Le Modèle de Sécurité dans JAVA

$Id : javaSecurity.lyx 1565 2008-10-22 13 :57 :30Z phil $

22 octobre 2008

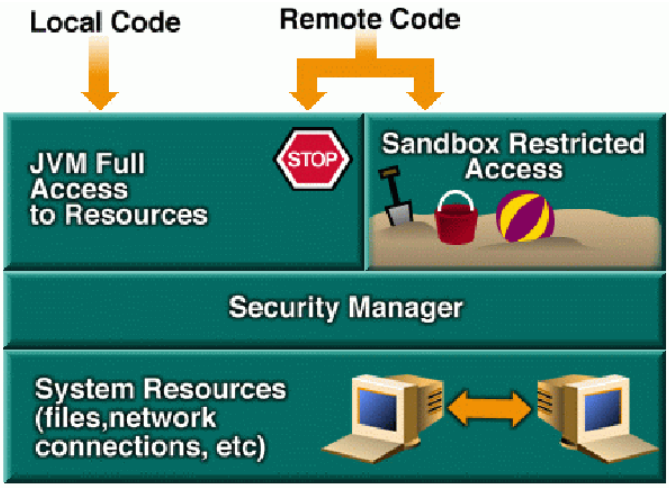
Université de Cergy-Pontoise,
2 rue A. Chauvin
95302 Cergy-Pontoise cedex
e-mail laroque@u-cergy.fr

Table des matières

1 Historique

1.1 La sécurité dans le JDK 1.0

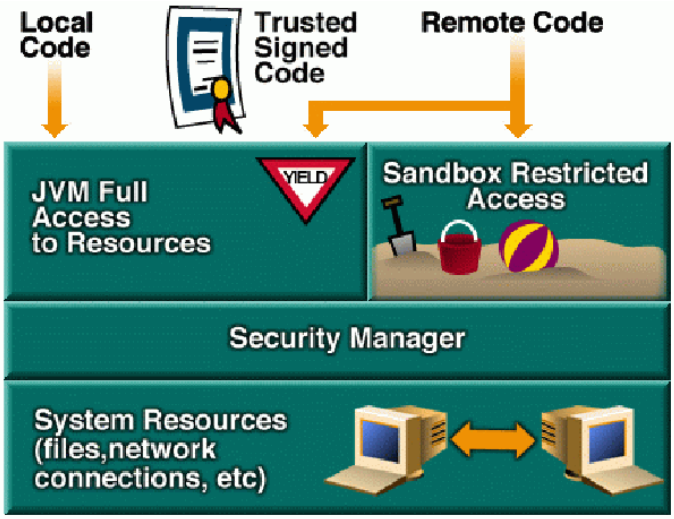
Binaire :
- Tout code local a un accès non limité aux ressources
- Tout code distant est exclu de l'accès à ces ressources, il tourne dans la "sandbox"



1.2 La sécurité dans le JDK 1.1

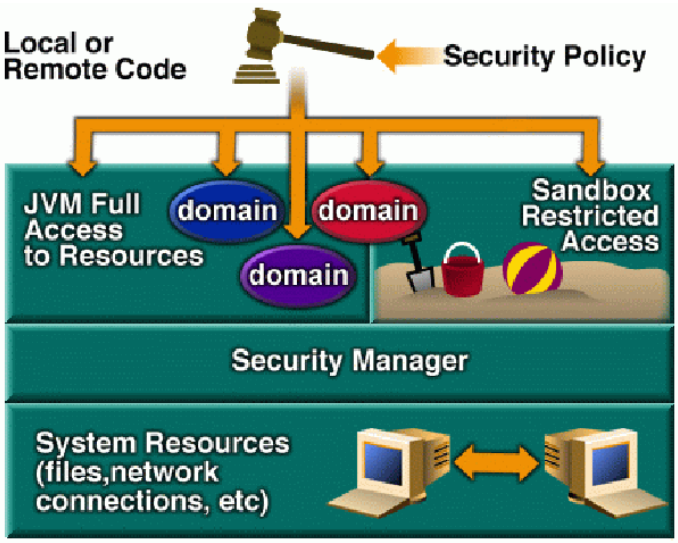
- La notion d'applet signée va sophistiquer le modèle
- Une applet signée a les mêmes droits que du code local

– Elle est livrée dans un fichier archive JAR, avec la signature électronique
– Encore très binaire



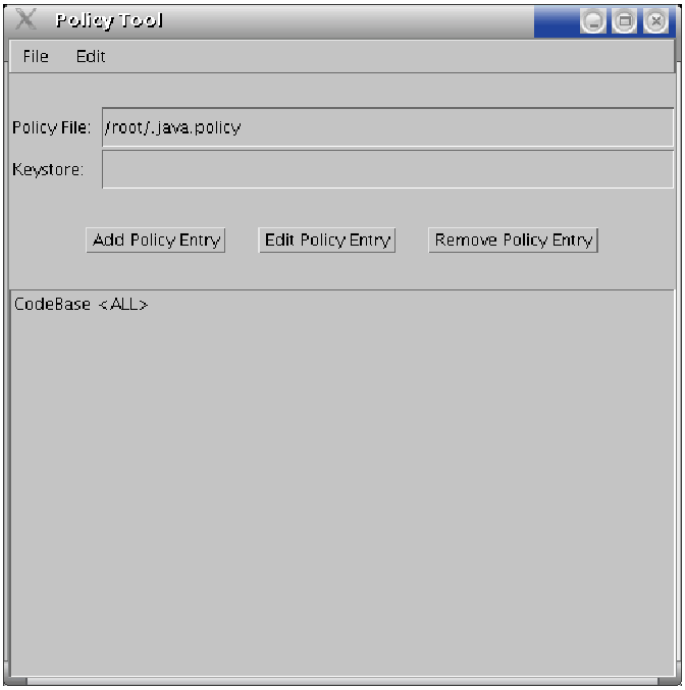
1.3 La sécurité dans la J2XE

Beaucoup plus sophistiqué :
– Tout code, local comme distant, est soumis à des règles d'accessibilité.
– Ces règles sont regroupées dans un *policy file*.
– Des outils sont fournis pour constituer/ modifier ce fichier (`policytool`). Il contient des ACL (*Access Control List*).
– Chaque ACL définit l'accessibilité à une (un groupe de) ressource(s) par un *domaine* (ensemble de classes).



2 Le contrôle des applets

- Le contrôle d'accès est fait à l'aide d'un security manager.
- Si l'accès est refusé, une exception est levée.
- Pour permettre l'accès à la ressource à l'applet, il faut modifier le *policy file* ou le *security manager*.
- La modification du policy file peut se faire à l'aide du *policy tool* (commande `policytool`) :



2.1 Exemple simple

On écrit une applet qui va tenter d'écrire un (nouveau) fichier dans le répertoire home (donc côté client).

2.1.1 Code source Java

Le code source de l'applet a l'allure suivante :

```
// $Id: AppletSecurity.java 1565 2008-10-22 13:57:30Z phil $

package pl.security;
```

```
import java.awt.*;
import java.io.*;
import java.applet.*;

/**
 * Test for java applet security.
 * This applet tries to write a local (client) file. Without ACL, a
 * SecurityException is thrown. With the correct ACL, the code executes
 * normally
 * @author laroque@u-cergy.fr
 * @version $Rev: 1565 $
*/
public class AppletSecurity extends Applet {
  String myFile = "newFile";
  File f = new File(myFile);
  DataOutputStream dos;

  public void init() {
    String osname = System.getProperty("os.name");
  }

  public void paint(Graphics g) {
    try {
      dos = new DataOutputStream
        (new BufferedOutputStream(new FileOutputStream(myFile),128));
      dos.writeChars("The applet could write local FS\n");
      dos.flush();
      g.drawString("Successfully wrote to the file named " + myFile, 10, 10);
    }
    catch (SecurityException e) {
      g.drawString("caught security exception: " + e, 10, 10);
    }
    catch (IOException ioe) {
      g.drawString("caught i/o exception", 10, 10);
    }
  }
}
```

2.1.2 Code source HTML

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
```

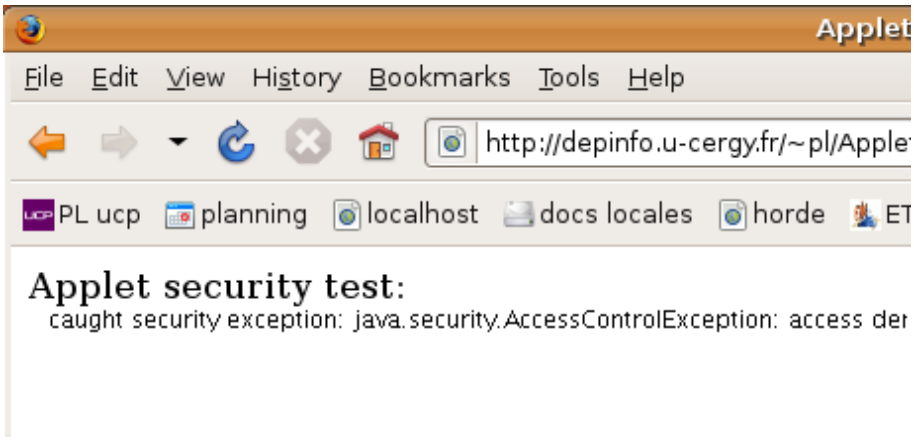
```
<head>
  <meta content="text/html; charset=ISO-8859-1" http-equiv="content-type">
  <title>Applet security test</title>
</head>

<body>
Applet security test:

<br>
<applet code="pl.security.AppletSecurity"
      archive="pl.jar"
      alt="If you see this, then the applet couldn't start"
      height=100
      width=400
>
This browser ignores "applet" tags
</applet>
</body>
</html>
```

2.1.3 Fonctionnement par défaut

Sans précaution particulière, le chargement de la page (soit dans l'appletviewer lancé depuis le home directory, soit depuis un navigateur web) provoque comme on peut s'y attendre une exception :

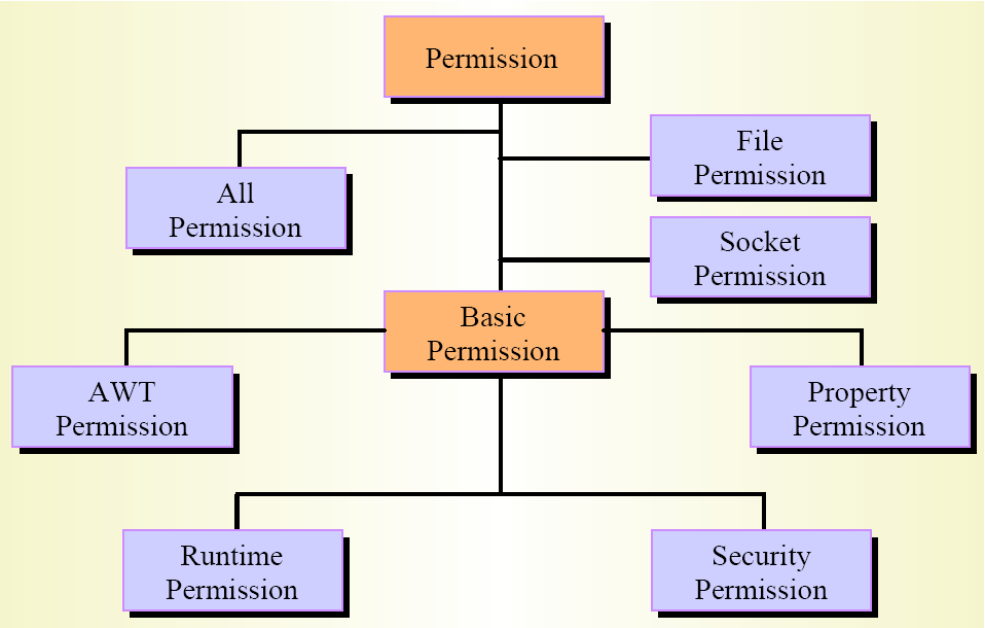


Il va falloir autoriser ce code à écrire dans le FS local. Pour cela, on va définir une ACL.

2.2 Ajout d'une ACL dans le policy tool

- Une ACL est constituée de trois paramètres :
  - Le type de permission (E/S, propriétés systèmes, etc.)
  - La ressource concernée (par exemple tel fichier/répertoire)
  - Les droits donnés (par exemple lecture et/ou écriture)
- On indique dans le champ "CodeBase" l'URL à laquelle on souhaite fournir un accès
- Il faut ensuite préciser le type de permission, le nom de la ressource concernée et le type d'opération qu'on autorise.

Les permissions existantes sont regroupées dans la figure suivante :



2.3 Contenu du policy file

Dans notre exemple, le fichier a l'allure suivante :

```
grant codeBase "http ://depinfo.u-cergy.fr/~pl/pl.jar" {
  permission java.io.FilePermission "/home/phil/newFile", "write";
};
```

- Si ce fichier a été sauvé sous le nom `~/myPolicy`, il faut indiquer au gestionnaire de sécurité (le *security file*, `~java/jre/lib/security/java.security`, où `~java` est le répertoire racine de JAVA) où il se trouve.
- On peut avoir plusieurs *policy files*. Chaque exemplaire doit faire l'objet d'une ligne de la forme suivante dans le fichier `java.security` :

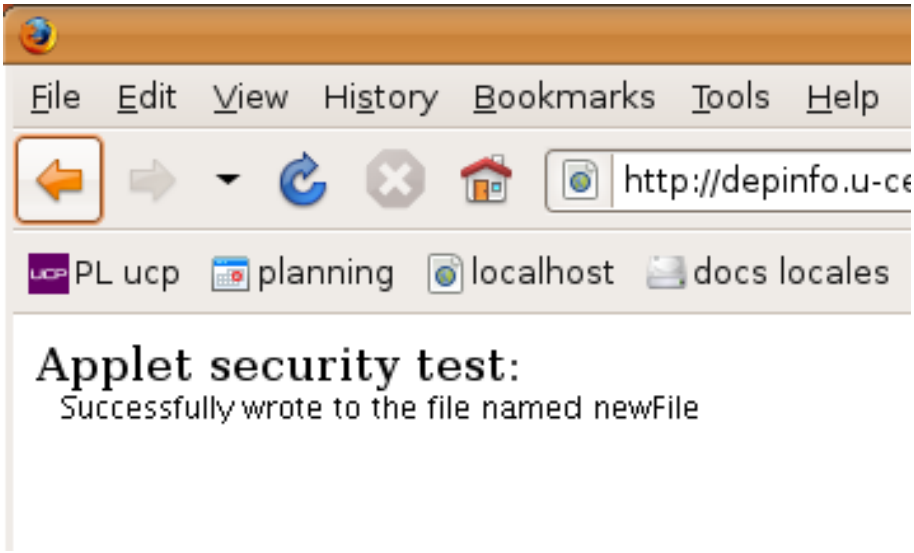
```
policy.url.1=file :${java.home}/lib/security/java.policy
```

```
policy.url.2=file :${user.home}/.java.policy
policy.url.3=file :${user.home}/myPolicy
```

– Enfin, on peut indiquer directement sur la ligne de commande que l'on désire exécuter l'applet avec les ACL définies dans le fichier `myPolicy` :

```
$ appletviewer -J-Djava.security.policy=~/myPolicy TheAppletURL.html
```

Après ajout de l'entrée précédente, on peut exécuter avec succès le code de l'applet de notre exemple :



3 Le contrôle des applications

– La différence principale avec les applets est que, par défaut, aucun gestionnaire de sécurité n'est actif pour les applications. On peut l'activer explicitement :

```
$ java -Djava.security.manager MonAppli
```

– Dans ce cas, l'accès à certaines propriétés (par exemple `user.home`) est refusé. Les droits par défaut sont donnés dans le system policy file, `~java/jre/lib/security/java.policy`.

– Dans le fichier ~/myPolicy précédent, il faut alors rajouter (par exemple) :

```
grant codeBase "file :${user.home}/src/java/-" {
  permission java.util.PropertyPermission "user.home", "read";
  permission java.util.PropertyPermission "java.home", "read";
};
```

– On peut également (à la place) spécifier explicitement le policy file à utiliser :

```
$ java -Djava.security.manager \
   -Djava.security.policy=mypolicy MonAppli
```

`N.B.` : le "-" en fin d'URL permet de donner les mêmes droits à toute l'arborescence de répertoires dont la racine est spécifiée par l'URL.

4 Les outils de gestion de la sécurité

– L'utilisation intensive d'Internet implique la nécessité d'être certain de la provenance d'un code (notamment applet) avant son utilisation locale.
– Les outils principaux permettant d'effectuer cette vérification sont au nombre de trois : les signatures électroniques, les certificats et les "keystores"

4.1 Signatures électroniques

Reposent sur des algorithmes a clés dissymétriques (publique / privée)
– On envoie sa clé publique à tous ses interlocuteurs
– On signe un document avec sa clé privée (qu'on est seul à connaître !)
– Le destinataire utilise la clé publique pour vérifier que l'on est bien l'auteur du document (par confrontation avec la clé privée)
– Le problème est repoussé à être certain de la validité de la clé publique ! C'est là qu'interviennent les certificats

4.2 Certificats

Un certificat contient :
– La clé publique de l'entité concernée.
– Le "*distinguished name*" de l'entité concernée (le propriétaire, *owner*), i.e.
  – son nom
  – son organisation (et sa sous-organisation au sens LDAP)
  – ses ville, état/province, pays et code postal
– Une signature électronique de l'émetteur du certificat (*issuer*) qui en certifie la validité.
– Le *distinguished name* de l'émetteur.

On peut souvent vérifier la clé publique d'une entité inconnue en vérifiant celle de son émetteur, et en remontant la chaîne jusqu'à arriver à un émetteur auquel on fait confiance.

4.3 Autorités de certification

Si cette vérification est impossible, on peut toujours créer une "empreinte" (*fingerprint*) du certificat, puis appeler l'émetteur en lui demandant de lui envoyer

son certificat. En comparant les deux empreintes, il peut s'assurer de la validité du premier...

Un émetteur peut certifier lui-même sa signature (*self-signed certificate*), en le signant à l'aide de sa clé privée. Bien entendud il doit être connu du destinataire comme fiable.

Sinon, l'authentification passe par un "tiers de confiance" (*Certification Authority*, CA) : l'émetteur envoie un *certificate signing request* (CSR) auto-signé au CA, qui en vérifie la validité. Il émet alors un certificat concernant l'émetteur qu'il signe lui-même. Ainsi, toute personne qui fait confiance au CA fait confiance à l'émetteur.

Les CA sont organisés de manière arborescente.

Le JDK fournit un certain nombre d'outils pour la gestion des signatures et des certificats.

4.4 Stockage des clés (*keystores*)

Dans des bases protégées par mot de passe.

Deux types d'information :

- Les certificats auxquels on fait confiance (*trusted certificates*) : les documents signés de la clé publique correspondante sont considérés comme authentiques.
- Des couples clé privée/certificat contenant la clé publique correspondante.

Chaque entrée est repérée par un alias unique.

4.5 Outils et moyens techniques dans le JDK

La constitution de signatures et de certificats peut être faite de plusieurs façons :

- A l'aide d'outils livrés avec la J2SE (et la J2EE) : `jar`, `jarsigner`, `keytool`
- A l'aide d'API incluses dans le JDK (sauf, pour l'instant, pour la création de certificats)
- Par un mélange des deux techniques.

Avant de pouvoir exécuter un code signé en local, il est nécessaire d'avoir (ou d'obtenir) un certificat validant cette signature, et de l'incorporer dans son *keystore*.

4.6 Mise en œuvre de la signature par les API du JDK

Ces classes permettent à l'émetteur de

- générer des paires "clé publique / clé privée" ;
- générer une signature électronique pour un document, à partir de la clé privée ;
- joindre la signature (et la clé publique) aux données signées.

Elles permettent au destinataire de
- importer une clé publique dont il a besoin ;
- vérifier l'authenticité d'une signature.

4.7 Mise en œuvre de la signature par les outils du JDK

Ces outils permettent à l'émetteur de
- générer des paires "clé publique / clé privée" ;
- générer une signature électronique pour un document, à partir de la clé privée ;
- joindre la signature (et la clé publique) aux données signées.

Ils permettent au destinataire de
- importer une clé publique dont il a besoin ;
- vérifier l'authenticité d'une signature.

5 Signature de code et permissions

Tâches côté fournisseur du code :
- placer le code dans une archive JAR ;
- signer l'archive ;
- exporter le certificat correspondant à la clé privée ayant signé l'archive.

Tâches côté utilisateur du code :
- vérifier que l'application ne peut normalement pas accéder à une ressource donnée si elle s'exécute sous la responsabilité d'un security manager ;
- importer le certificat relatif au code utilisé dans son keystore ;
- modifier le policy file pour fournir l'accès nécessaire au code authentifié par ce certificat ;
- vérifier que l'application peut alors accéder à la ressource demandée.

5.1 Un exemple simple

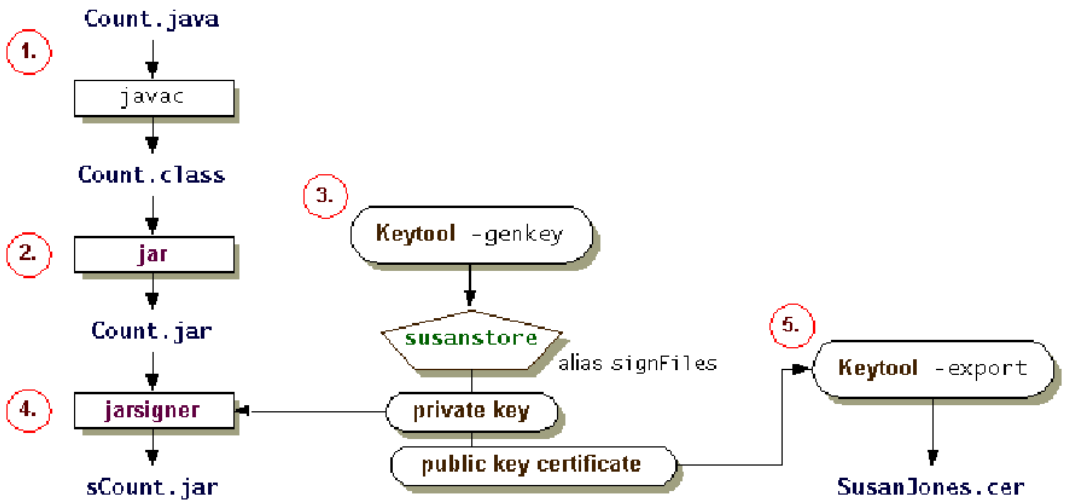
L'exemple repose sur une classe Count qui donne le nombre de caractères d'un fichier passé en paramètre :

```
import java.io.*;
public class Count {
    public static void countChars(InputStream in) throws IOException {
        int count = 0;
        while (in.read() != -1)
            count++;
        System.out.println("Counted " + count + " chars.");
    }
```

```
    public static void main(String[] args) throws Exception {
        if (args.length >= 1)
            countChars(new FileInputStream(args[0]));
        else
            System.err.println("Usage : Count filename");
    }
}
```

5.2 Les étapes côté fournisseur

L'émetteur prétend être Susan Georges. Les étapes à suivre sont résumées dans la figure suivante :



5.3 Création du JAR et génération des clés

1. $ jar cvf Count.jar Count.class

```
$ keytool -genkey -alias signFiles -keypass \
   kpi135 -keystore susanstore -storepass ab987c
```

Options de genkey :

- `-genkey` : indique à keytool que l'on souhaite générer des clés
- `-alias signFiles` : identifie cette entrée de manière unique sous l'alias signFiles
- `-keypass kpi135` : fournit un mot de passe lié à la clé privée qui va être générée. Il sera demandé pour tout accès à cette clé privée. Il peut être identique à celui du *keystore*.

– `-keystore susanstore` : indique le nom du *keystore* à utiliser (/créer). Par défaut il est placé dans `~/.keystore`
– `-storepass ab987c` : fournit un mot de passe pour le *keystore*

Cette commande crée un certificat auto-signé valide 90 jours (c'est le défaut, modifiable par `-validity`)

5.4 Signature du fichier JAR

```
$ jarsigner -keystore susanstore -signedjar \
   sCount.jar Count.jar signFiles
```

Le système demande alors le mot de passe du keystore et de la clé. S'ils sont corrects, le fichier `sCount.jar` est généré.

5.5 Exportation d'un certificat

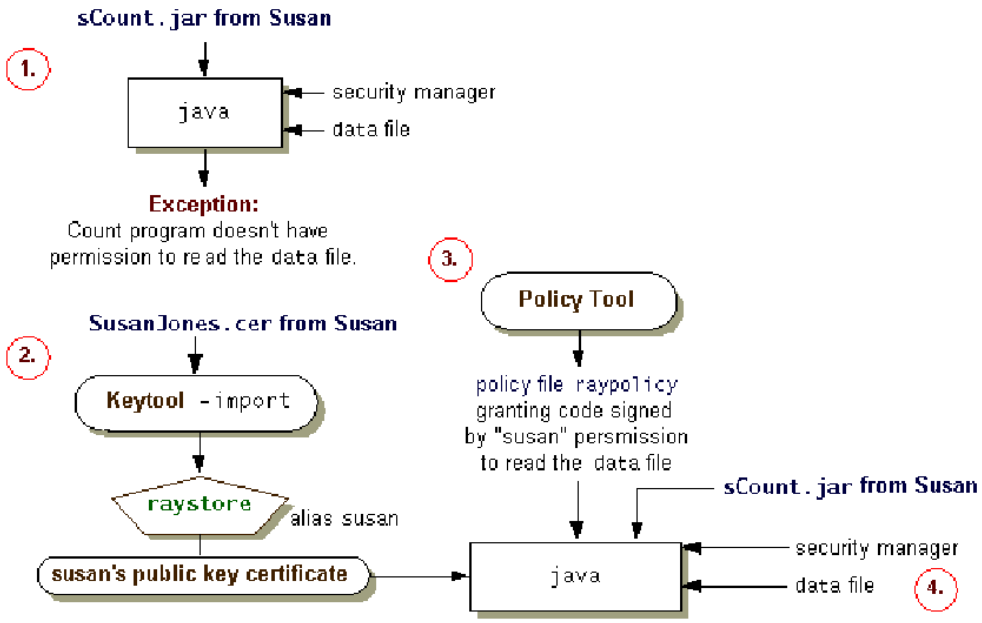
On produit un certificat par l'option `-export` de keytool :

```
$ keytool -export -keystore susanstore \
   -alias signFiles -file SusanJones.cer
```

Là encore, le mot de passe du keystore est demandé. Le fichier généré, `SusanJones.cer`, peut être envoyé au destinataire.

5.6 Les étapes côté utilisateur

Elles sont résumées dans la figure suivante :



5.7 Importation du certificat

L'utilisateur a reçu l'archive signée. Comme elle contient le certificat, il faut d'abord l'extraire de l'archive :

```
$ keytool -import -alias susan -file \
   SusanJones.cer -keystore raystore
```

Ceci crée un nouveau keystore (`raystore` n'existait pas), le protège par un mot de passe et affiche l'empreinte du certificat.

- On peut alors la comparer à celle fournie par des sources de confiance sur le certificat de Susan Georges (par exemple en lui téléphonant et en lui demandant de donner la suite des chiffres de l'empreinte de sa version de l'archive signée). On l'obtient par la commande

  ```
  $ keytool -printcert -file SusanJones.cer
  ```

- Si les empreintes sont identiques, on peut faire confiance à ce certificat et l'inclure dans son keystore.

5.8 Création d'une nouvelle ACL

Il faut modifier le *policy file* pour autoriser l'application à accéder aux ressources souhaitées (par exemple, lire les fichiers contenus dans le repertoire `test.d`).

Pour cela, il faut indiquer dans le policy tool

- que l'on va utiliser le keystore raystore pour le certificat
- le type d'accès souhaité pour tout code possédant ce certificat de Susan Georges

L'ACL constituée a l'aspect suivant :

```
grant signedBy "susan" {
  permission java.io.FilePermission
 "${user.home}/src/tex.d/javaSecurity.d/test.d/*", "read"
};
```

6 Echange de documents signés

La signature électronique est un excellent moyen d'assurer l'originalité d'un document.

Le principe est proche de celui mis en œuvre pour la signature d'un code JAVA.

La description qui suit suppose la transmission d'un contrat entre l'émetteur, *Spirou*, et le destinataire, *Fantasio*.

6.1 Etapes côté émetteur

- Introduction du document (le fichier contract) dans une archive JAR

```
$ jar cvf Contract.jar contract
```

- Génération des clés de l'émetteur (si elles n'existent pas encore)

```
$ keytool -genkey -alias signLegal \
   -keystore spiroustore
```

- (*optionnel*) Génération d'une demande de certificat (*Certificate Signing Request*, CSR)
- Signature de l'archive

```
$ jarsigner -keystore spiroustore -signedjar \
   sContract.jar Contract.jar signLegal
```

- Exportation du certificat contenant la clé publique

```
$ keytool -export -keystore spiroustore \
   -alias signLegal -file spirou.cer
```

- Fourniture de l'archive signée et du certificat au destinataire

6.2 Etapes côté destinataire

- Obtention de l'archive (`sContract.jar`) et du certificat (`spirou.cer`)
- Importation du certificat (après vérification éventuelle de sa validité par comparaison à partir de l'empreinte d'un autre certificat "*trusted*" (`autre.cer`))

```
$  keytool -import -alias spirou -file \
   spirou.cer -keystore fantasiostore
$ keytool -printcert -file autre.cer
         186 Fri May 26 14 :53 :28 CEST 2000 META-INF/SIGNLEGA.SF
         973 Fri May 26 14 :53 :28 CEST 2000 META-INF/SIGNLEGA.DSA
           0 Fri May 26 14 :48 :26 CEST 2000 META-INF/
smk     1023 Fri May 26 14 :48 :02 CEST 2000 contract

  s = signature was verified
  m = entry is listed in manifest
  k = at least one certificate was found in keystore
  i = at least one certificate was found in identity scope

jar verified.
```

- Vérification de la signature de l'archive

```
$ jarsigner -verify -verbose -keystore \
   fantasiostore sContract.jar
```

– Extraction du contrat à partir de l'archive vérifiée

```
$ jar xvf sContract.jar contract
```