation du code java à partir des commentaires. Vous l'utilisez quand vous lisez les pages de documentation sur l'API du JDK :

<http://java.sun.com/j2se/1.5.0/docs/api/>

Cette section décrit la forme que doivent prendre les commentaires pour être pris en compte par javadoc

4.1 Documentation d'une méthode

Voici un exemple de documentation d'une méthode :

```
/**
 * La methode ajoutePomme permet de rajouter une pomme dans le rayon.
 * @param t le nombre de pommes a ajouter dans le rayon.
 * @return objet de type java.lang.Integer contenant le nombre
 * de pommes effectivement ajoutees
 */
public Object ajoutePomme(ChangeRayon t) {
    ...
}
```

Le commentaire doit commencer par `/**`, les `*` en début de ligne ne sont pas pris en compte. Les arguments de la méthode sont documentés à l'aide de la déclaration `@param <nom-du-paramètre> <description>`. La valeur de retour est documentée à l'aide de la séquence : `@return <description>`. Cela va produire le résultat suivant :

- tout d'abord une nouvelle entrée dans la section Method Summary :

Method Summary	
java.lang.Object	ajoutePomme(ChangeRayon t) La methode ajoutePomme permet de rajouter une pomme dans le rayon.
static void	ajoutePomme(ChangeRayon t)

- mais également une nouvelle entrée dans la section Method Details :

Method Detail

ajoutePomme

```
public java.lang.Object ajoutePomme(ChangeRayon t)
```

La methode ajoutePomme permet de rajouter une pomme dans le rayon.

Specified by:

[ajoutePomme](#) in interface [Rayon](#)

Parameters:

t - le nombre de pommes a ajouter dans le rayon.

Returns:

objet de type java.lang.Integer contenant le nombre de pommes effectivement ajoutees

4.2 Documentation d'une classe

On peut de même mettre un commentaire similaire juste avant la déclaration d'une classe. Ce commentaire sera placé en tête de la classe.

```

/**
 * Cette classe represente un type particulier de rayon.
 * @author Fabrice Derepas
 */
public class FruitsEtLegumes
    extends java.rmi.server.UnicastRemoteObject
    implements boutique.Rayon
{
    ...
}

```

4.3 Exécuter javadoc

Pour exécuter javadoc en ligne de commande, il suffit de taper, si toutes les sources sont dans le répertoire src :

```
javadoc -classpath src ex2
```

Les fichiers html sont alors générés dans le répertoire courant.

4.4 Création d'une cible ant

On peut également créer une cible ant qui appelle javadoc :

```

<target name="doc">
  <javadoc packagenames="ex2.*"
    sourcepath="src"
    defaultexcludes="yes"
    destdir="docs"
    author="true"
    version="true"
    use="true"
    windowtitle="Exemple 2">
    <doctitle><![CDATA[<h1>Exemple 2</h1>]]></doctitle>
    <bottom><![CDATA[<i>Copyright &#169; 2005 Fabrice Derepas.</i>]]></bottom>
  </javadoc>
</target>

```

Ici l'appel à javadoc va mettre comme nom de document Exemple 2, et écrira en bas de chaque page Copyright © 2005 Fabrice Derepas..

4.5 Faire le test

Téléchargez l'archive :

http://www.derepas.com/java/cours2_exercice_1.jar

Exécuter la cible doc dans le fichier build.xml. On remarque qu'un répertoire docs a été créé. On peut alors visualiser le fichier index.html.

Pour plus de détails se reporter à l'url :

<http://java.sun.com/j2se/javadoc/writingdoccomments/>

5 Exercices

5.1 Exercice 5.1

Q1 Ecrire une classe `Signataire` qui génère une clé RSA de 1024 bits et signe un fichier.

Q2 Ecrire une classe `Chargeur` qui vérifie la signature d'un fichier étant donné la clé publique utilisée pour générer la signature.

Pour comparer les signatures on utilisera la méthode `Signature.verify`.

Ces classes peuvent ensuite être utilisées pour effectuer des téléchargements sûrs sur internet.

5.2 Exercice 5.2

Q1 Récupérer le certificat :

`http://www.derepas.com/java/derepas.cer`

et l'installer dans un keystore.

Q2 Téléchargez le fichier jar

`http://www.derepas.com/java/cours5_exercice2.jar`

l'exécuter sans politique de sécurité. L'exécuter avec une politique de sécurité interdisant tout.

Q3 Exécuter le fichier jar précédent avec une politique de sécurité autorisant l'écriture des classes certifiées par le certificat de la première question. Pour spécifier comment utiliser le keystore de la question 1, se reporter à l'url :

`http://java.sun.com/j2se/1.5.0/docs/guide/security/PolicyFiles.html`