L'architecture de sécurité de Java

Fabrice Rossi
http://apiacoa.org/

Mai/Juin 2003

Cet article est paru dans le numéro 7 de la revue MISC (Multi-System & Internet Security Cookbook) en mai 2003. Je remercie Frédéric Raynal, Denis Bodor et les éditions Diamond pour avoir autorisé la distribution de cet article selon les conditions indiquées dans la section A. L'article est disponible en ligne aux URLs http://www.miscmag.com/articles/index.php3?page=808 et http://apiacoa.org/publications/2003/misc-java-security.pdf

Résumé

La plate-forme Java gagne régulièrement de l'importance du côté serveur, en particulier avec la montée en puissance des serveurs JSP et des architectures de type Entreprise Java Beans. A l'origine, Java était associé aux applets et a conservé de ses débuts un mécanisme d'exécution sécurisée de codes potentiellement dangereux. L'architecture de sécurité a évolué avec les versions de Java et propose maintenant une infrastructure très intéressante tant pour les applications sur le poste client que pour l'utilisation sur le serveur. Cet article a pour objectif de présenter les mécanismes utilisés pour construire l'architecture de sécurité de Java et les possibilités de celle-ci.

1 Introduction

Le premier positionnement commercial de la plate-forme Java a été historiquement l'extension des navigateurs web du côté client, avec les applets. Rappelons que ce modèle consiste à télécharger un programme et à l'intégrer dans le navigateur lors d'un accès normal à une page web, et ce de façon totalement transparente pour l'utilisateur. Pour limiter les dangers inhérents à ce mécanisme, les concepteurs de Java ont intégré dès l'origine du langage un mécanisme d'exécution de programmes potentiellement dangereux dans un environnement contrôlé fondé sur le modèle classique du sandbox (littéralement, le bac à sable). Les utilisations de la plate-forme Java dépassent maintenant très largement le simple cadre des applets, ce qui rend le mécanisme du sandbox particulièrement intéressant, en particulier pour un emploi du côté serveur.

Le principe du sandbox est conceptuellement simple : il s'agit de fournir un environnement d'exécution d'un programme totalement contrôlé, c'est-à-dire dans lequel chaque opération potentiellement dangereuse (en particulier les accès à des ressources comme la mémoire, les fichiers, le réseau, etc.) est soumise à une autorisation préalable. Le mécanisme de chroot d'unix est un exemple de sandbox rudimentaire qui consiste à cacher un pan entier des ressources à un programme. La difficulté pratique principale est bien sûr son implémentation sans bug, mais il ne faut pas négliger l'aspect applicatif. L'enjeu n'est pas d'empêcher tout

accès (ce qui est relativement "facile"), mais au contraire de contrôler le plus finement possible l'accès à toutes les ressources.

L'architecture de sécurité de Java a évolué au fil des versions du jdk, passant d'un sandbox fonctionnel mais rudimentaire dans le jdk 1.0 à une architecture beaucoup plus souple et évoluée dans la version actuelle (1.4.1), que nous décrivons dans cet article.

2 Bref rappel sur Java

Pour comprendre l'architecture de sécurité de Java, il est important d'avoir en tête quelques caractéristiques majeures de la plate-forme, terme qui désigne trois composants :

1. la machine virtuelle Java (JVM pour Java Virtual Machine)
2. le langage Java
3. l'ensemble des api Java (les classes systèmes)

Conceptuellement, un programme Java est constitué de plusieurs classes fondées sur les classes systèmes. Lors de la compilation, le programme est traduit en byte-code, une sorte d'assembleur portable d'assez haut niveau (il comporte par exemple des instructions orientées objets et un *garbage collector*). L'exécution d'un programme Java est réalisée par un programme particulier, la machine virtuelle Java, qui se charge d'exécuter le byte-code, par interprétation ou par compilation au vol (*Just In Time*).

La structure de la plate-forme Java se retrouve dans l'architecture de sécurité : celle-ci intervient en effet au niveau du langage lui-même, au niveau de la machine virtuelle et au niveau des classes systèmes. Une autre conséquence de l'architecture de la plate-forme est que les mécanismes de sécurité travaillent sur du byte-code au format très contrôlé et d'un niveau d'abstraction bien supérieur à du code objet, ce qui facilite leurs tâches.

Java est un langage fortement typé, c'est-à-dire que toute valeur ou objet manipulé possède un type et que toute méthode possède une signature (c'est-à-dire qu'elle précise les types des paramètres qu'elle s'attend à recevoir). Il est impossible (en l'absence de bug dans la JVM utilisée) de contourner le système de typage, c'est-à-dire qu'on ne peut pas passer un réel (un double) à une méthode qui s'attend à recevoir un entier (un int) par exemple. Notons aussi que Java propose un contrôle d'accès au niveau des classes et de leur contenu (méthodes et variables). En effet, une variable, par exemple, peut être private, c'est-à-dire que seule la classe qui la définit peut la manipuler. D'autre part, la JVM vérifie les accès aux tableaux, ce qui interdit les débordements.

Précisons pour finir que le chargement dynamique de composants logiciels est au coeur de beaucoup de programmes Java. Un navigateur charge une nouvelle applet en pleine exécution des autres applets ; un moteur de servlets comme Tomcat fait la même chose avec les pages JSP et les servlets ; on retrouve le mécanisme dans les serveurs EJB, etc. Le point à retenir est que dans beaucoup de cas, la JVM peut être considérée comme un véritable ordinateur muni d'un système d'exploitation et qu'un programme Java est dynamiquement chargé dans la JVM pour être exécuté, en concurrence avec les autres programmes déjà en cours d'exécution dans la même machine. Le mécanisme de chargement des classes est donc au centre de la plate-forme Java et par conséquent au centre de l'architecture de sécurité.

3 Les grandes lignes de l'architecture de sécurité

L'architecture de sécurité de Java est fondée sur deux concepts : la notion de domaine et celle de politique de sécurité. La notion de domaine est une généralisation du mécanisme de sandbox proposé dans les versions 1.0 et 1.1 du jdk. Un domaine est en fait un ensemble de classes et d'objets Java, caractérisé par l'origine (physique) du byte-code des classes et par les certificats correspondants aux clés éventuellement utilisées pour signer les byte-codes. A chaque domaine est associé un ensemble de permissions, c'est-à-dire un ensemble de droit d'accès à différentes ressources (le droit de lire un fichier, celui d'ouvrir une socket, etc.). Une politique de sécurité est un ensemble de règles qui associent à chaque domaine des permissions, en se fondant essentiellement sur l'origine du byte-code des classes du domaine et sur les signatures de ce byte-code.

L'architecture de sécurité est réalisée à partir de plusieurs composants de la plate-forme Java :

1. le vérificateur de byte code (byte-code verifier) qui assure en particulier que le typage fort de Java est bien respecté
2. le chargeur de classes (classloader) détermine (entre autre) les domaines en précisant l'origine du byte-code
3. le contrôleur d'accès (access controller) implémente les droits d'accès

Nous allons détailler dans les sections suivantes ces différents composants.

4 Le vérificateur de byte-code

4.1 Principe

L'architecture de sécurité de Java repose sur des caractéristiques fondamentales du langage, en particulier le typage fort, sans lequel elle s'écroule. Le premier rempart d'une JVM est donc son vérificateur de byte-code qui a pour rôle, comme son nom l'indique, de s'assurer que les classes que la JVM charge ne contiennent pas un byte-code qui tente de contourner les contraintes imposées par le langage. Il s'agit à la fois de protéger la JVM contre les bugs des compilateurs Java, contre un byte-code spécifiquement écrit pour provoquer des problèmes, mais aussi de gagner en efficacité dans l'exécution du byte-code. Par exemple, les instructions du byte-code de la JVM sont typées. Grâce à une analyse du byte-code, la JVM s'assure une fois pour toute que les contraintes de type sont vérifiées, ce qui évite de tester les types à chaque exécution du byte-code correspondant.

4.2 Les vérifications effectuées

La JVM réalise de très nombreuses vérifications sur un byte-code (décrites avec précision dans la spécification de la machine virtuelle [4]). A des vérifications de format (présence des champs obligatoires, taille des champs, etc.) s'ajoute une étude systématique du code Java qui est validé en accord avec la définition de l'assembleur de la JVM. Le byte-code est en fait interprété de façon abstraite, c'est-à-dire que la JVM simule l'exécution du code sans réaliser véritablement les opérations et les appels de méthodes, et en ne tenant compte que des types des entités manipulés. Les éléments étudiés comprennent par exemple :

- **structure du code** : la JVM vérifie que les branchements dans le code se font bien vers du code existant, que l'intégrité des instructions est respectée (quand une instruction comporte plusieurs octets, on ne peut pas avoir un branchement au milieu de ceux-ci, etc.), que les blocs d'exception sont correctement disposés, etc.
- **frame de chaque méthode** : dans la JVM, une méthode comporte une frame contenant deux zones de tailles fixes déterminées à la compilation : un tableau de variables locales et une pile des opérandes (utilisées pour le stockage des résultats intermédiaires des calculs et appels de méthodes). La JVM s'assure que l'utilisation de variables locales est toujours correcte (variables initialisées avant utilisation, types corrects, etc.). De même elle vérifie que la pile des opérandes ne subit pas de débordement et que les types sont toujours corrects lors de l'utilisation du contenu de cette pile par des instructions de la JVM.
- **utilisation des objets et classes** : la JVM vérifie que les méthodes sont appelées seulement si elles peuvent l'être (vérification des droits d'accès) avec des paramètres corrects (en nombre et en type), que les variables sont utilisées en accord avec les droits d'accès et les types, que les objets sont construits correctement, etc.

La vérification du byte-code permet donc d'assurer que les contraintes imposées par le langage Java (essentiellement le contrôle d'accès et le typage fort) sont satisfaites. L'architecture de sécurité est très dépendante du respect de ces contraintes. Par exemple, une classe ne peut pas modifier son domaine car l'objet qui le représente ne propose pas de méthode de modification et ne donne pas accès à son contenu (qui est vraisemblablement entièrement représenté par des variables private). Le typage fort interdit d'accéder directement à la mémoire en transformant un entier en un pointeur, etc.

5 Les chargeurs de classes

5.1 Principe

Comme nous l'avons rappelé plus haut, beaucoup de programmes Java sont chargés dynamiquement dans une JVM déjà en cours d'exécution. Ce chargement est réalisé par un objet particulier dont la classe hérite de ClassLoader. Le rôle d'un chargeur de classes est essentiellement de convertir un flux d'octets en un objet Class qu'on peut ensuite utiliser pour étudier la classe correspondante ou encore pour créer un objet de cette classe (grâce à l'API d'introspection). A ce rôle opérationnel s'ajoute un aspect très important, la notion d'espace de noms (namespace). On pense en général qu'une classe est simplement identifiée par son nom complet, comme par exemple java.lang.String. En fait, dans une JVM, l'identification se fait par un couple composé du nom complet de la classe et du ClassLoader qui a été utilisé pour charger la classe.

Depuis le jdk 1.2, les ClassLoaders sont organisés de façon hiérarchique au sens ou chaque ClassLoader possède un père, à savoir le ClassLoader qui l'a chargé. Le ClassLoader bootstrap (aussi appelé le ClassLoader null) est situé en haut de cette hiérarchie. Il ne s'agit pas stricto sensu d'une classe Java, mais plutôt d'une implémentation native (spécifique à chaque système) de ClassLoader, destinée à charger les classes système sans lesquelles la JVM ne peut pas fonctionner.

5.2 Rôle dans le mécanisme d'exécution sécurisée

Les ClassLoaders jouent un rôle fondamental dans la mise en place de l'architecture de sécurité. Comme indiqué plus haut, chaque classe d'un programme doit appartenir à un domaine (décrit par une instance de ProtectionDomain) dont la mise en place est assurée par le ClassLoader (qui se charge donc de renseigner les informations importantes, à savoir l'origine du byte-code et les éventuelles signatures).

D'autre part, on utilise généralement plusieurs ClassLoaders pour éviter certaines attaques. En effet, pour des raisons d'efficacité, les classes sont chargées par la JVM de façon paresseuse, c'est-à-dire quand on en a besoin. De plus, un ClassLoader ne recharge jamais une classe qu'il connaît déjà. On imagine alors facilement une attaque de la façon suivante. On s'arrange pour faire charger par la JVM le programme d'attaque avant le programme attaqué. Dans le programme d'attaque, on inclut des classes qui portent le même nom que les classes sensibles du programme attaqué, et qui reproduisent les interfaces de celles-ci, mais en ajoutant des portes dérobées. Par exemple, on peut rendre une variable sensible publique. Quand le programme attaqué est chargé, il utilise naturellement les classes déjà chargées qui remplacent donc les classes qu'il devrait normalement utiliser. Le programme d'attaque peut alors utiliser ses portes dérobées pour obtenir des informations sensibles, modifier le comportement du programme attaqué, etc.

Diverses techniques sont proposées pour rendre impossible ce genre d'attaque. Pour lutter directement contre l'attaque décrite ci-dessus, les navigateurs web (par exemple) utilisent un ClassLoader différent par applet. Les mécanismes de la JVM interdisent totalement les accès entre des classes chargées par des ClassLoaders distincts. Plus précisément, une classe donnée n'a accès directement qu'au ClassLoader qui l'a chargée. Grâce à la structure hiérarchique des ClassLoaders elle peut ensuite accéder au père de son ClassLoader, puis de remonter dans la hiérarchie jusqu'au bootstrap. Cependant, la descente n'est pas possible : aucune méthode de ClassLoader n'existe pour accéder aux fils d'un chargeur donné. Si un navigateur doit charger deux applets A et B, il fabrique donc deux ClassLoaders, LA et LB, sans relation hiérarchique. Toutes les classes de A sont chargées par LA, alors que celles de B sont chargées par LB. Même si A et B définissent chacun une classe fr.toto.Foo, les deux applets possèdent une version chacune de cette classe, sans aucune interaction possible.

Pour préserver les classes systèmes, les ClassLoaders standards (dérivés de SecureClassLoader) utilisent un mécanisme de délégation. Quand une classe D demande le chargement d'une autre classe C, la JVM vérifie que C n'a pas déjà été chargée. Si ce n'est pas le cas, elle demande le chargement de C au ClassLoader de D, LD. Celui-ci délègue le chargement à son père. Si le père n'arrive pas à charger C, alors LD tente un chargement direct, par l'appel d'une méthode spécifique (par exemple par téléchargement du byte-code). Comme le mécanisme de délégation est récursif, toute classe système est chargée par le ClassLoader bootstrap. Bien entendu, rien n'empêche de fabriquer un ClassLoader qui ne délègue pas. C'est pourquoi la création d'un ClassLoader est une opération considérée comme sensible et protégée par le mécanisme de sécurité. Notons d'autre part qu'une application Java classique est chargée par un ClassLoader standard (à savoir l'URLClassLoader) qui respecte bien entendu le mécanisme de délégation. Conjointement avec le mécanisme décrit ci-dessous, il n'est donc pas possible d'écraser les classes systèmes (au sein de la JVM).

D'autres attaques et protections sont décrites dans [6] et [3].

6 Le contrôleur d'accès

6.1 Principe général

La dernière pièce de l'architecture de sécurité de Java est le contrôleur d'accès, implémenté par la classe AccessController. Les droits d'accès à une ressource sont représentés en Java par des objets dont la classe hérite de Permission. Par exemple, les accès à des fichiers sont représentés par des objets FilePermission. Il est ainsi possible de définir dans un programme son propre système de droits d'accès à diverses ressources (par exemple les manipulations d'un compte bancaire) en utilisant le contrôleur d'accès de Java. Quand une opération sensible doit être réalisée par un programme Java, celui-ci commence par créer un objet Permission qui représente les droits correspondant à l'opération sensible. Ensuite, le programme appelle la méthode checkPermission de la classe AccessController, avec comme paramètre l'objet en question. Cette méthode rend la main si l'opération est permise, c'est-à-dire si le code en cours d'exécution possède bien les droits décrits par l'objet passé en paramètre. Dans le cas contraire, la méthode échoue avec une exception de type AccessControlException. Dans l'exemple suivant, le programme vérifie qu'il peut accéder en lecture au fichier /etc/shadow :

```
FilePermission perm =
  new FilePermission("/etc/shadow", "read");
AccessController.checkPermission(perm);
```

L'exemple proposé est assez caractéristique des utilisations du contrôleur d'accès. En effet, un droit d'accès est caractérisé par un nom et un ensemble d'action. Le type du droit d'accès est une caractérisation de haut niveau de ce droit. Par exemple SocketPermission concerne les droits d'accès au réseau. Le nom du droit d'accès précise la ressource : dans notre exemple, il s'agit du nom du fichier. Enfin, la liste d'actions (réduite ici dans l'exemple à "read") précise les opérations que le code a le droit de réaliser sur la ressource. En général, on les représente sous forme d'une liste de mots clés séparés par des virgules. Notons que la plate-forme Java définit un nombre important de classes correspondant à des droits d'accès et que les bibliothèques système protègent toutes les ressources sensibles de cette façon (cf [2]).

6.2 Détermination des droits

Le contrôleur d'accès utilise un algorithme assez simple pour déterminer si le code qui effectue l'appel à checkPermission possède effectivement les droits représentés par l'objet passé en paramètre. A tout moment dans un programme, la JVM connaît le contexte (la pile d'appels), c'est-à-dire la liste des méthodes appelantes qui conduisent au code en cours d'exécution depuis le point d'entrée dans le programme. Le contrôleur d'accès utilise cette pile pour déterminer le domaine de chacune des classes traversées. Grâce à la politique de sécurité, il détermine les droits d'accès associés à chaque domaine et en réalise l'intersection. Le code en cours d'exécution obtient ainsi les droits d'accès seulement si la chaîne d'appels qui mène à ce code ne traverse que des classes qui ont les droits d'accès en question. Considérons l'exemple simple suivant. On propose d'abord une classe Main :

```
package main;
public class Main
{
  public static void main(String[] args)
```

```
  {
    Truc.test();
  }
}
```

et une classe Truc :

```
package main;
import java.io.File;
public class Truc
{
  public static void test()
  {
    File f=new File("/tmp/my-test-file");
    f.delete();
  }
}
```

On regroupe les deux classes dans le fichier main.jar, ce qui forme une application (quand on déclare Main comme classe principale). Quand on lance l'application, on finit par exécuter la méthode `f.delete()` dans la classe Truc. Le contrôleur d'accès vérifie alors que l'effacement du fichier /tmp/my-test-file est bien autorisé. La vérification est demandée par la classe File qui est une classe système. La pile d'appels contient à ce moment les méthodes Main.main, Truc.test et File.delete. Les classes systèmes ont bien sûr tout les droits sinon les programmes Java seraient intrinsèquement très limités ! Donc, en particulier, File a le droit d'effacer le fichier concerné (au moins au sens Java, le système d'exploitation n'est pas obligatoirement d'accord !). Cependant, la règle d'intersection fait que pour que la suppression du fichier soit autorisée, il faut que Main et Truc possèdent elles aussi le droit d'effacement.

Quand on lance une application Java, la machine virtuelle n'installe pas de contrôle d'accès et le programme fonctionne sans poser de problème (à condition bien sûr que le système d'exploitation permette la suppression du fichier considéré). Pour lancer une application avec un contrôle d'accès, il faut donner une valeur à la propriété java java.security.manager. On peut par exemple écrire :

```
java -Djava.security.manager -jar main.jar
```

La propriété java.security.manager indique le gestionnaire de sécurité à installer (dans les versions du jdk antérieures à la 1.2, l'objet SecurityManager jouait un rôle similaire à celui d'AccessController ; en interne, la plupart des bibliothèques Java continuent d'utiliser le SecurityManager, bien que celui-ci délègue maintenant ses actions à l'AccessController). Quand on n'installe pas de politique de sécurité, les classes locales (les classes de main.jar dans notre exemple) n'ont aucun droit et le programme est donc interrompu par une exception de sécurité.

6.3 Politique de sécurité

La politique de sécurité donne des droits aux classes d'un domaine. Elle est représentée par un objet Policy. Dans l'implémentation de référence (le jdk de sun), cet objet est renseigné grâce à un fichier texte qui décrit l'association entre les domaines et les droits. La syntaxe

exacte de ce fichier dépasse le cadre de cet article et nous renvoyons donc le lecteur à [2]. Voici un exemple associé à notre application de test (fichier policy.txt) :

```
grant codebase "file:main.jar" {
  permission java.io.FilePermission "/tmp/my-test-file", "delete";
};
```

Le sens de ce fichier est le suivant : le bloc grant donne le droit de supprimer (delete) le fichier /tmp/my-test-file au domaine correspondant aux classes chargées dans main.jar, depuis le dossier courant (c'est le sens de l'URL file :main.jar). Pour activer la politique de sécurité précisée dans ce fichier, il faut indiquer à la machine virtuelle son emplacement par l'intermédiaire de la propriété java.security.policy qui peut contenir l'URL désignant le fichier (ici le fichier policy.txt donné au dessus). Par exemple, on peut faire l'appel suivant :

```
java -Djava.security.manager -Djava.security.policy=policy.txt -jar main.jar
```

Cette fois-ci le programme fonctionne parfaitement. En effet, la classe File est une classe système et a donc toujours le droit d'effacer un fichier. Comme les deux classes de l'application sont dans main.jar qui est lui même dans le dossier courant, elles appartiennent au domaine décrit par la politique de sécurité et ont donc elles aussi le droit d'effacer le fichier. Le contrôleur d'accès autorise donc l'effacement.

De façon plus générale, la politique de sécurité se fonde sur l'emplacement d'un byte-code et sur les signatures associées pour définir les domaines et donc donner des droits d'accès. On peut par exemple donner le droit de lire des fichiers à un byte-code téléchargé depuis un URL donné à condition qu'il soit signé par une clé donnée. Les signatures sont obtenues à partir de certificats X.509. Le jdk de sun est livré avec différents outils permettant la gestion des certificats et des clés, la signature d'un byte-code ainsi que l'édition simple (avec interface graphique) des fichiers de politique de sécurité. L'intégration au fichier de politique de sécurité des droits définis par des objets spécifiques à l'application ne pose aucun problème.

Depuis la version 1.4 du jdk, les domaines peuvent aussi tenir compte de l'identité de l'utilisateur du programme Java. En effet, l'api Java Authentication and Authorization Service (JAAS) a été intégrée au jdk. Elle permet d'utiliser diverses techniques pour identifier un utilisateur (certificat X.509, tickets kerberos, etc.).

Notons pour finir que l'architecture de sécurité propose un mécanisme de passage de privilèges. En effet, l'algorithme de détermination des droits est très restrictif car il tient compte du contexte de l'appel. En utilisant la méthode doPrivileged de la classe AccessController, on supprime temporairement le prise en compte du contexte : le contrôleur d'accès ne tient plus compte des méthodes qui ont appelé la méthode qui utilise doPrivileged. Seuls les droits de cette méthode sont considérés. L'idée est d'introduire un contrôle très fin des droits : on peut ainsi autoriser une toute petite partie du code à réaliser certaines opérations sensibles, sans pour autant empêcher l'appel du code en question par le reste de l'application. Un bug dans le reste de l'application ne pourra pas contourner la protection puisque les opérations sensibles restent interdites. On retrouve ici un mécanisme assez proche des capabilities POSIX.

7 Critique de l'architecture

Notre principale critique à l'architecture de sécurité de Java est l'absence de protection fondée sur le "volume" d'utilisation d'une ressource. Il est impossible, par exemple, de limiter

la taille des fichiers créés, la puissance CPU utilisée, la bande passante consommée, etc. Bien entendu, il est parfaitement possible d'utiliser les mécanismes proposés par le système d'exploitation pour réaliser ce contrôle, mais ce n'est pas dans l'esprit de Java (qui met toujours en avant la portabilité des solutions proposées). Cela est d'autant plus gênant que l'architecture propose un contrôle très fin sur le reste des opérations de la JVM et offre une solution d'exécution sécurisée d'applications extérieures très avancée.

Un autre point important concerne bien sûr les bugs. Ceux-ci ont été nombreux depuis les premières versions du jdk, et ils continuent à apparaître relativement régulièrement. Par exemple, un bug permettant de provoquer l'arrêt d'une JVM à partir d'une applet (ou de tout autre code) a été découvert en mars 2003 par Marc Schoenefeld (cf [7]). De très nombreux bugs ont été découvert dans le passé (le livre Securing Java [5] en contient une liste et on peut en trouver une qui couvre les bugs découverts jusqu'à la fin 2002 sur le site de Sun [1]).

8 Conclusion

L'architecture de sécurité de Java propose des mécanismes très évolués pour exécuter du code dans des conditions contrôlées finement. Elle permet de construire des applications sécurisées par des mécanismes qui s'ajoutent à ceux proposés par le système d'exploitation hôte de la machine virtuelle. Il est possible en particulier d'exécuter des applications distinctes protégées les unes des autres au sein d'une même JVM, c'est-à-dire sans passer par un support noyau (et au final hardware) pour la protection de la mémoire. Cependant, comme toute technologie relativement récente, l'architecture comporte encore quelques limitations et a une histoire tourmentée en terme de bugs. Une bonne pratique à l'heure actuelle consiste, pour une utilisation serveur, a cumuler les protections en utilisant plusieurs JVM séparées les unes des autres par le système d'exploitation. Pour une utilisation sur le client (les applets), il convient, comme pour toute technologie sensible, d'être particulièrement vigilant et donc de réaliser des mises à jour régulières du plugin Java ou du navitageur utilisé.

Références

[1] Chronology of security-related bugs and issues, November 2002. Disponible à l'URL `http://java.sun.com/sfaq/chronology.html`.

[2] Li Gong. *Java 2 Platform Security Architecture*. SUN, 2002. Disponible à l'URL `http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-spec.doc.html`.

[3] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java Virtual Machine. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, Vancouver, 1998. Disponible à l'URL `http://citeseer.ist.psu.edu/liang98dynamic.html`.

[4] Tim Lindholm and Frank Yelling. *The Java Virtual Machine Specification*. Java Series. Addison-Wesley, September 1996. Disponible à l'URL `http://java.sun.com/docs/books/vmspec/index.html`.

[5] Gary McGraw and Ed Felten. *Securing Java*. John Wiley & Sons, Inc., 1999. Disponible à l'URL `http://www.securingjava.com/`.

[6] Vijay Saraswat. Java is not type-safe. Disponible à l'URL `http://citeseer.nj.nec.com/saraswat97java.html`.

[7] Marc Schoenefeld. Denial-of-service holes in jdk 1.4.1_01, 2003. Cf http://www.illegalaccess.org et http://www.securityfocus.com/bid/7109.

A Licence