Sécurité en Java 2

Université de Nice - Sophia Antipolis
Version 4.9 – 24/10/07
Richard Grin

Plan de cette partie

- Introduction
- Où intervient la sécurité
- Comment améliorer la sécurité des programmes
- Politique de sécurité
- Cryptographie
- Les outils pour la sécurité
- Compléments
- JAAS
- JCA/JCE

Introduction

La sécurité en informatique

- Authentifier les utilisateurs, ou les programmes avec des signatures
- Contrôler les autorisations d'accès aux ressources (système et utilisateur) d'une entité authentifiée ; par exemple, empêcher des lectures, suppressions ou des modifications non autorisées sur des fichiers
- Assurer par le cryptage la confidentialité des transmissions ou des données stockées

Le soucis de la sécurité

- Dès sa conception, Java a attaché beaucoup d'importance à la sécurité
- Les classes Java peuvent être chargées dynamiquement depuis des machines non sûres et il faut un moyen de contrôler les actions exécutées par ces classes
- En particulier, l'exécution des *applets* récupérées sur le Web présente un grand risque

Interdire pour protéger

- Pour effectuer la protection des ressources, Java interdit aux classes Java non sûres (essentiellement celles qui viennent d'une autre machine) d'effectuer des opérations potentiellement dangereuses :
  - obtenir des informations sur l'utilisateur ou la machine locale
  - écrire sur le disque local
  - se connecter sur une tierce machine non connue de l'utilisateur
  - ...

Le bac à sable des applets

- Par défaut les applets s'exécutent dans un « bac à sable » duquel elles ne peuvent sortir et qui leur interdit toute action dangereuse
- Mais les contraintes imposées par le bac à sable sont trop strictes pour certaines applets qui souhaitent obtenir des informations locales ou écrire des données sur les disques locaux
- Java 2 permet d'assouplir cette politique de sécurité

Sujet du cours

- Les exemples suivants montrent les restrictions imposées par le bac à sable
- Il faudra contourner ces restrictions si on veut faire fonctionner les applications
- Mais il ne faut pas pour cela ouvrir la porte aux utilisateurs mal intentionnés
- C'est le sujet de la 1ère partie de ce cours : comment permettre juste ce qu'il faut pour qu'une application ou applet (et pas les autres) puisse effectuer des opérations qui lui sont interdites normalement

Exemple

- Applet qui pose des QCM et gère les résultats
- Pour cela, l'utilisateur se connecte à une adresse Web et récupère une page qui contient une applet qui affiche les questions
- Pour obtenir l'identité de la personne qui répond, l'applet contient l'instruction **`System.getProperty("user.name");`**
- Le bac à sable dans lequel s'exécutent les applets interdit la lecture de la propriété système « **`user.name`** »

Exemple (2)

- Cette même application range les résultats des étudiants dans une base de données
- Le SGBD est placé sur une autre machine que le serveur Web
- L'applet se voit refuser l'accès à ce SGBD car le bac à sable interdit les connexions réseaux sur une autre machine que celle du serveur Web d'où vient l'applet

Exemple (3)

- L'applet souhaite écrire sur le disque de l'utilisateur des informations sur les QCM auxquels il a déjà répondu
- La politique de sécurité liée au bac à sable interdit à l'applet d'écrire un fichier sur le disque local de l'utilisateur
- En fait une applet ne peut lire que les fichiers qui sont placés dans le répertoire d'où elle vient ou dessous (et placés dans le même jar si elle est dans un jar)

Il n'y pas que les applets...

- Des restrictions comme celles imposées par le bac à sable des applets se retrouvent aussi
  - pour les applications qui chargent des classes dynamiquement depuis une autre machine (RMI en particulier)
  - pour les applications qui décident, pour une raison quelconque, d'installer un gestionnaire de sécurité
  - lorsque l'utilisateur l'a décidé en lançant l'application

Où intervient la sécurité

Plan

- Dans le langage Java
- Dans les API standard
- Injection de code SQL
- Vérificateur de *bytecode*
- Chargeur de classes
- Les différentes versions de Java et la sécurité

La sécurité dans le langage

- Vérifications à la compilation :
  - Java est fortement typé
  - Pas d'arithmétique des pointeurs
  - Les variables non initialisées sont inutilisables
  - Protection des variables d'état (**`private`**,…)
  - Possibilité de déclarer **`final`** les variables, méthodes et classes

La sécurité dans le langage

- Vérifications à l'exécution :
  - Contrôle des débordements dans les tableaux
  - Contrôle des *casts*
  - Vérification des classes au chargement

La sécurité dans les API

- Les API liés à la sécurité permettent de
  - délimiter ce qui est autorisé pour chaque programme Java, selon
    - le lieu d'où les classes ont été chargées
    - l'utilisateur qui les a signées
    - l'utilisateur qui l'exécute (JAAS ; pas pris en compte pour cette partie)
  - protéger la confidentialité des informations par le cryptage

Écrire des programmes sûrs

- Favoriser l'encapsulation : les variables doivent être **`private`**, sauf raison contraire
- Éviter autant que possible les variables **`protected`**
- Éviter de passer des références vers des variables sensibles en sortie ou en entrée des méthodes ; cloner (ou copier) les valeurs avant de les passer
- Déclarer **`final`** les méthodes ou les classes dont le fonctionnement ne doit pas être modifié

Injection de code SQL

- Danger si le texte d'une requête SQL comporte une partie fournie par l'utilisateur

Exemple d'injection de code SQL

- Un programme demande le nom et le mot de passe d'un utilisateur et les range dans 2 variables **nom** et **mdp**
- Il vérifie la validité du nom et du mot de passe par cette requête qui doit renvoyer une ligne :

```
"select * from utilisateur"
+ " where nom = '" + nom
+ "' and mdp = '" + mdp + "'"
```

- Quel est le problème ?

Le problème

- Un pirate sait qu'un des utilisateurs autorisés s'appelle Dupond
- Il saisit « **Dupond' --** » pour le nom et « **a** » pour le mot de passe
- La requête devient :

```
select * from utilisateur
where nom = 'Dupond' --' and mdp = 'a'
```

- Mais « -- » indique un commentaire avec le SGBD utilisé ; donc la requête exécutée sera :

```
select * from utilisateur
where nom = 'Dupond'
```


Les parades

- Toujours vérifier la saisie d'un utilisateur avant de s'en servir pour construire une requête SQL
- Pour l'exemple, il aurait suffit d'interdire le caractère « ' »
- Avec JDBC, il est plus sûr d'utiliser une **PreparedStatement** et les méthodes setXXX plutôt que de concaténer des **String** et d'utiliser **Statement** ; la valeur de la variable ainsi saisie ne sera jamais considérée comme un mot clé SQL

Exemple

- Si le code est

```
PreparedStatement stmt2 = conn.prepareStatement(
   "select * from utilisateur "
   + " where nom = ? + and mdp = ?");
stmt2.setString(1, matr);
rset = stmt2.executeQuery();
```

- La saisie par l'utilisateur de « **Dupond' --** » pour le login et de « a » pour le mot de passe lancera la requête SQL suivante :

```
select * from utilisateur
where nom = 'Dupond'' --' and mdp = 'a'
```

qui ne renverra aucune ligne

Vérificateur de *bytecode*

- Il fait partie de la JVM
- Son rôle est d'examiner le *bytecode* des classes au moment de leur transformation en objet **Class** par le chargeur de classe
- Il vérifie que la structure des classes chargées par la JVM est correcte
- En plus des avantages pour la sécurité, il évite ainsi à la JVM d'effectuer certaines vérification à l'exécution (par exemple, les dépassements de capacité de la pile), ce qui améliore les performances

Les 4 passes de la vérification

- Vérification de la structure du fichier
- Vérifications qui ne dépendent pas du code particulier des méthodes
- Vérification du code de chaque méthode
- Vérifications sur le code des méthodes, qui sont repoussées pour des raisons d'efficacité jusqu'au moment où le code est exécuté pour la première fois

Passes 1 et 2

- Structure du fichier :
  - nombre « magique » correct au début du fichier (**`0xCAFEBABE`**)
  - pool des constantes a une structure correcte
- Vérifications qui ne dépendent pas du code particulier des méthodes :
  - pas de classe fille d'une classe **`final`**
  - pas de méthode **`final`** redéfinie
  - toute classe a bien une classe mère (sauf **`Object`**)
  - références vers le pool des constantes sont valables

Passe 3 (1)

- Jusqu'à maintenant les vérifications ne prenaient en compte que la classe vérifiée
- Les passes 3 et 4 prennent en compte les classes utilisées par la classe vérifiée
- La passe 3 nécessite une analyse du cheminement d'exécution ; elle est la vérification la plus complexe

Passe 3 (2)

- La répartition des tâches entre les passes 3 et 4 n'est pas complètement spécifiée et peut dépendre des JVM
- Le principe général est qu'on essaie de repousser dans la passe 4 les vérifications qui nécessiteraient le chargement de nouvelles classes
- Par exemple, si le profil de f est « **`B f()`** », « **`B b = f();`** » peut être vérifié par la passe 3 sans charger la classe B, mais « **`A b = f();`** » va nécessiter le chargement des classes A et B pour s'assurer que B est une classe fille de A

Passe 3 (3)

- Vérifications effectuées :
  - variables locales pas utilisées sans être initialisées
  - les méthodes sont appelées avec des paramètres corrects (nombre et type) et retournent les bons types
  - les accès aux membres (public, protected,…) sont autorisées
  - les affectations sont effectuées avec les bons types

Passe 4

- Les vérifications effectuées sont celles de la passe 3 qui nécessitent le chargement de classes
- La JVM peut alors remplacer les instructions qui ont déclenché la vérification par d'autres instructions spéciales de telle sorte qu'aucune vérification ne sera plus déclenchée par la suite

Classes vérifiées par le vérificateur de *bytecode*

- Les classes de l'API standard (et des extensions) ne sont pas vérifiées
- Depuis la version 1.2, les classes venant du *classpath* sont vérifiées

Protection par le chargeur de classes

- Les chargeurs de classes sont les piliers de la gestion de la sécurité en Java 2
- Lors du chargement des classes ils mettent en place tout le contexte qui sera utilisé par les gestionnaires de sécurité pour délimiter ce qui sera autorisé pendant l'exécution
- En particulier, ils associent les classes chargées à un domaine de protection (voir suite du cours)
- Les chargeurs de classes sont étudiés en détails dans un autre cours

Espaces de noms liés aux chargeurs de classes

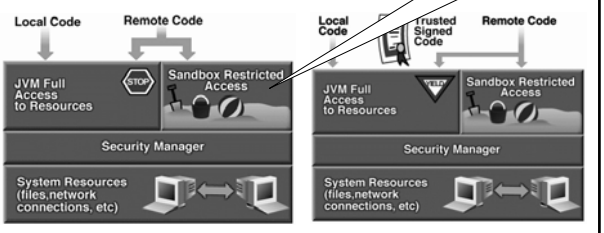
- Un chargeur de classes définit un espace de noms : 2 classes de même nom chargées par 2 chargeurs de classes différents sont considérées comme différentes
- Les navigateurs créent un chargeur de classes différent pour chaque *codebase*
- Ainsi, par exemple, 2 applets venant de 2 URL différents ont des chargeurs de classes différents et ne partagent pas les mêmes classes distantes (même si elles ont le même nom)

La sécurité dans les différentes versions

- La politique de sécurité de Java a évolué au cours des différentes versions : JDK 1.0, JDK 1.1 et Java 2
- Au fil des versions Java a ajouté de la souplesse aux possibilités offertes aux utilisateurs, développeurs et administrateurs

Les versions de la politique de sécurité de Java 1

Bac à sable des applets

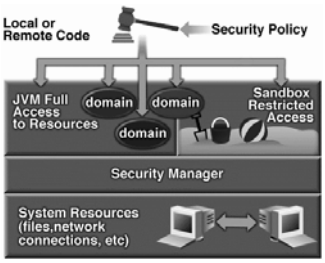


JDK 1.0
Tout ou (presque) rien

JDK 1.1
Tout (aussi pour le code signé) ou (presque) rien

(Images du tutoriel en ligne de Sun)

La politique de sécurité de Java 2



Plusieurs domaines de sécurité avec plus ou moins d'autorisations (de tout jusqu'à presque rien)

Les domaines de sécurité sont déterminés par les chargeurs de classe

Politiques de sécurité en Java 2

Plan

- Les grands principes
- Les acteurs
- Les fichiers de sécurité

Contrôle des applications

- Par défaut les applications locales, c'est-à-dire, lancées par une classe placée dans le *classpath* ne sont pas contrôlées
- Toutes les autres sont contrôlées automatiquement par un gestionnaire de sécurité
- On peut aussi faire contrôler les applications locales si on le souhaite

Principe général de la sécurité en Java 2

- Les droits d'un code ne sont pas écrits en dur dans du code
- Ils sont définis par une politique/police de sécurité enregistrée à part dans des fichiers

Exemple de fichier de police de sécurité

```
keystore ".keystore";
grant signedBy "toto" {
  permission java.io.FilePermission
  "${user.home}${/}-", "read";
};
grant codeBase
  "http://deptinfo.unice.fr/~toto/*" {
  permission java.util.PropertyPermission
  "user.home", "read";
};
```

N'oubliez pas les « ; » !

Domaine de sécurité

- *Quand elle est chargée dans la JVM*, une classe se voit associer un domaine de sécurité
- Un domaine de sécurité ajoute des permissions à une classe (en plus de ce que permet le bac à sable)
- C'est une sorte de bac à sable élargi et assoupli

Ce qui détermine un domaine de sécurité

- Un domaine de sécurité est déterminé par
  - l'origine de la classe : d'où vient-elle, qui l'a signée
  - la politique de sécurité au moment où la classe est chargée par le chargeur de classes
- La politique de sécurité peut être modifiée en cours d'exécution, mais ça ne change pas les permissions associées aux classes déjà chargées

Exemple

- L'utilisateur qui a lancé l'exécution a ceci dans son fichier de police de sécurité :

```
grant signedBy "toto" {
  permission java.io.FilePermission
          "${user.home}${/}-", "read";
};
```

- Alors, si une classe signée par toto est chargée en mémoire, le domaine de sécurité qui lui est associé contiendra la permission de lire les fichiers placés dans toute l'arborescence du répertoire HOME de l'utilisateur

Toutes les API sont concernées

- Toute action potentiellement dangereuse effectuée par une méthode des API Java est contrôlée par le gestionnaire de sécurité
- Les paquetages **`java.io`** et **`java.net`** sont particulièrement concernés, mais aussi les autres paquetages comme **`java.lang`**, **`java.awt`** ou **`javax.swing`**

API et outils spécifiquement destinés à la sécurité

- Paquetage **`java.security`** et ses sous-paquetages : classes pour les permissions, les signatures, les polices de sécurité,...
- Outils pour la sécurité fournis avec le SDK : *jarsigner*, *keytool*, *policytool*

Les acteurs de la politique de sécurité

Acteurs de la politique de sécurité

- Politique de sécurité
- Domaines de sécurité
- Gestionnaire de sécurité et contrôleur d'accès
- Contexte d'appel

- La *politique de sécurité* détermine le *domaine de sécurité* d'une classe au moment de son chargement
- Si une méthode veut exécuter une action potentiellement dangereuse, le *gestionnaire de sécurité* charge le *contrôleur d'accès* de vérifier qu'elle en a le droit dans le *contexte de l'appel*

Politique de sécurité

- Elle est déterminée par la lecture de fichiers de propriétés au démarrage de la JVM (voir section « Fichiers de police de sécurité »)
- Elle est utilisée au moment du chargement d'une classe pour lui associer un domaine de sécurité
- La vérification des droits de la classe n'utilise ensuite plus que ce domaine de sécurité

Classe Policy

- La politique de sécurité en cours à un moment donné est représentée par la classe **`Policy`**
- La méthode **`static Policy.getPolicy()`** renvoie la politique en cours
- La méthode **`refresh()`** de la classe **`Policy`** permet de relire le fichier de police de sécurité (pour l'attribution des domaines des nouvelles classes chargées ; pas pour les classes déjà chargées)

Domaine de sécurité

- Une classe appartient à un et un seul domaine déterminé par
  - la politique de sécurité au moment de son chargement
  - et par son « *code source* », c'est-à-dire l'origine du code
- Un domaine de protection est composé d'un *codesource*, d'une collection de permissions, d'un chargeur de classes et d'un tableau de « *principals* » (servent à identifier un utilisateur)
- Représenté par la classe **`ProtectionDomain`**

Notion de CodeSource

- En Java 2, l'origine du code d'une classe est constituée
  - de l'URL d'où a été chargée la classe
  - du signataire de la classe, avec les certificats qui ont été utilisés pour vérifier la signature
- Représenté par la classe **`CodeSource`**

Domaine système

- Le domaine « système » est formé des classes chargées par la chargeur de classe primordial (celui qui charge les classes au démarrage de Java)
- Les classes du domaine système ne sont pas contrôlées
- Actuellement, toutes les classes du JDK sont placées dans le domaine système
- Les applets ou les applications appartiennent à d'autres domaines déterminés par leur « *codeSource* »

Gestionnaire de sécurité

- Il contrôle l'accès aux ressources protégées
- Représenté par la classe **`java.lang.SecurityManager`** ou par une classe descendante
- En fait, le **`SecurityManager`** délègue le travail au contrôleur d'accès (nouvelle classe **`java.security.AccessController`** de Java 2)

Choix du gestionnaire de sécurité

- Ordinairement, il est installé par le contexte d'exécution (par exemple, par le navigateur Internet ou par une option de la commande java)
- Par défaut, les application locales, c'est-à-dire celles qui sont lancées par une classe principale venant du *classpath*, ne sont pas contrôlées par un gestionnaire de sécurité
- Aucun contrôle d'accès n'est donc effectué pour les applications locales, même pour les classes distantes chargées par l'application, par exemple, en utilisant un URLClassLoader

Choix du gestionnaire de sécurité sur la ligne de commande

- On peut installer un gestionnaire de sécurité au lancement d'une application :
  - gestionnaire par défaut (bac à sable, instance de java.lang.SecurityManager) :

```
java -Djava.security.manager … Classe
```

  - gestionnaire particulier :

```
java -Djava.security.manager=MonSecurityManager
… Classe
```


Choix du gestionnaire de sécurité dans un programme

- Installer un gestionnaire de sécurité (pour RMI sur cet exemple) :

```
System.setSecurityManager(
                new RMISecurityManager())
```

- Cet appel nécessite le droit `java.lang.RuntimePermission("setSecurityManager")`
- Récupérer le gestionnaire de sécurité :

```
System.getSecurityManager()
```


Déboguer les problèmes liés aux permissions

- Pour les applications, la propriété **`java.security.debug`** offre plusieurs options pour faire afficher les problèmes liées aux permissions ; ces options s'affichent par : **`java –Djava.security.debug=help`**
- Pour les applets, il faut faire afficher la console Java du navigateur ou du plugin Java de Sun

Contrôleur d'accès

- C'est lui qui décide si un accès à une ressource est autorisé
- Il tient compte
  - du contexte d'appel de la méthode qui veut accéder à la ressource système
  - du domaine de sécurité de *toutes les classes* qui sont dans le contexte d'appel
- Il est représenté par la classe **`AccessController`** qui ne contient que des méthodes **`static`** (**`checkPermission`** en particulier)

Contexte d'appel

- Il est utilisé pour vérifier les droits d'une méthode
- Il est formé des classes dont les méthodes sont dans les piles d'exécution des différents *threads* qui ont conduit à l'appel de la méthode
- Pour qu'un méthode ait un droit, il faut que tous les domaines des classes du contexte d'appel aient ce droit
- Représenté par la classe **`AccessControlContext`**

Extrait de **`FileOutputStream`**

```
public FileOutputStream(String name,
                        boolean append)
    throws FileNotFoundException {
  SecurityManager
    security = System.getSecurityManager();
  if (security != null) {
    security.checkWrite(name);
  }
  fd = new FileDescriptor();
  if (append) { openAppend(name); }
  else { open(name); }
}
```


Extrait de **`SecurityManager`**

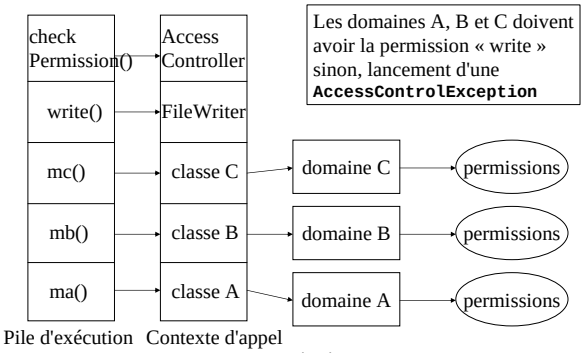
```
public void checkWrite(String file) {
  checkPermission(
     new FilePermission(file, "write"));
}

public void checkPermission(Permission perm) {
  AccessController.checkPermission(perm);
}
```

Appel à la classe **`AccessController`**

`AccessControlException`

- Si un appel est rejeté par le contrôleur d'accès, une **`AccessControlException`** (paquetage **`java.security`**) est levée
- Cette classe d'exception hérite de la classe **`java.lang.SecurityException`** qui elle-même hérite de **`RuntimeException`**
- Il n'est donc pas obligatoire de déclarer cette exception dans la définition des méthodes

Algorithme pour le contrôle d'accès



Code privilégié

- La méthode de la classe **`AccessController`**
  `static <T> doPrivileged(PrivilegedAction<T> action)`
  permet de donner des droits particuliers à du code
- Tout code exécuté par un appel à **`doPrivileged`** est considéré comme du code privilégié

Interface **`PrivilegedAction<T>`**

- Cette interface ne comprend qu'une méthode « **`T run()`** »
- La méthode **`doPrivileged`** exécute cette méthode **`run()`** et renvoie l'objet renvoyé par **`run()`**
- Tout le code contenu dans la méthode **`run()`** est donc considéré comme privilégié par le contrôleur d'accès

Autorisations du code privilégié

- Les autorisations d'accès du code privilégié sont déterminées uniquement par les droits
  - du domaine de la classe qui a appelé **`doPrivileged`**
  - des méthodes appelées par le code privilégié
- Il y a donc moins de vérifications à faire : les limitations liées aux méthodes placées en dessous de la pile d'exécution ne sont plus prises en compte
- Soyez donc très prudent avec ce que vous permettez dans du code privilégié !!

Exceptions et code privilégié

- La méthode **`run()`** ne doit pas renvoyer d'exception
- Sinon, on doit utiliser l'interface **`PrivilegedExceptionAction`** dont la méthode **`run()`** renvoie une **`PrivilegedException`** ; on peut récupérer l'exception lancée par **`run`** en appelant la méthode **`getException()`** de **`PrivilegedException`**

Restriction des droits accordés à du code privilégié

- On peut passer un 2ème paramètre de type **`AccessControlContext`** à la méthode **`doPrivileged`**
- La classe **`AccessControlContext`** contient une méthode **`checkPermission`** qui peut être redéfinie pour restreindre les droits accordés

Permissions pour exécuter du code privilégié

- Aucune permission spéciale n'est requise pour exécuter du code privilégié
- Ça n'est pas nécessaire car le code privilégié ne permet pas d'exécuter du code qui n'est pas autorisé normalement par la classe

Exemple de code privilégié

- Code extrait du constructeur de la classe **`java.io.PrintWriter`** :

```
lineSeparator =
  (String)AccessController.doPrivileged(
    new sun.security.action.GetPropertyAction(
      "line.separator"));
```

- La variable **`lineSeparator`** est utilisée par le code de la méthode **`println()`** ; **`doPrivileged`** assure que la propriété **`line.separator`** pourra être lue quelle que soit la politique de sécurité et les méthodes qui ont appelé ce code (puisque **`PrintWriter`** a tous les droits, étant dans le JDK)

Autre exemple de code privilégié : le problème à résoudre

- On a un fichier sensible **`password`** dans lequel il faut absolument respecter un format spécial
- Comment permettre l'écriture dans ce fichier sans permettre l'écriture de lignes qui ne respectent pas ce format ?

Une solution

- Écrire une classe spéciale, seule autorisée à écrire dans le fichier (utiliser son codebase dans le fichier de police), comportant une méthode **`ecrire`** qui effectue les écritures dans le fichier **`password`** en mode privilégié
- Les autres classes devront déléguer à cette classe les écritures dans le fichier
- Elles pourront ainsi enregistrer dans le fichier, mais sans risquer de casser le format du fichier
- Sans le mode privilégié, elles n'auraient pas eu le droit d'écrire dans le fichier

Exemple de code privilégié

```
static void ecrire(final String s) throws IOException {
  if (s n'a pas le bon format)
    throw new IOException("Mauvais format");
  try {
    AccessController.doPrivileged(
       new PrivilegedExceptionAction() {
          . . .
         public Object run() throws IOException {
            Ecriture de s dans le fichier password
         }
       });
  } catch(PrivilegedException e) {
    throw e.getException();
  }
}
```

Renvoie l'exception levée par run()

Threads et héritage du contexte pour le contrôle d'accès

- Quand un nouveau *thread* est créé, le contexte pour le contrôle d'accès est hérité par ce *thread*
- L'héritage se fait au moment de la création et pas du lancement du *thread*

Les fichiers de police de sécurité

Configuration de la sécurité

- Les grandes lignes pour la sécurité sont configurées par le fichier de propriétés `<java.home>/lib/security/java.security` où *<java.home>* est le répertoire où est installé le programme *jre (Java Runtime Environment),* en général le sous-répertoire **`jre`** du répertoire où a été installé le SDK

Fichiers de police (.policy)

- Ces fichiers sont utilisés pour définir les droits des classes Java
- Leur emplacement est donné par les propriétés **`policy.url.`***n* (*n*=1, 2,…) du fichier de propriétés **`java.security`**
- La convention est de donner l'extension **`.policy`** aux fichiers de police

Emplacement par défaut des fichiers de police (.policy)

- Emplacement de fichiers des polices « système » et « utilisateur » dans le fichier **`java.security`** :

```
policy.url.1=
  file:${java.home}/lib/security/java.policy
policy.url.2=
  file:${user.home}/.java.policy
```

- S'il n'existe pas de fichiers de police, la politique de sécurité correspond aux restrictions du bac à sable des applets (presque rien n'est autorisé)

`${java.home}` = valeur de la propriété java.home

Ajouter un fichier de police

Remplace les fichiers de police par défaut si « **==** »

Nom d'un fichier

- 2 moyens pour ajouter un fichier :
  - option **`-Djava.security.policy=`*`maPolice`*** des commandes *java* ou *appletviewer*
  - ajouter une propriété **`policy.url.`***n* dans un fichier de propriété ; par exemple, **`policy.url.3=file:${user.home}/mapolice`**
- La 1ère solution est la meilleure pendant les tests
- Remarque : ne pas oublier l'option **`-Djava.security.manager`**

Contenu des fichiers de police

- Un fichier de police de sécurité contient des entrées du type **`grant`** (**`signedBy`** et **`codeBase`** sont optionnels) :

```
grant signedBy "signataires", codeBase "url"
{
  permission1;
  permission2;
};
```

Ne pas oublier les « ; » **en particulier celui-ci !!**

- Il peut aussi contenir une entrée **`keystore :`**

```
keystore "emplacement", "type";
```

jks par défaut (format de *Sun*)

Exemple de fichier de police de sécurité

```
// Un commentaire
keystore ".keystore";
grant signedBy "toto" {
  permission java.io.FilePermission
  "${user.home}${/}-", "read";
};
grant codeBase "http://truc.fr/~toto/*" {
  permission java.util.PropertyPermission
  "user.home", "read";
};
```


Entrée **`grant`**

- Une entrée **`grant`** commence par **`grant`** suivi optionnellement d'un **`signedBy`** et/ou d'un **`codeBase`** (dans un ordre quelconque) :

```
grant
grant signedBy "signataire[,signataire2,…]"
grant codeBase "unURL"
grant signedBy "signataire", codeBase "unURL"
```


`signedBy` et **`codeBase`**

- **`signedBy`** indique par qui la classe doit être signée ; si plusieurs signataires sont mentionnés, la classe doit être signées par tous les signataires (un seul ne suffit pas)
- **`codeBase`** indique de quel URL la classe a été chargée par le chargeur de classes
- Par défaut, les permissions du **`grant`** sont accordées à toutes les classes, signées ou non, et venant de n'importe quel URL

URL pour le codeBase

- Différentes significations selon la fin de l'URL :
  - "**/**" désigne toutes les classes (pas les JAR) situées dans le répertoire placé avant le "/"
  - "**/*-**" désigne toutes les classes (y compris les classes dans les fichiers JAR) situées dans le répertoire
  - "**/-**" désigne toutes les classes (y compris les classes dans les fichiers JAR) situées dans l'arborescence du répertoire
- Si le codeBase désigne un répertoire, les classes se trouvent dessous **avec les chemins correspondant à leur paquetage**

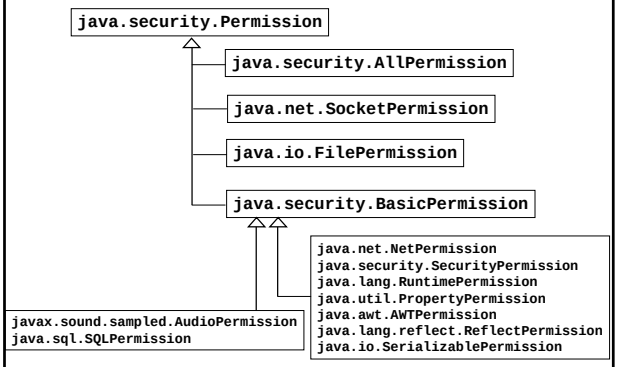
Format des URL

- Le séparateur est « **/** » dans tous les systèmes (même sous Windows ou MacOS) : la notion d'URL est « universelle » et ne doit pas dépendre d'un système particulier
- L'URL d'un fichier local commence par « **file:** »
- Exemples d'URL sous Windows : **/C:/rep1/**

Entrée **permission**

- Le format standard est **permission *classePermission* "*but*", "*action*";**
  - ***classePermission*** indique le type de permission que l'on donne (nom d'une classe de permission)
  - ***but*** indique sur quel objet on donne la permission
  - ***action*** indique quel type d'action on autorise sur le ***but***
- Certaines permissions peuvent ne pas avoir d'action, ni même de but
- Une permission peut comporter une clause **signedBy "*noms…*"** (***classePermission*** doit être signée)

Quelques classes de permissions



Exemples de permissions

- Les classes placées à la racine du serveur Web de **machine** ont la permission de lire les fichiers de **/tmp** et le droit de lire les valeurs de toutes les propriétés comme **user.home** ou **user.name**

```
grant codeBase "http://machine/*" {
  permission java.io.FilePermission "/tmp/*",
"read";
  permission java.util.PropertyPermission
"user.*", "read";
};
```


Exemples de permissions (2)

- Toutes les classes peuvent se connecter partout par socket et peuvent lire tous les fichiers du répertoire indiqué

```
grant {
  permission java.io.SocketPermission "*",
"connect";
  permission java.io.FilePermission
"C:\\users\\truc\\*", "read";
};
```

Doubler les "\" sous Windows

Exemples de permissions (3)

- Permission de lire les fichiers du répertoire **`HOME`** de l'utilisateur
- Cet exemple utilise la valeur de la propriété système **`user.home`** (**`${user.home}`**), et **`${/}`** qui désigne le séparateur utilisé dans les noms de fichiers (qui dépend du système d'exploitation)

```
grant {
  permission java.io.FilePermission
    "${user.home}", "read";
  permission java.io.FilePermission
    "${user.home}${/}*", "read";
};
```


Exemples de permissions (4)

- Certaines permissions n'ont pas d'*action* :

```
permission java.lang.RuntimePermission
  "getClassLoader";
```

- Ou même pas de but comme la permission **`AllPermission`**

Exemples de permissions (5)

- Si on veut tester du code indépendamment des problèmes d'autorisation, on peut donner momentanément toutes les permissions

```
grant {
  permission java.security.AllPermission;
};
```

- Très dangereux ! Ne jamais utiliser un tel fichier quand l'ordinateur est connecté à Internet
- À n'utiliser qu'avec l'option **`-D`** de java ; sinon, limiter la portée par un codebase ; par exemple :

```
grant codeBase "file:/-" {
 permission java.security.AllPermission;
};
```


Exemples de permissions (6)

- Si on veut donner l'autorisation d'accéder aux membres non public d'une classe avec la réflexivité, on peut donner la permission suivante :

```
permission
    java.lang.reflect.ReflectPermission
    "suppressAccessChecks";
```

- C'est évidemment très dangereux et doit être réservé à des classes très particulières comme les débuggeurs ou les outils interactifs pour construire des applications

`BasicPermission`

- Les derniers exemples de permissions, comme les **`PropertyPermission`** sont des classes filles de **`BasicPermission`**
- Elles permettent de donner un nom de but hiérarchique de type **`user.home`**, qui peut comporter un « * » à la place d'un des noms
- Par exemple,
  `permission java.util.PropertyPermission "user.*"`
  donne des permissions sur les propriétés **`user.home`**, **`user.dir`**, **`user.name`**,…

`FilePermission`

- Le but peut être de plusieurs types :
  - **`fichier`** ou **`répertoire`**
  - **`repertoire/*`** (ou **`*`**) : fichiers ou répertoires situés juste sous **`repertoire`**
  - **`repertoire/-`** (ou **`-`**) : fichiers ou répertoires situés dans l'arborescence de **`repertoire`**
  - **`<<ALL FILES>>`**
- Attention, **`repertoire/*`** et **`repertoire/-`** ne donnent pas la permission sur le répertoire lui-même !
- Pour Windows, remplacer « **`/`** » par « **`\\`** » ou par « ${/} »

`FilePermission`

- L'action peut être **`read`**, **`write`**, **`delete`**, **`execute`**, mais pas **`"read,write"`** (2 entrées sont nécessaires)

Exemples de `FilePermission`

```
grant {
  permission java.net.FilePermission
  "/tmp/*" "read";
};
```

```
grant codeBase "http://deptinfo.unice.fr/-",
    signedBy "paul" {
  permission java.net.FilePermission
    "C:\\users\\bibi\\*", "write";
};
```


Permissions pour le Web

- Dès qu'une connexion avec une machine distante a lieu, il est vraisemblable que des sockets sont utilisés et on devra ajouter des permissions **`java.net.SocketPermission`**
- Par exemple, pour charger un fichier depuis la machine deptinfo.unice.fr, en passant par un serveur HTTP, on devra ajouter la permission suivante :

```
permission "deptinfo.unice.fr",
"connect,accept,resolve";
```


Messages d'erreur liés à la sécurité

```
Exception occurred during event dispatching:
java.security.AccessControlException: access denied
  (java.io.FilePermission C:\ read)
  at
  java.security.AccessControlContext.checkPermission(...)
  at java.security.AccessController.checkPermission(...)
  at java.lang.SecurityManager.checkPermission(...)
  at java.lang.SecurityManager.checkRead(...)
  at java.io.File.isDirectory(...)
```

Il manque cette permission
dans les fichiers de police de sécurité

Trouver les permissions qui manquent

- Pour faire afficher les noms des permissions nécessaires à l'exécution, on peut lancer une application avec

```
java –Djava.security.debug=access,failure
```


Faire afficher les permissions accordées à une classe

```
ProtectionDomain domain =
  this.getClass().getProtectionDomain();
PermissionCollection pcoll =
  Policy.getPolicy().getPermissions(domain);
Enumeration enum = pcoll.elements();
while (enum.hasMoreElements()) {
  Permission p =
    (Permission)enum.nextElement();
}
```

Repérer la bonne JVM

- Si malgré tous vos efforts, l'applet ou l'application refuse de tenir compte de vos fichiers de permissions, vous vous trompez peut-être de JVM
- Il est en effet possible d'avoir plusieurs JVM sur une machine, en particulier quand on utilise un navigateur Web qui a sa propre JVM, ou qui utilise le plugin Java de Sun, avec sa propre JVM

Entrée keystore

- `keystore "urlFichier" [, "type"];`
- Cette entrée est obligatoire dès qu'une entrée **`grant`** fait référence à une signature
- Elle doit alors être unique
- Elle peut être n'importe où dans le fichier de police
- **`emplacement`** est un URL absolu, ou relatif à l'emplacement du fichier de police de sécurité
- type définit le format de stockage et de cryptage des informations contenues dans le fichier
  - "jks" est un type défini par *Sun* ; c'est le type par défaut

Cryptographie

Plan

- Concepts
- Signature digitale
- Certificat

Concepts pour la cryptographie

Échanger des messages confidentiels

- On veut
  - confidentialité : seul le destinataire peut lire le message
  - authentification de l'expéditeur
  - intégrité du contenu du message : un tiers ne peut modifier le message sans que ça se voit
  - non-répudiation : l'expéditeur ne peut nier avoir envoyer le message et le destinataire ne peut nier l'avoir reçu

Cryptographie

- Science et techniques pour chiffrer des informations
- Les algorithmes reposent sur la notion de clé
- Une clé est une information utilisée pour chiffrer ou déchiffrer une information
- 2 types principaux d'algorithmes de chiffrement/déchiffrement :
  - à clé cachée (ou symétrique)
  - à clé publique (ou asymétrique)

- Dans ce cours nous n'étudierons pas les algorithmes de chiffrement, ni le chiffrement des messages
- Nous verrons comment
  - signer des documents
  - fournir des certificats pour assurer
    - l'authentification de l'auteur d'un message
    - l'intégrité du message
    - la non-répudiation de l'auteur du message

Cryptographie à clé cachée

- C'est la plus ancienne technique de cryptographie
- Une même clé permet de chiffrer et de déchiffrer les messages
- Cette clé doit être partagée par l'expéditeur et le destinataire

Problèmes de la cryptographie à clé cachée

- Comment s'échanger la clé ?
- Si un tiers peut intercepter la clé pendant l'échange, il peut lire tous les messages

Comparaison clé cachée - clé publique

- La cryptographie à clé publique n'a pas le problème de la transmission de la clé
- Les possibilités sont plus riches avec la cryptographie à clé publique
- Mais le chiffrement/déchiffrement est souvent plus rapide avec une clé cachée

Cryptographie à clé publique

- Chaque acteur (expéditeur ou destinataire) a 2 clés différentes :
  - une clé publique connue de tous
  - une clé privée connue seulement par l'acteur
- Un message chiffré avec une des 2 clés est décryptée avec l'autre clé
- Permet d'éviter le problème d'échange de clé symétrique et offre de nouvelles possibilités

Principe essentiel de la cryptographie à clé publique

- Les 2 clés sont générées par des algorithmes qui s'appuient sur des théories mathématiques (arithmétique des grands nombres premiers) qui leur assurent la propriété suivante :
  - il est facile de chiffrer un message avec la clé publique
  - il est extrêmement difficile de déchiffrer le message si on ne connaît pas la clé privée

Infrastructure à clé publique

- L'utilisation des clés publiques-privées impose la gestion et mise à disposition des clés publiques, la certification des identités associées à ces clés
- Des tiers effectuent ces tâches : autorités d'enregistrement et de certification
- Pour travailler à grande échelle avec les clés publiques il est donc nécessaire d'installer une infrastructure à clé publique (ICP en abrégé, PKI en anglais)

Utilisation de la cryptographie à clé publique

- Nous allons étudier comment elle permet
  - la confidentialité
  - l'authentification
  - la non-répudiation
  - l'intégrité

Chiffrement avec clé publique

- Envoi d'un message chiffré :

1. l'expéditeur chiffre son message avec la clé publique du destinataire (elle est connue de tous)
2. le destinataire déchiffre avec sa clé privée, restée chez lui bien en sécurité

- On assure ainsi la confidentialité et l'authentification du destinataire
- Pour l'intégrité, l'authentification et la non-répudiation de l'expéditeur, il faut ajouter une signature

Signatures digitales

Signature

- Une signature numérique à clé publique permet :
  - authentification de l'expéditeur
  - intégrité : vérifier que ces données n'ont pas été modifiées depuis que le document a été signé
  - non répudiation : le signataire ne peut pas nier avoir signé le document

Cryptage

- Un message peut être crypté avec une clé publique ; ce message pourra être décrypté par la clé privée correspondante
- Ce sens sert à transmettre des messages secrets
- Un message peut être crypté avec une clé privée ; ce message pourra être décrypté par la clé publique correspondante
- Ce sens sert pour les signatures digitales

Propriétés des signatures à clés asymétriques

- Elle permet d'affirmer que le signataire du message a bien une certaine clé publique
- Elle ne peut être imitée si on ne connaît pas la clé privée du signataire

Résumé

- Un résumé (*message digest* en anglais) est une information de taille fixe calculée à partir du contenu du message par une fonction de hachage
- La taille du résumé est bien plus petite que la taille du message

Fonctions de hachage

- Une fonction de hachage h : T $\rightarrow$ C associe à tout texte T un condensé (empreinte, résumé) C de longueur fixe de ce texte
- Une bonne fonction de hachage h a les propriétés suivantes :
  - il est facile de calculer C
  - il est très difficile (sinon impossible) de calculer T à partir de C
  - il est très difficile de trouver T' tel que h(T) = h(T')
  - h est « presque injective » ; si T1 $\neq$ T2, il est « presque sûr » que h(T1) $\neq$ h(T2)

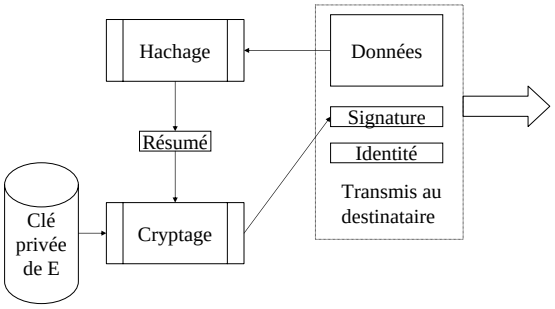
Fonctions de hachage

- Les fonctions les plus couramment utilisées :
  - MD5 (128 bits)
  - SHA
  - SHS (160 bits)

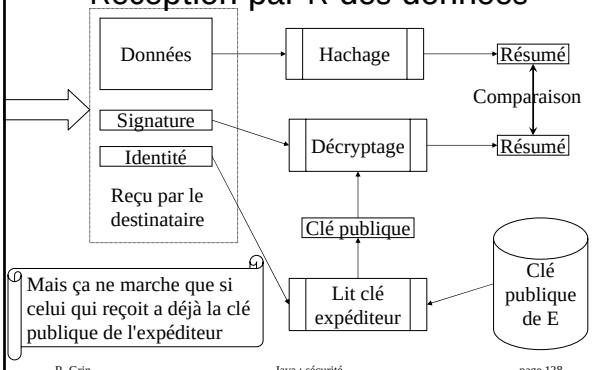
Comment signer

- Une signature est composée en chiffrant un résumé du message avec la clé privée du signataire
- Le destinataire va décrypter le résumé avec la clé publique du signataire
- Si le résultat du décryptage correspond bien au résumé du message reçu, ça signifie que
  - le signataire est bien le bon (celui qui a cette clé publique)
  - le message n'a pas été modifié depuis le calcul de la signature

Envoi par E de données signées



Réception par R des données



Certificats

Association clés-identités

- Celui qui reçoit un message signé doit posséder la clé publique du signataire et avoir un moyen d'associer cette clé publique à l'identité du signataire (base de donnée ou autre)
- Imaginez le travail pour
  - recevoir et gérer les milliers de clés publiques des clients et des partenaires
  - s'assurer que ces clés publiques appartiennent bien à ces correspondants

Certificats

- Comment ça se passe en réalité :
- Le plus souvent, celui qui reçoit le message n'a pas la clé de l'expéditeur
- La clé publique du signataire est transmise avec le message, dans un certificat qui associe cette clé avec l'identité du signataire

Utilisation des certificats

- Comment avoir confiance en ce certificat ?
- Ce certificat est signé par une autorité publique de certification (comme *Verisign*) qui a une clé publique bien connues de tous (ou il existe des moyens sûrs et faciles de l'obtenir)
- On se retrouve alors dans le cas où on connaît la clé de l'expéditeur des données (la donnée est le certificat) ; on peut donc s'assurer que le certificat contient des données exactes
- On a donc la clé publique de l'expéditeur, et on peut donc vérifier la signature des données

Ce que contient un certificat

- L'autorité fournit un certificat signé par elle, qui contient
  - des informations sur le possesseur du certificat : nom, adresse, etc.
  - la clé publique de ce possesseur
  - des informations liées au certificat : dates de validation, numéro de série, etc.

Ce qui est envoyé

Un certificat est un message signé. Si on connaît la clé publique de l'autorité, et si on a confiance en cette autorité, on peut être sûr des informations qu'il contient.

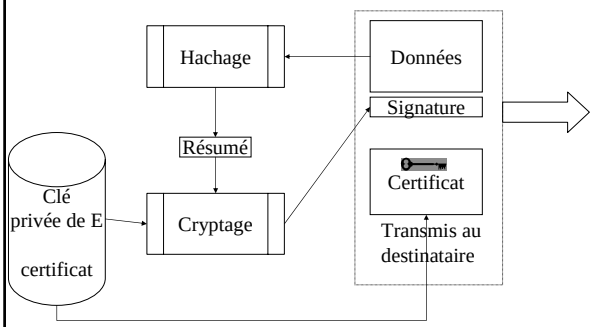
Stockage des clés

- Toute application Java peut avoir un lieu de stockage des clés publiques
- Cette base de données contient des clés publiques et des chaînes de certificats qui authentifient les clés publiques
- Elle peut aussi contenir des clés privées locales (celles de l'utilisateur de l'application) avec leurs chaînes de certificats
- Des mots de passe peuvent être associés au lieu de stockage et à chacune des clés

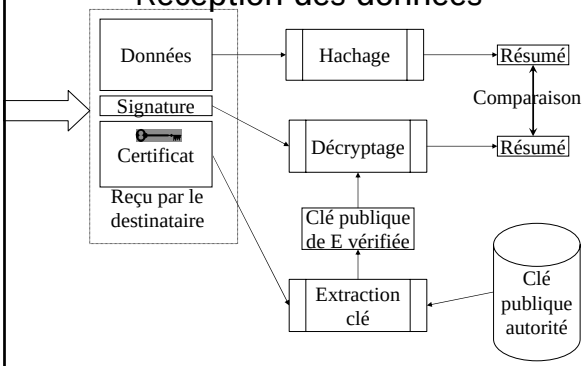
Entrepôt des certificats par défaut

- Par défaut les certificats sont entreposés dans le fichier *java.home*/lib/security/cacerts (java.home est le répertoire jre d'installation de java)
- Pour lister les certificats : keytool -list
- On peut indiquer un autre fichier entrepôt : keytool –list –file entrepot

Envoi de données signées



Réception des données



La chaîne de vérification

- Le vérificateur du certificat doit déjà connaître la clé publique de l'autorité de certification (appelons-la CA1)
- Sinon, le vérificateur doit avoir un certificat, émis par une autorité de certification CA2 dont il connaît la clé publique, et qui authentifie la clé publique de CA1
- On peut ainsi avoir une chaîne de certificats signés par CA1, CA2, CA3, ...
- Cette chaîne confirme l'identité du signataire si un des certificats a été signé par une autorité déjà connue par le vérificateur

Comment obtenir un certificat

1. Génération d'une paire de clés publique-privée
2. Fournir à l'autorité qui délivre le certificat
   - la clé publique
   - des documents qui certifient l'identité
3. L'autorité fournit un certificat signé par elle, qui permet d'associer la clé publique et l'identité

- Pour les cas de diffusion restreinte, on peut se contenter d'un certificat auto-signé que l'on distribue aux utilisateurs

Types de certificats

- Les autorités universellement connues ne sont pas les seules à délivrer des certificats
- Un organisme peut aussi utiliser en interne sa propre autorité de certification
- Un certificat peut aussi être auto-signé, c'est-à-dire signé par la personne qui est authentifiée par le certificat (convient si le destinataire connaît la clé publique de celui qui a signé)

Les outils

Plan

- Plugin Java pour les navigateurs Web
- Les outils pour la signature

Plugin Java pour les navigateurs Web

Qu'est-ce que c'est ?

- Sun fournit un plugin Java pour les navigateurs Web qui permet aux applets de s'exécuter indépendamment de la version de la machine virtuelle Java installée dans le navigateur
- Des informations sur le plugin en http://java.sun.com/products/plugin/
- FAQ : http://java.sun.com/j2se/1.5.0/docs/guide/plugin/developer_guide/faq/index.html

Installation du plugin

- Ce plugin s'installe sur la machine d'un client Web en même temps que le JRE Java
- S'il n'est pas installé, l'applet peut demander son installation
- L'installation est longue mais elle ne se fait qu'une fois ; les applets suivantes pourront utiliser le plugin installé par la première applet

Pratique de la signature avec les outils fournis par *Sun*

Outils pour la sécurité

- *policytool* facilite la saisie d'une police de sécurité (évite les fautes de syntaxe que l'on peut faire si on modifie « à la main » les fichiers de police)
- *jarsigner* permet de signer un fichier JAR ou de vérifier sa signature
- *keytool* permet de gérer des clés et des certificats

Algorithmes utilisés dans les API

- Fonction de hachage par défaut : SHA1
- Chiffrement par défaut pour les signatures : DSA, ce qui donne l'algorithme de signature de nom interne **`SHA1withDSA`**
- Autres possibilités : **`MD2withRSA`**, **`MD5withRSA`**, **`SHA1withRSA`**
- Certificats selon la norme **`X.509`**

- Nous allons donner 2 exemples concrets :
  - signer du code qui sera exécuté par un autre (une applet)
  - signer un document que l'on envoie à un autre

Créer les clés

- Création des clés privée et publique de toto, et d'un certificat auto signé valable 90 jours (rangés dans le fichier **.keystore**) :

```
> keytool -genkey -alias toto -keystore .keystore
Enter keystore password: abc123
What is your first and last name?
  [Unknown]: Pierre Toto
What is the name of your organizational unit?
  [Unknown]: Dept informatique
What is the name of your organization?
  [Unknown]: UNSA
What is the name of your City or Locality?
  [Unknown]: Nice
. . .
```

créé s'il n'existe pas ; par défaut `$HOME/.keystore`

Obtenir un certificat

- Obtenir un certificat (ou une chaîne de certificats) pour certifier la clé publique de *toto*, émis par une autorité publique connue
- Pour cela il faut envoyer la clé publique de *toto* et les documents demandés par l'autorité pour identifier *toto*
- En retour l'autorité publique envoie un certificat qui certifie que la clé publique envoyée est bien celle de *toto*

Obtenir un certificat

- Lorsque l'on a obtenu le certificat, on l'ajoute au fichier où sont entreposées les clés (ici le fichier **.keystore**) de *toto* :

```
> keytool -import -alias toto
    -keystore .keystore -file reponseCA
```

- Cette étape n'est pas obligatoire, on peut se contenter du certificat auto-signé si la communauté qui échange des informations n'est pas trop importante

Exporter un certificat

- Exporter le certificat :

```
> keytool -export -alias toto
    –keystore .keystore -file toto.crt
```

- Ce certificat pourra être importé dans la base de données des clés de ceux qui feront confiance à *toto*

Exporter un certificat

- Exporter le certificat :

```
> keytool -export -alias toto
    –keystore .keystore -file toto.crt
```

- Ce certificat pourra être importé dans la base de données des clés de celui qui fera confiance à *toto*

Signer un fichier jar

- Créer le fichier **unjar.jar**

```
> jar cvf unjar.jar Classe.class
```

- *toto* signe le fichier **unjarsigned.jar** qui ajoute son certificat au contenu du fichier **unjar.jar**

```
> jarsigner -keystore .keystore
    -signedjar unjarsigne.jar unjar.jar toto
```

Exemple de fichier MANIFEST.MF d'un fichier jar signé

```
Manifest-Version: 1.0
Created-By: 1.3.0 (Sun Microsystems Inc.)

Name: TestAppletEcriture.class
SHA1-Digest: mOaQUgZhHj1ksr53hHk/FAGLdWE=

Name: rep/Classe.class
SHA1-Digest: xrQEm9gZhHj1ksr53hHkYV4XIt0=
```

résumé du fichier `rep/Classe.class`

Etapes à la réception des messages

- L'utilisateur qui recevra de toto le fichier signé va vérifier que c'est bien toto qui a signé
  1. il vérifie d'abord que le certificat de toto qu'on lui a envoyé contient des informations qui correspondent aux informations qu'il a reçu par d'autres voies fiables
  2. il doit tout d'abord importer le certificat de toto dans sa base de clés
  2. il peut ensuite utiliser cette base de clé en donnant son emplacement dans le fichier de police de sécurité (entrée "keystore")

Importer un certificat

- Vérifier le certificat en comparant les informations imprimées par keytool avec des informations fiables obtenues par ailleurs :
  `keytool -printcert -file toto.cer`
- Importer le certificat de toto utilisateur en qui on peut avoir confiance :

```
keytool -import -alias toto
  -file toto.cer -keystore .keystore
```


Certificats révoqués

- Java ne vérifie pas auprès de l'autorité de certification si le certificat a été révoqué (en cas, par exemple, de vol ou de certificat attribué par erreur)
- Pour les cas où une sécurité renforcée est nécessaire, il faudra ajouter du code pour effectuer cette vérification

Signature et plugin Java

- Depuis la version JDK 1.3, une applet signée et certifiée peut avoir tous les droits si l'utilisateur déclare avoir confiance dans cette applet signée (le navigateur lui pose la question)
- Cette fonctionnalité a été ajoutée pour faciliter l'exécution des applets sans installation de fichiers de police de sécurité spéciaux chez les clients Web

Signature et plugin Java

- Mais cette possibilité peut être jugée dangereuse
- Le fichier de police de sécurité du client peut comporter une ligne

```
permission java.lang.RuntimePermission
"usePolicy";
```

pour indiquer que seule la police de sécurité doit être examinée, et que même une applet signée devra s'y tenir

Signature et plugin Java

- Le plugin Java a changé de politique de sécurité à chaque version du jre, ce qui ne favorise pas la portabilité
- Les fonctionnalités ont été ajoutées pour faciliter le déploiement des applets
  - avec le jre 1.2, seuls les fichiers de police de sécurité sont pris en compte
  - avec le jre 1.3, il n'est pas tenu compte de ces fichiers de police si l'applet est signée avec un certificat obtenu par une autorité connue de certification, et si l'utilisateur affirme avoir confiance en ce certificat
  - avec le jre 1.4, ça marche aussi avec les certificats auto-signés

Compléments

Utiliser MD5

- Exemple de code qui renvoie le MD5 d'un tableau d'octets (il faut importer **`java.security`**) :

```
try {
  MessageDigest md =
    MessageDigest.getInstance("MD5");
  return md.digest();
}
catch(NoSuchAlgorithmException e) {
  . . .
}
```


Utiliser MD5

- Une petite difficulté : les mots de passe sont récupérés dans un **`char[]`** (par exemple avec **`getPassword()`** de **`JPasswordField`**) et **`digest`** prend un **`byte[]`** en paramètre
- Pour transformer un **`char[]`** en **`byte[]`** on peut utiliser **`Charset`** :

```
Charset charset =
  Charset.forName("ISO-8859-1");
CharsetEncoder encoder =
  charset.newEncoder();
ByteBuffer bbuf =
  encoder.encode(CharBuffer.wrap(mdp));
byte[] mdpByte = bbuf.array();
```


Protéger les paquetages

- Par défaut, les paquetages ne sont pas protégés
- Un programmeur peut ajouter une classe dans n'importe quel paquetage et avoir ainsi accès à tous les membres des classes du paquetage qui ont un accès réservé au paquetage
- On peut empêcher l'ajout de nouvelles classes dans les paquetages par divers moyens :
  - par les mécanismes de protection liés aux fichiers de polices de sécurité
  - par les fichiers jar "scellés"

Protéger des paquetages avec les fichiers de sécurité

- On peut empêcher
  - le chargement direct d'une classe d'un paquetage
  - l'ajout de nouvelles classes dans un paquetage
- Pour commencer, il faut ajouter des entrées dans le fichier de sécurité (.security)
- Ces entrées sont des listes de noms séparés par des « , » ; tout paquetage dont le nom *commence* ainsi est protégé
- Pour empêcher le chargement direct :

```
package.access=fr.unice.librairie,com.truc
```

- Pour empêcher l'ajout de nouvelles classes :

```
package.definition= fr.unice,com.truc
```

Permissions pour les paquetages

- Si on a protéger un paquetage on peut autoriser l'action interdite en ajoutant des entrées dans les fichiers de police de sécurité (**`.policy`**)
- L'autorisation suivante permettra un accès direct aux classes du paquetage **`fr.unice.toto.truc`**

```
RuntimePermission
"accessClassInPackage.fr.unice.toto.truc"
```

- De même, il faudra ajouter une autorisation « **`RuntimePermission "defineClassInPackage.."`** » pour permettre l'ajout d'une classe dans un paquetage

Exemple

```
grant codeBase "http:mezzo.unice.fr/-" {
    java.lang.RuntimePermission
   "accessClassInPackage.fr.unice.librairie";
};
```


Prise en compte des protections des paquetages

- La protection des paquetage n'est pas vraiment prise en compte par les chargeurs de classes de la version 1.3
- La protection d'accès n'est prise en compte que par la classe **`URLClassLoader`**, et encore, seulement si l'instance du chargeur de classes a été obtenue par la méthode **`static newInstance()`** et pas par **`new`**
- La protection pour l'ajout d'une classe dans un paquetage n'est prise en compte par aucun chargeur de classes

Fichiers jar scellés

- L'entrée "**`Sealed: true`**" dans la section principale du fichier MANIFEST d'un fichier jar empêche l'ajout de classes des paquetages de ce fichier, venant d'une autre source que ce fichier jar
- On peut aussi ne protéger que certains paquetages du fichier jar en donnant des entrées du type

```
Name: fr/unice/p1
Sealed: true
```


Protéger des objets

- On peut protéger un objet particulier
- **`GuardedObject`** permet de faire garder un objet par un autre objet qui est d'une classe qui implémente l'interface **`Guard`** :

```
GuardedObject go = new GuardedObject(
  new ClasseProtegee(…),
  new GardeObjet(…));
```

La classe implémente l'interface **`Guard`**

Interface **`Guard`**

- Elle contient la seule méthode **`checkGuard()`** qui doit lancer une **`SecurityException`** si l'accès à l'objet est interdit
- Cette interface est implémentée par les classes filles de la classe **`java.Security.Permission`**
- Dans ce cas, **`checkGuard()`** fait un appel à **`securityManager.checkPermission(this);`**

Exemple

- Si on garde un objet avec une instance de **`FilePermission`**, l'accès à l'objet sera autorisé dans les mêmes conditions que l'accès à un certain fichier
- Avec le code suivant, **`objet`** ne pourra être récupéré que par le code qui aura l'autorisation de lire le fichier **`/rep1/fichier`** :

```
GuardedObject go =
  new GuardedObject(
    objet,
    new FilePermission("/rep1/fichier",
                       "read"));
```


Récupération de l'objet

- Quand on veut récupérer l'objet protégé, on appelle la méthode **`getObject()`** de la classe **`GuardedObject`**
- **`getObject()`** fait un appel automatique à la méthode **`checkGuard()`** qui lancera une **`SecurityException`** pour signaler que l'accès à l'objet est interdit

JAAS (*Java Authentification and Authorization Service*)

Présentation

- Le mécanisme des autorisations de Java 2 est centré sur le code : les permissions sont données ou non selon l'origine du code (d'où vient-il, qui l'a signé)
- JAAS est centré sur l'utilisateur : il va permettre de donner des permissions en se basant sur l'utilisateur qui exécute le code
- Les utilisateurs doivent s'authentifier et certaines parties du code ne pourront être exécutées que par les utilisateurs autorisés

Concepts de base de JAAS

- Sujet (*subject*) : représente une entité authentifiée ; utilisateur, administrateur, service Web, processus,…
- Identité (*principal*) : une des identités d'un sujet ; un sujet peut en avoir plusieurs, sur le modèle d'un utilisateur qui peut avoir plusieurs noms pour des services différents, un numéro de sécurité sociale, une adresse mail, etc.
- Pièce d'identité (*credential*) : justificatif pour une identité ; peut être un objet quelconque

Paquetages et quelques classes

- Depuis la version 1.4, JAAS est inclus dans le J2SE
- **`javax.security.auth`** contient la classe **`Subject`**
- **`javax.security.auth.login`** contient la classe **`LoginContext`**
- **`javax.security.auth.spi`** contient l'interface **`LoginModule`**

Paquetages et quelques classes

- **`javax.security.auth.callback`** contient les interfaces **`Callback`** et **`CallbackHandler`**, et quelques classes qui implémentent **`Callback`**

2 parties dans JAAS

- L'authentification de l'utilisateur (souvent avec un système de mot de passe)
- Les autorisations, liées à l'utilisateur qui s'est authentifié, qui s'ajoutent au système des autorisations de Java 2 déjà étudié

Exemple minimal de code

```
LoginContext lc = new LoginContext("Appli1");
try {
  lc.login();
}
catch(LoginException e) {
  // Le login n'a pas marché
  . . .
}
// Le login a marché
Subject sujet = lc.getSubject();
Subject.doAs(sujet, new ActionProtegee());
```

utilise des CallbackHandler pour demander à l'utilisateur des informations

Authentification

- Elle est effectuée par une instance de la classe **`LoginContext`**
- Quand on crée cette instance, on lui passe un identificateur qui va permettre à JAAS de savoir comment se fera l'authentification de l'utilisateur
- Cet identificateur correspond à une entrée dans un fichier de configuration ; il indique le type d'authentification qui sera effectuée

Emplacement du fichier de configuration

- Il peut être donné par
  - la propriété Java **`java.security.auth.login.config`**
  - une ou plusieurs entrées dans le fichier **`java.security`** :

```
login.config.url.1=file:${java.home}/
lib/security/truc.config
```


Fichier de configuration

- Il contient des entrées qui indiquent quels modules de login devront être utilisés :

```
entree1 {
  ClasseModule1 flag options;
  ClasseModule2 flag options;
  . . .
}
entree2 {
  . . .
}
```

Exemple de fichier de configuration

Identifiant passé en paramètre du constructeur de LoginContext

```
Appli1 {
  fr.unice.SgbdLoginModule Required
    driver="org.gjt.mm.mysql.Driver"
    url="jdbc:mysql://m.unice.fr/jaas?user=adm"
    debug="true";
}
Appli2 {
  fr.unice.FichierLoginModule Required
    fichier="rep/motdepasse";
}
```


Classe `LoginContext`

- Classe du paquetage **`javax.security.auth.login`**
- Elle contient une méthode **`login()`** pour authentifier une entité liée à l'application
- Le nom du contexte est fourni au constructeur
- Ce nom correspond à une entrée du fichier de configuration
- Cette entrée indique les **`LoginModule`**s qui seront utilisés pour authentifier l'entité

Système de plugin

- Le **`LoginContext`** utilise des plugins, appelés modules de login, qui permettent d'authentifier l'utilisateur suivant différents modes ; par exemple, en consultant une base de données ou en vérifiant que l'utilisateur est bien enregistré dans un registre ou dans le système d'exploitation de l'ordinateur
- Les modules de login sont des classes qui implémentent l'interface **`javax.security.auth.spi.LoginModule`**

Modules de login

- Le paquetage **`com.sun.security.auth.module`** contient des modules JAAS pour authentifier avec :
  - nom et mot de passe Unix (**`UnixLoginModule`**) ou Windows (**`NTLoginModule`**)
  - un service JNDI (**`JndiLoginModule`**)
  - le protocole Kerberos (**`Kbr5LoginModule`**)
  - une clé enregistrée dans un fichier KeyStore (**`KeyStoreLoginModule`**)

Modules de login

- On peut aussi écrire ses propres modules de login, ou les acheter ; par exemple des modules pour authentifier avec
  - un nom et mot de passe enregistrés dans une base de données, ou un fichier

Interface `LoginModule`

- **`boolean login()`** : authentifie un **`Subject`** ; utilise le plus souvent des callbacks pour communiquer avec l'entité à authentifier si c'est nécessaire (par exemple pour obtenir un nom et un mot de passe) ; elle récupère auprès des callback les informations obtenues
- **`boolean commit()`** : appelée si l'authentification a bien eu lieu, cette méthode associe des identités (Principal) au Subject
- **`boolean abort()`** : appelée si l'authentification a échoué

Interface **`LoginModule`**

- **`void initialize(…)`** : appelée juste après la création du LoginModule pour lui passer le Subject, le CallbackHandler.
  Un des paramètres est une *map* qui contient les options associées au module (voir le format du fichier de configuration).
  Par exemple, si le module authentifie les utilisateurs grâce à une table d'une base de données relationnelle, ces valeurs décriront la base de données et la table (driver, URL,…)
- **`boolean logout()`** : sort du Subject en cours

Classe **`javax.security.auth.Subject`**

- Contient les méthodes **`static doAs`**, **`doAsPriviledged`** et **`getSubject`** (renvoie le **`Subject`** associé au contexte actuel)
- et des accesseurs pour les **`Principal`** et les **`Credential`** qui sont associés à ce **`Subject`**

Interface **`java.security.Principal`**

- Elle ne contient que la méthode
  `String getName()`
  qui renvoie le nom de l'identité

Interface **`javax.security.auth.callback.CallbackHandler`**

- Contient la seule méthode
  `void handle(Callback[] callbacks)`
- Cette méthode utilise les différents callbacks pour obtenir des informations pour authentifier l'utilisateur
- L'implémentation dépend du type d'application ; par exemple, si l'application utilise un GUI et qu'un callback a besoin d'un nom et d'un mot de passe, elle peut faire afficher une fenêtre de dialogue pour cela

Interface **`Callback`**

- Interface de **`javax.security.auth.callback`** qui ne contient aucune méthode
- Un callback permet à l'application d'obtenir les informations qui seront utilisées pour l'authentification et de passer ces informations au code qui va authentifier l'utilisateur
- Par exemple, un callback peut avoir besoin d'un nom et un mot de passe et conserver ces informations pour pouvoir les redonner ensuite au code de la méthode login() du **`LoginModule`** qui a appelé la méthode **`handle`** du **`CallbackHandler`**

Interface **`Callback`**

- Un callback est indépendant du type de l'application
- Il a seulement besoin de certaines informations pour pouvoir les redonner au code qui va authentifier l'utilisateur
- C'est le callbackHandler qui va s'adapter à l'application pour obtenir ces informations

Autorisations

- Une fois que l'utilisateur s'est authentifié, on peut récupérer une instance de **`Subject`** par **`loginContext.getSubject()`**
- Ensuite, on peut n'autoriser certaines actions qu'à des utilisateurs particuliers
- L'exécution de ces actions devra être lancée par la méthode **`static doAs`** de la classe **`Subject`** ; par exemple :

```
ActionSpeciale action =
    new ActionSpeciale();
Subject.doAs(subject, action);
```


Interface `PrivilegedAction`

- Les actions passées à la méthode **`doAs`** doivent appartenir à une classe qui implémente l'interface **`PrivilegedAction`**
- Cette interface contient la méthode **`run()`** qui contient l'action à exécuter
- La méthode **`doAs`** renvoie un **`Object`**, renvoyé par la méthode **`run()`**
- Comme avec la méthode **`doPrivileged`** étudiée avant, si une action peut lancer des exceptions, elle doit implémenter **`PrivilegedExceptionAction`**

Mode privilégié

- Une méthode **`doAsPrivileged`** permet de lancer l'action dans un autre contexte d'exécution (en mode privilégié si on lui passe **`null`** comme contexte)
- Attention à ne pas ouvrir des portes aux pirates !

Permissions indispensables

- Les classes qui créent des **`LoginContext`** ou appellent la méthode **`doAsPrivileged`** doivent avoir certaines autorisations :

```
permission
  javax.security.auth.AuthPermission
  "createLoginContext"
permission
  javax.security.auth.AuthPermission
  "doAsPrivileged"
```


Autorisations

- Il est possible d'attribuer certaines autorisations à des utilisateurs authentifiés par JAAS dans les fichiers de police de sécurité
- La syntaxe est :

```
grant codebase …, signed by …,
Principal classePrincipal utilisateur {
  permission . . .;
  permission . . .;
}
```


Exemple d'autorisation

```
grant codebase "http://. . . "
  Principal fr.truc.Principal1 "pierre" {
  permission java.util.PropertyPermission
  "user.home", "read";
  . . .
}
```

`CallbackHandler`

- Pour accroître la portabilité des applications, la méthode **`login`** échange des informations avec l'utilisateur à authentifier, par l'intermédiaire d'un callbackHandler
- Un callbackHandler est dépendant de l'application, au contraire des callbacks qu'il utilise

Exemples de `CallbackHandler`

- En changeant ainsi de handler, on pourra, par exemple, demander le nom et le mot de passe d'un utilisateur en mode texte dans une console, ou dans une fenêtre Swing, ou par tout autre moyen, sans changer le code de la méthode **`login`**

Utilité des callbacks

- Les échanges avec l'utilisateur sont déterminés par un tableau de callbacks qui va être passé au handler
- Un callback n'a pas d'action directe avec l'utilisateur ; il sert seulement à passer les informations échangées entre l'application et l'utilisateur
- C'est le handler qui va, selon les callbacks, afficher des informations à l'utilisateur et lui demander des informations qu'il va ensuite ranger dans les callbacks

Utilisation des callbacks (1)

- La méthode login crée des callbacks, par exemple, 2 callbacks **`NameCallback`** et **`PasswordCallback`**
- Elle lance alors un handler en lui passant les callbacks dans un tableau
- Le handler connaît les interactions qu'il doit faire avec l'utilisateur, associées à chaque type de callback
- Il peut interroger un callback (méthodes getXXX) s'il veut des précisions ; pour obtenir, par exemple, un texte à afficher à l'utilisateur

Utilisation des callbacks (2)

- Le handler met dans les callback les informations recueillies auprès de l'utilisateur (méthodes setXXX)
- La méthode login récupère ces informations avec les méthodes getXXX de la classe du callback

Exemple de callback

- **`NameCallback`** implémente l'interface « marqueur » Callback (obligatoire)
- Il a les 2 constructeurs (utilisés au début de la méthode login) :
  - **`NameCallback(String prompt)`**
  - **`NameCallback(String prompt, String nomParDefaut)`**
- Il a les méthodes suivantes :
  - **`getPrompt()`**
  - **`getName()`**
  - **`setName(String nom)`**
  - **`getDefaultName()`**

Callbacks du JDK

- Le plus souvent le développeur utilisera les classes de callbacks fournies avec l'API (ou avec un module de login JAAS)
- Ils sont dans le paquetage **`javax.security.auth.callback`** : **`NameCallback`**, **`PasswordCallback`**, **`TextOutputCallback`**, **`ChoiceCallback`**,…
- Il peut cependant avoir à écrire ses propres callback

CallbackHandlers du JDK

- Le plus souvent le développeur aura à écrire un callbackHandler pour indiquer comment demander des informations à l'utilisateur à l'aide des callbacks
- Il n'y en a pas dans l'API de base mais on en trouve 2 qui demandent un nom et un mot de passe dans le paquetage **`com.sun.security.auth.callback`** : **`DialogCallbackHandler`** (utilise une fenêtre de dialogue Swing), **`TextCallbackHandler`** (demande dans une console)

Écrire un **`CallbackHandler`**

- Il doit implémenter l'interface **`CallbackHandler`** du paquetage **`javax.security.auth.callback`**, c'est-à-dire implémenter la méthode **`void handle(Callback[])`**
- Un exemple est donné dans le tutorial distribué avec le J2SE (docs/guide/security/jaas/tutorials)

JCA/JCE

Les 2 APIs

- L'API pour la cryptographie est partagée en 2 parties à cause des restrictions d'exportation dans certains pays en dehors des États-Unis
- JCA (*Java Cryptography Architecture*) est livrée avec le SDK ; elle contient le paquetage **`java.security`** ; elle est exportable
- JCE (*Java Cryptography Extension*) est une extension ; elle contient le paquetage **`javax.crypto`** ; elle n'est pas exportable dans tous les pays

Annexes

Fichier .security par défaut

```
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.rsajca.Provider
policy.provider=sun.security.provider.PolicyFile
policy.url.1=file:${java.home}/lib/security/java
  .policy
policy.url.2=file:${user.home}/.java.policy
policy.expandProperties=true
policy.allowSystemProperty=true
policy.ignoreIdentityScope=false
keystore.type=jks
system.scope=sun.security.provider.IdentityDatab
  ase
package.access=sun.
```


Fichier .policy par défaut

```
grant codeBase "file:${java.home}/lib/ext/*" {
  permission java.security.AllPermission;
};
grant {
  permission java.lang.RuntimePermission
  "stopThread";
  permission java.net.SocketPermission
  "localhost:1024-", "listen";
  permission java.util.PropertyPermission
  "java.version", "read";
  permission java.util.PropertyPermission
  "java.vendor", "read";
  permission java.util.PropertyPermission
  "java.vendor.url", "read";
```


Fichier .policy par défaut (suite)

```
  permission java.util.PropertyPermission
  "java.class.version", "read";
  permission java.util.PropertyPermission
  "os.name", "read";
  permission java.util.PropertyPermission
  "os.version", "read";
  permission java.util.PropertyPermission
  "os.arch", "read";
  permission java.util.PropertyPermission
  "file.separator", "read";
  permission java.util.PropertyPermission
  "path.separator", "read";
  permission java.util.PropertyPermission
  "line.separator", "read";
```


Fichier .policy par défaut (suite 2)

```
  permission java.util.PropertyPermission
  "java.specification.version", "read";
  permission java.util.PropertyPermission
  "java.specification.vendor", "read";
  permission java.util.PropertyPermission
  "java.specification.name", "read";
  permission java.util.PropertyPermission
  "java.vm.specification.version", "read";
  permission java.util.PropertyPermission
  "java.vm.specification.vendor", "read";
  permission java.util.PropertyPermission
  "java.vm.specification.name", "read";
```


Fichier .policy par défaut (fin)

```
  permission java.util.PropertyPermission
  "java.vm.version", "read";
  permission java.util.PropertyPermission
  "java.vm.vendor", "read";
  permission java.util.PropertyPermission
  "java.vm.name", "read";
};
```


Principaux algorithmes de chiffrement

- DES (Data Encryption Standard), à clés symétriques
- RSA (Rivest, Shamir, Adleman), à clés asymétriques
- RC4, RC5 (Rivest), à clés symétriques
- AES (Advanced Encryption Standard), tout récent, à clés asymétriques