

# Sécurité JVM - 3A STI

Jean-Francois Lalande - December 2015 - Version 2.0

Le but de ce cours est d'aborder les notions de sécurité des langages de programmation, et plus particulièrement les mécanismes fournis dans l'environnement Java.

Les grandes notions abordées dans ce cours sont:

- La sécurité fournie par la JVM
- La sécurité dans Python
- Aspets de sécurité spécifiques à Android



Ce cours est mis à disposition par Jean-François Lalande selon les termes de la [licence](#) Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage à l'Identique 3.0 non transposé.

# 1 Plan du module

## Plan du module

|                           |    |
|---------------------------|----|
| 1 Plan du module          | 2  |
| 2 La sécurité dans la JVM | 3  |
| 3 La sécurité dans python | 12 |
| 4 Bibliographie           | 15 |



## 2 La sécurité dans la JVM

|     |  |    |
|-----|--|----|
| 2.1 | Besoin: le sandboxing                          | 3  |
| 2.2 | Introduction: les composants de sécurité la VM | 3  |
| 2.3 | Politiques de sécurité pour la VM              | 4  |
| 2.4 | Le security manager                            | 6  |
| 2.5 | Le Bytecode verifier                           | 8  |
| 2.6 | Le class loader                                | 10 |
| 2.7 | La sérialisation                               | 10 |
| 2.8 | Le keystore                                    | 10 |

### Objectifs:

- Comment définir des politiques de contrôle des applications Java ?
- Comment implémenter un composant de contrôle d'application Java ?
- Que vérifie la Machine Virtuelle lors de l'exécution du *byte code* ?

### 2.1 Besoin: le sandboxing

Le *sandboxing* est le concept qui recouvre le contrôle d'une application. On souhaite contrôler ce qu'elle fait (les opérations autorisées) et à quoi l'application accède (les ressources).

Les ressources sont multiples:

- Disques locaux, distants et leurs systèmes de fichier
- Mémoire locale / distantes (RAM)
- Serveurs (notamment dans le cas d'une applet)

Le sandboxing classique le plus restreint est de laisser l'application accéder au CPU, à l'écran, clavier, souris, et à une partie de la mémoire. Un degré de liberté supplémentaire consiste à donner accès au disque dur.

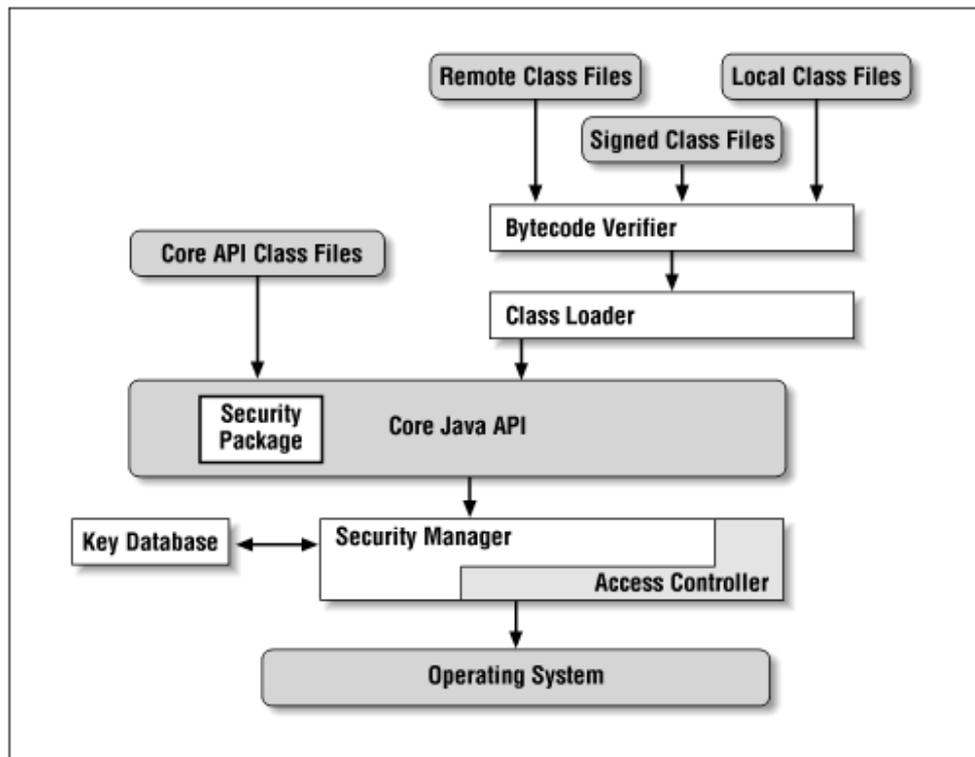
Deux modes classiques d'exécution existent: - Applet: sandboxing le plus restreint - Application: aucun sandboxing

### Objectifs:

- Comment définir des politiques de sandboxing plus fines ?
- Comment charger ces politiques dans la VM ?

### 2.2 Introduction: les composants de sécurité la VM





Architecture de la sécurité dans Java, extrait de <sup>JSEC</sup>.

## Architecture de la sécurité dans Java

### Le Bytecode verifier

Le *Bytecode verifier* s'assure que le code ne viole pas des règles du langage Java (cf [Le Bytecode verifier](#)). Toutes les classes ne sont pas vérifiées, notamment celle du *core API*.

### Le class loader

Le class loader est responsable du chargement des classes qui ne sont pas trouvées dans le CLASSPATH.

### L'access controller

Il autorise ou interdit les accès des classes du *core API* vers l'operating system. C'est une sorte de contrôle des appels système.

### Le security manager

Prend les décisions d'accès aux ressources du système (fichiers, réseau, etc.) en fonction de la politique chargée. Il peut coopérer avec l'*access controller* pour une partie de ses décisions.

### Le security package

Le package `java.security` est un ensemble d'outils dédiés aux mécanismes d'authentification des classes Java signées (clefs, certificats, signatures, chiffrement).

## 2.3 Politiques de sécurité pour la VM

Une politique de sécurité s'écrit sous la forme suivante:

```

grant codeBase "domaine cible" {
  permission X [...];
  permission Y [...];
  permission Z [...];
}
  
```

Le domaine cible est:

- des ressources locales (fichiers)
- des ressources distantes (serveurs)

Les permissions ont:

- un nom
- des paramètres optionnels (actions, types)

Exemple:

```
grant codeBase "http://www.ensi-bourges.fr/" {
    permission java.security.AllPermission;
};
```

### Permissions des politiques (1)

**File permission:** java.io.FilePermission

```
permission java.io.FilePermission "tmpFoo", "write";
permission java.io.FilePermission "<<ALL FILES>>",
    "read,write,delete,execute";
permission java.io.FilePermission "${user.home}/-", "read";
```

**Socket permission:** java.net.SocketPermission

```
permission java.net.SocketPermission "www.ensi-bourges.fr:1-",
    "accept,listen,connect,resolve"
```

**Runtime permission:** java.lang.RuntimePermission

createClassLoader, getClassLoader, createSecurityManager, setIO, stopThread.

```
permission java.lang.RuntimePermission "accessClassInPackage.sdo.foo";
```

### Permissions des politiques (2)

**Security permission:** java.io.SerializablePermission

enableSubstitution: autorise enableReplaceObject() de ObjectInputStream

enableSubclassImplementation: autorise la surcharge de readObject() et writeObject()

**Property permission:** java.util.PropertyPermission

```
permission java.util.PropertyPermission "java.*", "read";
```

**All permission:** java.security.AllPermission

### Domaines

Fichiers:

```
file:/C:/files/truc.txt
file://files/truc.txt
file:${java.home}/files/truc.txt
```

URLs:

```
// One jar filer
http://www.ensi-bourges.fr/files/truc.jar
```

```
// Class files:
http://www.ensi-bourges.fr/files/
// Class and jar files:
http://www.ensi-bourges.fr/files/*
// Class and jar files in any sub directories:
http://www.ensi-bourges.fr/files/-
```

### Exemple complet de fichier de politique

```
keystore "${user.home}${/}.keystore";

grant codeBase "file:${java.home}/lib/ext/-" {
    permission java.security.AllPermission;
};

grant codeBase "http://www.ensi-bourges.fr/files/" {
    permission java.io.FilePermission "/tmp", "read";
    permission java.lang.RuntimePermission "queuePrintJob";
};
```

### Invocation du security manager

Permettre le sandboxing:

```
java -Djava.security.manager TestEcriture
Exception in thread "main" java.security.AccessControlException:
    access denied (java.io.FilePermission testEcriture.txt write)
```

Préciser une politique (url du type http: ou file):

```
java -Djava.security.manager -Djava.security.policy=URL TestEcriture
```

Préciser une et une seule politique:

```
java -Djava.security.manager -Djava.security.policy==URL TestEcriture
```

Appletviewer:

```
appletviewer -J-Djava.security.policy=URL urls
```

## 2.4 Le security manager

Le security manager est mis en oeuvre lorsqu'un appel vers l'API java est faite depuis un objet du programmeur. Principes du security manager:

- Requête du programme vers l'API java
- L'API java demande l'autorisation au security manager
- Si refusée, une exception est levée
- Sinon l'appel est honoré

Il est possible de *catcher* l'exception. Par exemple, dans le cas particulier des applets, il est classique d'avoir l'exception **sun.applet.AppletSecurityException: checkread**. Voici une meilleure façon de procéder, lorsque l'on est informé de l'exception, plutôt que de perdre la trace de l'exception:

```
OutputStream os;
try {
    os = new FileOutputStream("statefile");
```

```

} catch (SecurityException e) {
    os = new Socket (webhost, webport).getOutputStream ();
}
// Ecriture dans os...

```

### Mettre en place un security manager

Changer de security manager:

- Récupérer le security manager courant (ou null)
- Installer le security manager (ne peut plus être enlevé) (Utile pour une applet qui ne peut plus le changer)

```

public static SecurityManager getSecurityManager()
public static void setSecurityManager(SecurityManager sm)

```

Exemple:

```

public class TestSecurityManager {
    public static void main(String args[]) {
        System.setSecurityManager(new SecurityManagerImpl());
        ...do the work of the application... } }

```

### Méthodes du security manager: io

Vérification de lecture/écriture/suppression de fichier:

```

public void checkRead(FileDescriptor fd)
public void checkRead(String file)
public void checkWrite(FileDescriptor fd)
public void checkWrite(String file)
public void checkDelete(String file)

```

`checkRead()` est appelé pour:

```

File.canRead()
File.isDirectory()
File.isFile()
File.lastModified()

```

Extrait Javadoc:

*Check if the current thread is allowed to read the given file. This method is called from `FileInputStream.FileInputStream()`, `RandomAccessFile.RandomAccessFile()`, `File.exists()`, `canRead()`, `isFile()`, `isDirectory()`, `lastModified()`, `length()` and `list()`. The default implementation checks `FilePermission(filename, "read")`. If you override this, call `super.checkRead` rather than throwing an exception.*

### Méthodes du security manager: réseau

Vérifier l'ouverture d'une socket (appelé pour `Socket()`):

```

public void checkConnect(String host, int port)

```

Vérifier l'écoute d'un port et l'acceptation d'une requête (appelé pour `ServerSocket.accept()`):

```

public void checkListen(int port)
public void checkAccept(String host, int port)

```

Changer l'implémentation par défaut d'une socket:



```
public void checkSetFactory()
```

Extrait Javadoc:

*Check if the current thread is allowed to accept a connection from a particular host on a particular port. This method is called by `ServerSocket.implAccept()`. The default implementation checks `SocketPermission(host + ":" + port, "accept")`. If you override this, call `super.checkAccept` rather than throwing an exception.*

### Méthodes du security manager: vm

Protection de la virtual machine:

```
checkCreateClassLoader() --> ClassLoader() // Changement de class loader
checkExit() --> Runtime.exit() // Arrêt de la virtual machine
checkLink() --> Runtime.loadLibrary() // Chargement de code natif
checkExec() --> Runtime.exec() // Exécution de commande shell
```

Protection des Threads avec `checkAccess(Thread g)`:

```
Thread.stop() Thread.interrupt()
Thread.suspend() Thread.resume()
Thread.setPriority() Thread.setName()
Thread.setDaemon()
```

Protection de l'inspection de classes avec `checkMemberAccess()`:

```
Class.getFields()
Class.getMethods()
Class.getConstructors()
Class.getField()
Class.getMethod()
Class.getConstructor()
```

## 2.5 Le Bytecode verifier

L'expérience des langages dérivés du C a conduit les concepteurs de Java à définir un certain nombre de règles de sécurité lors de l'interprétation du *bytecode verifier*. La difficulté de garantir de la sécurité provient aussi de la **dynamicité du langage**: les classes peuvent être chargées à la volée, de manière distante, et peuvent être inspectées.

Par rapport à C++, on doit au moins garantir:

- l'encapsulation: private, protected, public
- la protection contre les accès mémoire arbitraires

Java définit donc des règles qui garantissent que:

- L'attribut privé reste privé (sauf Serialisation)
- La mémoire ne peut pas être accédée arbitrairement
- Entité ayant une méthode final ne peut-être changé - Variable public final - Une sous classe: exemple **Thread.setPriority()**
- Certaines entités sont finales: exemple String (constante)
- Une variable non initialisée ne peut être lue (sinon cela revient à lire la mémoire)
- Les bornes des tableaux sont vérifiées
- Les casts sont vérifiés

Objectif: répondre à la question: ce byte code est-il légal ? (même s'il provient d'un autre langage que java...)

Garanties apportées par le *bytecode verifier* <sup>FR03</sup>:

- des vérification de format (champs obligatoires, taille de ces champs)

- structure du code: vérification du contrôle de flot (saut vers du code vraiment existant), intégrité des instructions à plusieurs octets
- cohérence de la frame (variables locales et opérandes): il ne doit pas y avoir de débordement de la pile d'opérande (*overflow* ou *underflow*) lorsque l'on stocke des résultats intermédiaires; les variables locales doivent être initialisées avant utilisation
- bonne utilisation des appels de méthodes: vérification des droits d'accès, bon nombre et type des paramètres
- Classes finales: n'ont pas de sous classes
- Méthodes finales: non surchargées
- Classe: une seule superclasse

Et à l'exécution:

- Pas de cast illégal (ex int vers Object)
- Vérification upcast/downcast
- Data stack: débordement (récursion infinie)

Les vérifications sont faites avant l'exécution et/ou en temps réel (remplacé par un code spécial indiquant que les tests ont été faits)

### Attaque par définition de classe

```
public class CreditCard {
    public String acctNo = "0001 0002 0003 0004"; }
```

```
public class TestCreditCard {
    public static void main(String args[]) {
        CreditCard cc = new CreditCard();
        System.out.println("Your account number is " + cc.acctNo); }
```

On change public en private:

```
jf@linotte:security/code> javac CreditCard.java
jf@linotte:security/code> javac TestCreditCard.java
jf@linotte:security/code> java TestCreditCard
Your account number is 0001 0002 0003 0004
jf@linotte:security/code> gvim CreditCard.java
jf@linotte:security/code> javac CreditCard.java
jf@linotte:security/code> java TestCreditCard
Exception in thread "main" java.lang.IllegalAccessException:
  tried to access field CreditCard.acctNo from class
  TestCreditCard at TestCreditCard.main(TestCreditCard.java:4)
jf@linotte:security/code> javac TestCreditCard.java
TestCreditCard.java:4: acctNo has private access in CreditCard
    System.out.println("Your account number is " + cc.acctNo);
```

### Attaque par substitution de classe

Autre exemple tiré de [JSEC](#):

- Copy the java.lang.String source file into our CLASSPATH.
- In the copy of the file, modify the definition of value--the private array that holds the actual characters of the string--to be public.
- Compile this modified class, and replace the String.class file in the JDK.
- Compile some new code against this modified version of the String class. The new code could include something like this:

```
public class CorruptString {
    public static void modifyString(String src, String dst) {
        for (int i = 0; i < src.length; i++) {
            if (i == dst.length)
```

```
return;
src.value[i] = dst.value[i]; }}
```

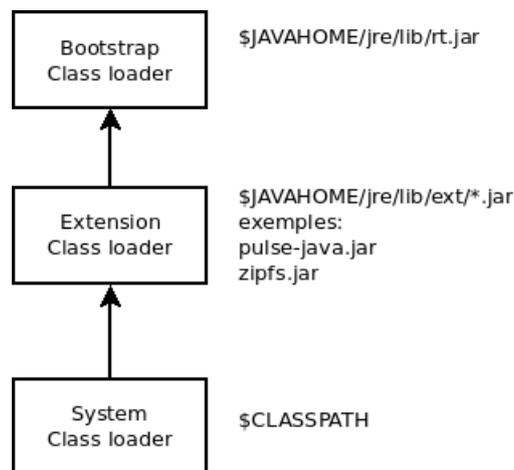
Now any time you want to modify a string in place, simply call this `modifyString()` method with the string you want to corrupt (`src`) and the new string you want it to have (`dst`).

Remove the modified version of the `String` class.

## 2.6 Le class loader

Le class loader permet de résoudre les importations nécessaires à l'exécution d'une classe et d'éviter de charger en mémoire des classes inutiles car non utilisées à l'exécution. De plus, la définition d'une classe peut n'être connue qu'à l'exécution, par exemple si celle-ci est chargée au travers du réseau.

Pour rappel, le *class loader* est en fait une chaîne de délégation (et non pas d'héritage) de *class loader*.



### Sécurité et class loader

Si un *class loader* charge du code depuis un endroit potentiellement attaquable, alors cela augmente la surface d'attaque de l'application. Il faut donc que le *class loader* s'assure de ce qu'il charge: il doit notamment vérifier le nom de *package* des classes chargées. Comme expliqué dans <sup>LJB</sup>, si l'on dépose la classe `java.lang.FooBar` dans un repository externe chargé dynamiquement, ce genre d'objet pourra aller manipuler les attributs *protected* du package `java.lang`, ce qui peut s'avérer dangereux.

Un *class loader* peut éviter un tel comportement en vérifiant le nom du package de ce qu'il charge, par exemple:

```
if (className.startsWith("java."))
    throw new ClassNotFoundException();
```

## 2.7 La sérialisation

La sérialisation est un mécanisme simple pour écrire des objets (et leurs dépendances pour les objets composites) dans des fichiers. Il est aussi utile par exemple pour l'api RMI pour la migration des objets sur le réseau. Pour rendre une classe sérialisable, on utilise un *flag* en implémentant l'interface **Serializable**.

Cependant, la sérialisation révèle les attributs privés:

- Accessible en lecture sur le disque (ou autre buffer)
- Peut même être édité

Le langage Java fournit le mot clef **Transient** pour spécifier qu'une variable ne doit pas être sérialisée.

```
private transient String creditCardNumber;
```

## 2.8 Le keystore



Code signé, cf tool jarsigner.

- keystore: clef publiques des entités signées
- A l'exécution: - Récupération de la clé publique du code signé - Installation dans le keystore (.keystore)
- Administration du keystore: keytool

## 3 La sécurité dans python

### 3.1 Les vieilleries: rexec

Le module rexec était fait pour exécuter du code dans lequel le programmeur n'a pas confiance dans un environnement préparé à cet effet: c'est l'idée classique du sandboxing.

Un code *non-trusted* peut 1) violer des restrictions du bac à sable: rexec lève une exception; 2) attaquer des ressources (cpu, mémoire): non prévu, utiliser un process à part

Constructeur: `class RExec(hooks=None, verbose=False)`

```
r = import rexec
r = rexec.RExec()
```

Méthodes:

- `r.r_import(modname)`: importe le module dans le bac à sable
- `r.r_exec(code)`: exécute le code dans le bac à sable
- `r.r_execfile(filename)`: exécute le fichier dans le bac à sable
- `r.r_add_module(modname)`: ajoute et retourne un pointeur sur module

Attributs de l'objet rexec:

- `nok_builtin_names`: Built-in functions not to be supplied in the sandbox
- `ok_builtin_modules`: Built-in modules that the sandbox can import
- `ok_path`: Used as `sys.path` for the sandbox's import statements

### 3.2 mxProxy - Proxy-Access to Python Objects

mxProxy est une bibliothèque open source soutenue par eGenix.com. Son but est de fournir une méthodologie générale pour restreindre l'accès à des attributs ou des méthodes de classe.

Le proxy fournit une protection des attributs de classe en définissant une sorte d'interface: une liste des noms des attributs vers lesquelles les accès sont autorisés.

**Et les méthodes et attributs privés ?**

- ils existent et peuvent être utilisé à l'aide du `__`
- leur caractère privé est une convention, rien n'est garanti.

```
1 class Toto():
2     def test(self):
3         return 10;
4     def __test2(self):
5         return 15;
6 if __name__ == "__main__":
7     t = Toto()
8     print t.test(); # Ok, that's public
9     #print t.test2(); # Not ok, that's private
10    print t._Toto__test2(); # Ok, too !
```

L'appel d'une fonction reste toujours possible en utilisant `__Classe__methode`.

#### Types de proxy

Constructeur du proxy pour un objet:

```
Proxy(object, interface=None, passobj=None)
```

- `interface`: séquence de string autorisant l'accès aux attributs/méthodes

- `passobj`: permet d'autoriser la récupération de l'objet via `.proxy_object()`

Proxy avec `read-cache`: met en cache les attributs de l'objet *wrappé*

```
CachingInstanceProxy(object, interface=None, passobj=None)
```

(attention aux remises à jour de l'objet *wrappé*)

Proxy de type `read-only`:

```
ReadonlyInstanceProxy(object, interface=None, passobj=None)
```

Proxy factory: permet de créer des instances de Class *wrappé* dans un Proxy

```
ProxyFactory(Class, interface=None)
```

### Méthodes sur un Proxy: récupérer les attributs

Méthode permettant de récupérer des attributs:

- `.proxy_getattr(name)`: récupère l'attribut `name`
- `.proxy_setattr(name)`: modifie l'attribut `name`
- `.proxy_defunct()`: retourne 1 si l'objet a déjà été garbage collecté

```
1 from mx import Proxy
2 class DataRecord:
3     a = 2
4     b = 3
5     # Make read-only:
6     def __public_setattr__(self, what, to):
7         raise Proxy.AccessError('read-only')
8     # Cleanup protocol
9     def __cleanup__(self):
10        print 'cleaning up', self
11 o = DataRecord()
12 # Wrap the instance:
13 p = Proxy.InstanceProxy(o, ('a',))
14 # Remove o from the accessible system:
15 del o
16
17 print p.proxy_getattr('a')
18 print p.a
```

### Méthodes sur un Proxy: récupérer les méthodes

Pour récupérer des méthodes:

```
1 from mx import Proxy
2 class DataRecord:
3     a = 2
4     b = 3
5     # Make read-only:
6     def __public_setattr__(self, what, to):
7         raise Proxy.AccessError('read-only')
8     # Cleanup protocol
9     def __cleanup__(self):
10        print 'cleaning up', self
11     def essai(self):
12        print 'coucou'
13 o = DataRecord()
14 # Wrap the instance:
```

```
15 p = Proxy.InstanceProxy(o,('essai',))
16 # Remove o from the accessible system:
17 del o
18
19 print p.essai()
20 print p.b # Ne marche pas
```

### ***Substitution de classe par son proxy***

```
1 from mx import Proxy
2 class DataRecord:
3     a = 2
4     b = 3
5     # Make read-only:
6     def __public_setattr__(self,what,to):
7         raise Proxy.AccessError('read-only')
8     # Cleanup protocol
9     def __cleanup__(self):
10        print 'cleaning up',self
11 # If you want to have the wrapping done automatically, you can use
12 # the InstanceProxyFactory:
13 DataRecord = Proxy.InstanceProxyFactory(DataRecord,('a',))
14 # This gives the same behaviour...
15 p = DataRecord()
16 print p.a
17 #p.a = 3 # interdit par le code de setattr
18 #print p.b # interdit par le proxy
```

## 4 Bibliographie

---

- |            |   |
|------------|---|
| JSE        | The Java Tutorials: <a href="#">Exceptions</a> .  |
| Rifflet    | Une présentation des principaux <a href="#">concepts objets</a> dans Java 1.4.2, J.-M. Rifflet, 28 juin 2005. |
| JUCookbook | <a href="#">JUnit Cookbook</a> , Kent Beck and Erich Gamma.   |
| JSEC(1, 2) | Scott Oaks, Java Security, O'Reilly.  |
| LJB        | Chick Mcmanis, The basics of Java class loaders, 1996.  |
| FR03       | Fabrice Rossi, L'architecture de sécurité de Java, MISC n°7, 2003.  |