Sécurité de la plate-forme d'exécution Java : limites et propositions d'améliorations

Guillaume Hiet[1], Frédéric Guihéry[1], Goulven Guiheux[1], David Pichardie[2], Christian Brunette[3]

[1] AMOSSYS
[2] INRIA Rennes
[3] SILICOM Région Ouest

Résumé Le choix de Java est souvent guidé par la sécurité qu'il est censé apporter. La plate-forme d'exécution Java assure en effet des propriétés de sécurité permettant notamment de se prémunir contre l'exploitation de la mémoire. Toutefois, de nombreuses vulnérabilités publiques concernent Java, notamment sa bibliothèque standard. Qu'en est-il donc de l'apport de Java en termes de sécurité ? Quelles sont ses faiblesses ? Quelles améliorations sont envisageables ? Cet article tente de répondre à ces questions. Il décrit tout d'abord les différents composants de la plate-forme d'exécution Java en détaillant les mécanismes de sécurité offerts et les propriétés garanties. Il analyse ensuite les différentes faiblesses de ces composants et propose enfin des pistes pour l'amélioration de la sécurité de la plate-forme.

1 Introduction et problématique

Les biens d'un système, qu'il s'agisse d'un poste client ou d'un serveur, requièrent différents besoins de sécurité (intégrité, confidentialité et disponibilité). Toutefois, l'exécution d'applications sur un poste client ou un serveur peut remettre en cause la sécurité du système :

- une application, ou une des bibliothèques qu'elle utilise, peut comporter des vulnérabilités qu'un attaquant pourra exploiter ;
- une application ou une bibliothèque peut être piégée ou malveillante.

Il existe donc un besoin de renforcement de la sécurité au niveau applicatif.

Java paraît être un candidat idéal pour répondre à ce besoin, car il est connu pour être un langage de programmation et un environnement d'exécution « sécurisé ». Le modèle initial de Java repose en effet sur différents mécanismes de sécurité (vérification de *bytecode*, chargement de classes signées, exécution cloisonnée dans une *sandbox*, etc.), dont le fonctionnement nominal doit permettre d'assurer un niveau de sécurité minimal.

Toutefois, la réalité se montre quelque peu différente car Java a beaucoup évolué depuis le modèle initial et les premières implémentations. En outre, certains besoins de sécurité ne sont toujours pas couverts. Pour étayer ces propos, voici quelques constatations :

- la JVM est de plus en plus complexe : les premières versions utilisaient un modèle d'exécution par interprétation de *bytecode*, les dernières versions ont recours à la compilation à l'exécution (*Just In Time* ou JIT), et à la compilation dynamique (basculement entre l'interprétation et le JIT pendant l'exécution). Cette complexité pose le problème de la confiance dans la JVM, qu'il est de plus en plus difficile d'évaluer ;
- il existe plusieurs vulnérabilités CVE concernant l'implémentation de Sun, essentiellement localisées dans la bibliothèque standard ;
- la gestion des droits d'accès aux ressources par le système de contrôle d'accès de Java est malheureusement peu utilisée ;
- il n'est pas toujours évident de mettre en œuvre les services de sécurité fournis par l'OS au niveau des applications Java.

Il paraît donc intéressant de faire un état des lieux du gain effectif qu'apporte le choix de Java, en termes de sécurité. L'essentiel des travaux existants sur la sécurité de Java est d'ordre académique (preuves formelles, optimisations lors de la compilation, etc.). Certains travaux se concentrent sur la recherche de vulnérabilités dans la bibliothèque standard [21,23] ou évoquent principalement les aspects spécifiques aux applications Web (J2EE) [19,18]. Peu de travaux [15,20,17] se sont attachés à évaluer la robustesse des mécanismes intrinsèques de Java. Le présent article s'inscrit dans cette démarche. Il complète les travaux précédents sur différents points :

- il présente une analyse globale des mécanismes de sécurité de Java ;
- il analyse les faiblesses de ces mécanismes et de la plate-forme d'exécution Java en général ;
- il propose des évolutions permettant d'améliorer la sécurité de la plate-forme Java.

Les résultats présentés dans cet article sont issus de l'étude JAVASEC réalisée pour la sous-direction « Assistance, Conseil et Expertise » de l'ANSSI (Agence Nationale de la Sécurité des Systèmes d'Information). JAVASEC a pour objet l'adéquation de Java pour le développement d'applications de sécurité. Cette étude a donné lieu à différents travaux d'analyse sur le langage Java, les modèles d'exécution, les compilateurs et les JVM. En particulier, trois JVM *open source* ont été analysées (OpenJDK, cacaovm et JamVM). Des propositions ont été faites afin d'améliorer la sécurité des applications Java. Certaines s'adressent aux développeurs Java, sous la forme d'un guide de règles et recommandations pour le développement. D'autres concernent le déploiement ou l'évolution de la JVM. Des travaux d'évolution d'une JVM sont en cours et pourront donner lieu à une évaluation.

Cet article ne se veut pas exhaustif sur les problématiques de sécurité dans Java, mais vise plutôt à mettre en exergue plusieurs points qui nous paraissent soit critiques, soit symptomatiques d'une évolution de Java qui a tendance à oublier ses principes fondateurs[4]. L'article est organisé de la manière suivante : la section 2 présente l'architecture de la plate-forme d'exécution Java, ses mécanismes de sécurité et les propriétés de sécurité assurées ; la section 3 analyse

[4] `http://java.sun.com/docs/white/langenv/Intro.doc2.html`

les faiblesses de cette plate-forme au niveau de la bibliothèque standard et de la JVM ; la section 4 propose enfin différentes approches pour renforcer la sécurité de la plate-forme d'exécution Java.

2 Rappels sur l'architecture de Java

Le langage et l'environnement d'exécution Java sont issus de travaux de recherche menés dans les années 1990 par Sun Microsystems dans le cadre du projet Stealth (renommé Green project par la suite). Le but était de définir un langage de haut niveau, orienté objet et s'inspirant de C++, tout en comblant les lacunes de ce dernier en ce qui concerne la gestion de la mémoire (ajout d'un ramasse-miettes ou *garbage collector* en anglais) et la programmation concurrentielle (gestion native de plusieurs *threads*). Le projet visait initialement le marché des clients légers (PDA, etc.), mais il se réoriente au début de l'année 1990 sur les applications liées au Web.

L'essor et la popularité de Java dépendent en grande partie de l'essor des technologies Web. La mise à disposition gratuite par Sun de l'environnement d'exécution et des outils de développement (compilateur) via internet a également contribué à son succès. Les différentes versions du langage et de l'environnement d'exécution se sont succédées depuis la version 1.0, qui est apparue en 1996, à la dernière version stable (1.6), disponible depuis 2006 [5].

Il ne s'agit pas ici de décrire en détail les principes et la sémantique du langage Java. Le lecteur désirant s'initier à la programmation Java pourra se référer aux documents [11,8,5,6]. Les ouvrages [9,11,4,12] constituent quant à eux des références sur le langage et sa bibliothèque standard. La figure A.1 en annexe illustre la sémantique du langage Java.

Un des objectifs de Java était de faciliter le déploiement des applications en s'assurant qu'un même programme Java puisse s'exécuter sur des environnements différents (UNIX, Windows, Mac, etc.). Sun a pour cela décidé de définir une plate-forme (ou environnement) d'exécution standard, s'appuyant sur une machine virtuelle. L'architecture de cette plate-forme est détaillée dans la section suivante.

2.1 Description générale de l'architecture

L'environnement d'exécution d'un programme Java, illustré par la figure 1, comprend différents éléments plus ou moins standardisés.

La figure distingue trois ensembles :

1. L'application Java est développée par le programmeur à l'aide du langage Java. Le code source de cette application est compilé vers une forme intermédiaire : le *bytecode* Java. C'est ce dernier qui est exécuté par la plate-forme

[5] Sun a modifié plusieurs fois la dénomination commerciale, et la numérotation, des différentes versions de la plate-forme Java. Par soucis de simplicité, nous utilisons systématiquement dans cet article la numérotation employée par les développeurs de la plate-forme (la version 1.6 est désignée commercialement sous le nom Java SE 6)

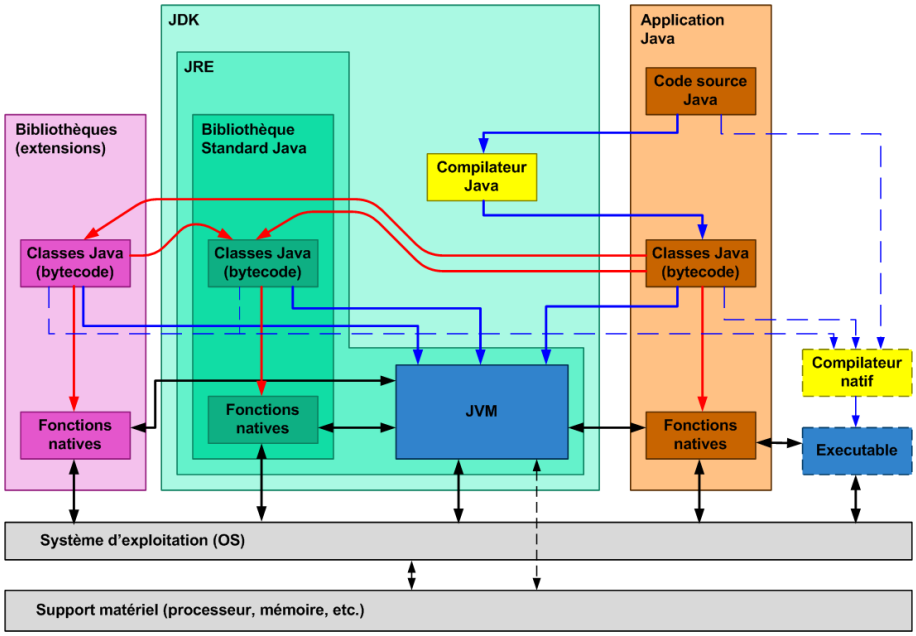

Fig. 1. Architecture de la plate-forme d'exécution Java

d'exécution Java. L'application peut également comprendre du code natif (développé par exemple en C ou C++) pour réaliser les opérations qui ne peuvent être spécifiées en langage Java. L'usage de ce type de code n'est toutefois pas conseillé, car il limite notamment la portabilité de l'application.

2. Afin d'exécuter les applications Java sur des plate-formes matérielles et logicielles (OS) différentes, Sun a défini une plate-forme d'exécution standard s'appuyant sur une machine virtuelle : la JVM (Java Virtual Machine). Outre la JVM, la plate-forme comprend également une bibliothèque standard Java. Ces deux éléments essentiels sont décrits plus en détail en section 2.3 et 2.4. L'ensemble minimal des éléments permettant d'exécuter une application Java, qui comprend entre autres une implémentation de la JVM et de la bibliothèque standard, est désigné sous le terme de JRE (Java Runtime Environment). Le JDK (Java Development Kit) constitue un sur-ensemble du JRE permettant de développer des applications Java. Il comprend, outre les éléments du JRE, le compilateur Java qui convertit les fichiers sources (en Java) en fichiers `class` (en *bytecode* Java).
3. L'application Java peut également s'appuyer sur des bibliothèques tierces (désignées ici sous le terme de bibliothèques d'extensions). Celles-ci sont généralement fournies sous la forme de fichiers `class` (*bytecode*) regroupées dans des archives (JAR). Ces bibliothèques peuvent elles aussi utiliser du code natif. À la différence de la bibliothèque standard, ces bibliothèques ne font pas partie de la plate-forme d'exécution standard et doivent être

installées séparément. Elles permettent simplement de ré-utiliser du code existant sans disposer nécessairement du code source.

En théorie, les spécifications fournies par Sun permettent à différents éditeurs d'implémenter leur propre version de la plate-forme d'exécution standard. En pratique, la tâche n'est pas aisée dès lors qu'il s'agit de garantir une compatibilité totale vis-à-vis de l'implémentation de référence fournie par Sun. En effet, la spécification est incomplète. Comme l'illustre la figure 6 en annexe, seuls certains éléments possèdent une spécification publique :

- le langage Java, décrit dans le Java Language Specification [12], bien que cette spécification soit relativement informelle ;
- l'API de la bibliothèque standard [6] ;
- la JVM [22].

La spécification de la JVM est relativement abstraite. Elle définit les différents blocs fonctionnels et leurs rôles ainsi que certaines interfaces. En revanche, elle donne peu de détails sur la manière d'implémenter ces blocs fonctionnels. Différentes stratégies d'implémentation de la JVM ont donc vu le jour :

- l'implémentation sous la forme d'un émulateur logiciel (utilisée notamment par HotSpot, l'implémentation de référence fournie par Sun) ;
- l'utilisation d'un support matériel (généralement via l'utilisation d'un co-processeur comme c'est le cas, par exemple, pour la technologie Jazelle d'ARM [7]) ;
- l'intégration dans un binaire incluant l'application Java (par exemple, GCJ [8]).

La quasi totalité des implémentations repose sur la première stratégie. Cet article s'appuie donc essentiellement sur elle. Si les différences entre les implémentations de la JVM ont, en pratique, très peu d'impacts sur la compatilibilité des applications Java, elles peuvent en revanche avoir un impact en termes de sécurité. Ce point est détaillé dans les sections suivantes.

À la différence de la JVM, la bibliothèque standard est un composant dont la spécification publique est incomplète : seules les couches « hautes » de la bibliothèque (l'API, les prototypes des éléments publics accessibles au développeur Java) sont définies. L'implémentation des classes et le prototype même de certaines classes (par exemple, celle du package `sun` de l'implémentation de Sun) ne sont pas définis. En outre, le couplage entre une implémentation de la bibliothèque standard et celle d'une implémentation de la JVM est important. En effet, certaines fonctionnalités offertes par la bibliothèque standard (typiquement, la gestion des *threads* Java) requièrent des services qui doivent être fournis par la JVM et qui ne sont pas accessibles via une opérande du *bytecode*.

Pour ces raisons, il est relativement difficile de fournir une implémentation complète de la bibliothèque standard, à la différence de la JVM. Seules certaines implémentations propriétaires (par exemple, celle fournie avec J9, la JVM d'IBM) atteignent un tel niveau de compatibilité. Dans le monde open-source,

[6] `http://java.sun.com/javase/6/docs/api/`
[7] http ://www.arm.com/products/multimedia/java/jazelle.html
[8] http ://gcc.gnu.org/java/

GNU Classpath [9], qui constitue le projet le plus abouti, n'implémente pas la totalité de l'API Java SE6 [10]. L'implémentation de Sun, dont une version open-source est disponible via le projet OpenJDK [11], est donc quasiment incontournable.

La plate-forme d'exécution Java a été introduite principalement pour des raisons de portabilité. Toutefois, dès les premières versions de Java, elle implémente des fonctions de sécurité destinées notamment à garantir un certain nombre de propriétés de sécurité intrinsèques au langage Java. Ces fonctions de sécurité sont fournies par la JVM ou la bibliothèque standard. Elles peuvent être classées suivant différentes catégories :

- les fonctions de sécurité visant à garantir une propriété intrinsèque au langage et qui s'appliquent automatiquement à tout programme Java sans nécessiter de paramétrage particulier ;
- les fonctions de sécurité qui s'appliquent à tout code Java mais nécessitent un paramétrage de la part du développeur ou de l'utilisateur (par exemple, le contrôle d'accès JPSA) ;
- les fonctions de sécurité qui sont disponibles mais que le développeur doit utiliser explicitement (par exemple, les API cryptographiques).

La section 2.2 présente les propriétés intrinsèques du langage Java ainsi que les mécanismes qui les assurent. Les sections 2.3 et 2.4 détaillent les mécanismes de sécurité implémentés respectivement dans la JVM et la bibliothèque standard.

2.2 Les propriétés de sécurité

Le langage Java, à la fois dans sa version source et *bytecode*, est doté de propriétés intrinsèques fortes. Un langage de bas niveau comme C autorisera des programmes sans comportement clairement défini (comme la forge de pointeur par trans-typage), ou bien avec des comportements définis mais dangereux et difficiles à prévoir avant l'exécution (comme les débordements de tampon). Dans le monde Java (en excluant bien sûr les méthodes natives écrites en C), de tels comportements, qui sont potentiellement vecteurs d'attaques, sont soit interdits (par un rejet du compilateur ou du vérificateur de *bytecode* de la machine virtuelle), soit donnent lieu à la levée d'une exception spécifique pendant l'exécution du programme. Ces trois niveaux de barrières (vérification lors de la compilation, vérification de *bytecode*, vérification dynamique) assurent toute une panoplie de propriétés intrinsèques. Un point important est que certaines propriétés sont vérifiées deux fois : une première fois par le compilateur et une deuxième fois par le vérificateur de *bytecode*. Les vérifications du compilateur qui ont trait à la sécurité sont systématiquement reproduites au niveau *bytecode*. On peut donc considérer que le compilateur n'implémente pas de fonction de sécurité (son bon fonctionnement est toutefois requis afin de garantir que la sémantique du programme Java est préservée lors de la compilation).

Les principales propriétés de sécurité de Java sont présentées par la suite. Le tableau de la figure 2 les résume en précisant quel mécanisme les met en œuvre.

[9] `http://www.gnu.org/software/classpath/`
[10] `http://builder.classpath.org/japi/openjdk6-classpath.html`
[11] http ://openjdk.java.net/

Propriétés (extrait)	**Compilateur**	**Vérifieur de *bytecode***	**Vérifications dynamiques**
Ni arithmétique de pointeurs, ni forge de pointeurs	Système de type fort entre types références et types numériques	Système de type similaire, moins la séparation booléen / numérique	Aucune vérification
Pas de dépassement des bornes d'un tableau	Aucune vérification	Aucune vérification	Si dépassement, exception spécifique
Champ `final` écrit une seule fois	Analyse de flots de données des constructeurs	Aucune vérification	Exception spécifique si le champ est écrit par une méthode hors-classe
Typage des instructions (respect de la hiérarchie de classes)	Système de types (toute variable a un type déclaré)	Analyse de flots de données pour inférer le type des variables ou vérification par *stackmaps*	Si type invalide, exception `ClassCastException` (nécessaire pour l'écriture dans un tableau)
Initialisation correcte des variables	Analyse de flots de données pour assurer l'écriture avant lecture des variables locales	Analyse de flots de données pour assurer l'écriture avant lecture des variables locales	Mise à zéro des champs lors de l'allocation des objets

Fig. 2. Quelques propriétés intrinsèques et lieux de mise en œuvre

La distinction entre valeur numérique et pointeur mémoire est assurée statiquement par le compilateur et par la vérificateur de *bytecode*. À l'exécution, aucune vérification n'est réalisée, ce qui place le vérificateur de *bytecode* comme dernier rempart contre l'arithmétique de pointeur illicite.

La défense contre les attaques de type « dépassement de tampon » est assurée uniquement à l'exécution. La spécification officielle de la JVM impose donc d'enregistrer la taille allouée dynamiquement pour chaque tableau et de vérifier que chaque accès a lieu dans les bornes. En pratique, les compilateurs optimisant embarqués dans la JVM (JIT ou AOT) prennent parfois l'initiative de supprimer certaines de ces vérifications coûteuses lorsqu'ils prédisent (par une analyse statique à la volée) qu'un accès est nécessairement correct.

Un champ d'instance déclaré `final` subit une vérification poussée par le compilateur, mais très légère une fois le programme compilé au format *bytecode*. Le compilateur assure qu'un tel champ est écrit au moins une fois et au plus une fois sur tous les chemins d'exécution d'un constructeur. Cette propriété n'étant pas considérée comme liée à la sécurité, elle disparaît quasiment totalement lors de la compilation au format *bytecode*. À l'exécution, la JVM empêche seulement

d'affecter un champ `final` depuis une méthode qui ne serait pas définie dans la classe où le champ est déclaré. Il s'agit d'une différence flagrante de changement de propriété intrinsèque entre le niveau source et le niveau *bytecode*.

Les systèmes de type source et *bytecode* assurent que la plupart des instructions recevront des arguments du bon type à l'exécution. Certaines opérations nécessitent cependant une vérification dynamique, car elles sont trop difficiles à valider par vérification statique. C'est le cas des affectations de tableau pour lequel il est nécessaire que la valeur affectée ait un type compatible avec le type déclaré pour le tableau. Une telle vérification est inutile lors de l'affectation d'un champ.

Les variables locales d'une méthode doivent toujours être écrites au moins une fois avant d'être lues. Cette vérification a lieu à la fois au niveau source et au niveau *bytecode*. Il est ainsi inutile que la JVM initialise les variables locales (autres que les paramètres) au lancement de chaque méthode. La situation est différente pour les champs (statiques ou d'instance) : aucune vérification statique n'assure généralement qu'ils seront affectés avant d'être lus. La JVM leur donne donc une valeur par défaut à chaque création de classe (pour les champs statiques) et d'objet (pour les champs d'instance). Sans ce mécanisme, une lecture mémoire illicite serait possible.

2.3 La machine virtuelle Java (JVM)

Le terme JVM couvre différentes notions :
- il désigne la spécification abstraite fournie par Sun (*JVM specifiation*) ;
- il désigne également une implémentation de cette spécification : Hotspot (l'implémentation de Sun), J9 (l'implémentation d'IBM), cacaovm, JamVM, etc ;
- il est parfois employé pour désigner une instance particulière de cette implémentation exécutant une application Java.

La confusion est d'autant plus grande que le terme est souvent utilisé à tort pour désigner le JRE, qui comprend, outre une implémentation de la JVM, une implémentation de la bibliothèque standard.

Comme évoqué précédemment, il existe différentes stratégies d'implémentation de la JVM, cet article se focalisant sur l'implémentation sous la forme d'un émulateur logiciel. Dans la plupart des cas, la JVM est développée dans un langage compilé nativement (C, C++, etc.), sous la forme d'une bibliothèque partagée. L'initialisation de la JVM et le chargement de la classe principale de l'application Java qu'elle exécute sont généralement réalisés par un lanceur générique (la commande `java`). Il est également possible « d'embarquer » la JVM dans une application native et de développer des lanceurs personnalisés en utilisant JNI.

La notion d'application Java désigne un programme autonome fourni sous forme de *bytecode* et s'exécutant sur une JVM. La spécification de cette dernière ne précise pas si une instance d'une JVM permet d'exécuter une ou plusieurs applications. Toutefois, dans la majorité des cas, les implémentations de la JVM adoptent le modèle suivant : chaque application Java s'exécute sur une instance

différente de la JVM sous la forme d'un processus de l'OS de la plate-forme native.

Description de l'architecture de la JVM. L'architecture de la JVM, illustrée par la figure 3, comprend différents blocs fonctionnels :

- Le bloc fonctionnel de **chargement** assure les étapes de chargement des classes (identification et lecture du fichier `class`) et d'édition des liens (vérification de la classe, préparation de l'espace mémoire et résolution des noms). Il délègue la vérification au vérificateur de *bytecode*. Il s'appuie également sur les chargeurs de classes Java (notamment ceux fournis par la bibliothèque standard) pour l'étape de chargement.
- Le **vérificateur de *bytecode*** s'assure que le fichier `class` est correctement formé et que le *bytecode* des méthodes de la classe vérifie un certain nombre de propriétés. Il garantit en quelque sorte l'innocuité du code chargé dans la JVM. Il s'agit de la principale fonction de sécurité de la JVM.
- La **gestion de la mémoire** gère l'allocation de la mémoire nécessaire à l'exécution de l'application Java. Le code et les valeurs des champs sont stockés dans le tas Java. Les valeurs des variables locales et les paramètres des méthodes sont stockés sur les piles Java (une pile par *thread* Java), qui sont parfois distinctes des piles natives correspondant à l'exécution de fonction natives appelées depuis le code Java.
- La désallocation des objets du tas Java et le déchargement des classes sont gérés automatiquement par le **glanneur de cellules** ou ramasse-miettes (*garbage collector*). Un objet est désalloué dès lors qu'il n'est plus accessible par aucun code susceptible de s'exécuter.
- L'**exécution du *bytecode*** est un des principaux blocs fonctionnels de la JVM. Différentes implémentations plus ou moins complexes peuvent être utilisées, de l'interpréteur au compilateur à la volée (JIT, *Just In Time*) ou en avance de phase (AOT, *Ahead Of Time*).
- Java supporte nativement différents **fils d'exécution** (*thread* Java). Cette fonctionnalité est offerte par la bibliothèque standard qui s'appuie généralement sur les mécanismes fournis par l'OS (les *green threads* sont aujourd'hui rarement employés) et un support fourni par la JVM. Celle-ci peut gérer, outre les *threads* Java, différents *thread* correspondant à des services de la JVM (notamment le *garbage collector*). Elle doit également gérer les problèmes de synchronisation entre ces *threads*.
- La JVM comprend également des **interfaces**, notamment JNI et JVMTI, cette dernière permettant à un programme externe d'interférer dans l'exécution d'une application Java à des fins de mise au point.

L'objectif de cet article n'est pas de présenter en détail le fonctionnement et les différentes stratégies d'implémentation de l'ensemble de ces blocs fonctionnels. Le lecteur désireux d'approfondir ces éléments pourra consulter les ouvrages de référence [22] ainsi que les sites Internet des projets open-source d'implémen-

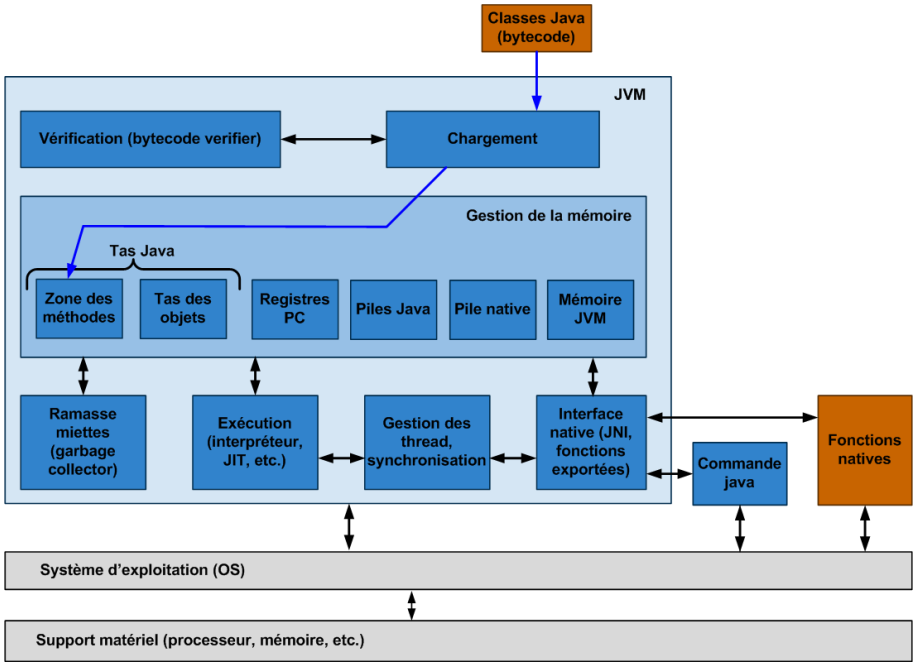

Fig. 3. Architecture de la JVM

tation de la JVM, notamment OpenJDK [12], cacaovm [13] et JamVM [14]. Certains blocs fonctionnels méritent cependant une description plus approfondie en raison de leur criticité et de leur importance pour la sécurité. Le fonctionnement de la principale fonction de sécurité de la JVM, le vérificateur de bytecode, est décrit plus en détail dans le paragraphe suivant. Les principales stratégies d'implémentation des blocs d'exécution et de gestion de la mémoire sont explicités en annexe A.3 et A.4.

Vérificateur de *bytecode*. Le vérificateur de *bytecode* vérifie les classes d'un programme une par une lors de leur chargement dynamique. Il assure plusieurs propriétés indispensables pour la sécurité comme l'absence d'arithmétique de pointeurs ou de *forge* de pointeurs à partir de valeurs numériques, ou encore que le compteur de programme pointera, à tout moment de l'exécution, sur une instruction valide de la méthode en cours. Ces propriétés ne sont plus vérifiées par la suite afin d'améliorer les performances de l'exécution en libérant l'interpréteur des tests dynamiques correspondants.

Une machine virtuelle Java dispose de deux stratégies pour vérifier une classe. Les deux types de vérifications sont l'inférence et la vérification (au sens strict

[12] `http://openjdk.java.net/`
[13] `http://www.cacaovm.org/`
[14] `http://jamvm.sourceforge.net/`

du terme) par *stackmaps*. Dans le premier cas, la machine virtuelle calcule entièrement l'information nécessaire à la vérification de la conformité du *bytecode*. Dans le second cas, une information pré-calculée (typiquement par le compilateur) est incluse dans le fichier. La présence de cette information, appelée table de *stackmaps*, permet d'accélérer le processus de vérification. Cette information ajoutée aux programmes ne peut pas corrompre la vérification de *bytecode*, car aucune confiance ne lui est accordée. Elle évite seulement une recherche itérative à un problème d'équation de flots de données. Le vérificateur s'assure facilement de la validité de l'information transmise.

La vérification d'une classe n'est cependant pas totalement statique (*i.e.* réalisée avant l'exécution de la classe) car certaines parties de la vérification sont effectuées pendant l'exécution de la classe. Par exemple, l'instruction de lecture d'un champ d'un fichier `.class` est de la forme `getfield C.f:`τ avec `C` le nom de la classe où le champ `f` est censé être déclaré avec le type τ. Le vérificateur de *bytecode* utilise l'information de typage τ sans vérifier l'existence du champ `f` dans la classe `C` avec le bon type. Ce n'est que lors de l'exécution de l'instruction que le champ `f` et ses informations de typage seront recherchés dans la classe `C`. Il est par conséquent possible de faire accepter par le vérificateur de *bytecode* le programme suivant, pourtant mal typé, mais n'exécutant jamais la lecture `a.f` fautive.

```
public class A { static int g = 2; A f; }
public class B {
    public int f(A a) { return a.f;}
    public static void main(String[] arg)
      { System.out.println(""+A.g); }
}
```

2.4 La bibliothèque standard

La bibliothèque standard Java constitue le second élément essentiel du JRE dont l'importance est souvent négligée. Depuis la version 2 de Java, celle-ci est déclinée suivant trois variantes :

- Java SE (Standard Edition), qui constitue l'environnement standard de référence ;
- Java ME (Micro Edition), dédiée aux applications embarquées ;
- Java EE (Enterprise Edition), dédiée aux applications Web (notamment grâce aux technologies Servlet, JSP et EJB).

L'étude JAVASEC se concentre sur le cœur de Java donc sur l'environnement standard Java SE et n'aborde pas les spécificités de Java ME ou Java SE. L'exécution de code distribué (de type *applet*) ou d'applications côté serveur (JSP, Servlet, WebServices, etc.) sort du cadre de cette étude. Ces aspects particuliers ne sont donc pas évoqués en détail dans cet article.

Les composants de la bibliothèque standard. L'API de la bibliothèque standard est publique et documentée par Sun pour chaque version de Java [15]. Celle-ci est implémentée sous forme de classes Java regroupées en *packages*. La bibliothèque standard comprend également des classes ainsi que des fonctions natives implémentées en C/C++ qui sont utilisées par les classes de l'API mais qui n'implémentent pas de méthodes de l'API (par exemple, `sun.misc.VM`). Ces classes sont propres à chaque implémentation et ne sont donc pas normalisées.

La bibliothèque standard de Java offre un nombre important de fonctionnalités dont entre autres :

- les classes fondamentales du langage Java (`java.lang`) comme les chaînes de caractères (`String`) ou les classes enveloppant les types primitifs (`Integer`, `Character`, etc.) ;
- la gestion des interfaces graphiques (`java.lang.awt`, `javax.swing`, etc.) ;
- la gestion des entrées/sorties, notamment sur les fichiers gérés par l'OS (`java.lang.io`, `java.lang.nio`, etc.) ;
- la gestion des communications réseau, notamment via la notion de *socket* (`java.net`, `javax.net`, etc.) ;
- les classes implémentant des mécanismes de sécurité comme le mécanisme de contrôle d'accès (`java.security`) ou des primitives cryptographiques (`javax.crypto`).

Ce composant du JRE représente une quantité de code importante. Il est essentiel pour garantir la portabilité des applications Java. Il ne s'agit pas ici de décrire l'intégralité des services proposés, mais de s'intéresser plus particulièrement aux fonctions de sécurité. Les principaux services de sécurité proposés par la bibliothèque standard sont les suivants :

- le contrôle d'accès en fonction de l'origine du code (JPSA [16], Java Plateform Security Architecture) ;
- le mécanisme de vérification des signatures et de l'intégrité des archives JAR ;
- le mécanisme d'authentification et de contrôle d'accès orienté utilisateur (Java Authentication and Authorization Service) ;
- le mécanisme de chargement de classes qui participe au contrôle d'accès et permet une certaine forme de cloisonnement entre les classes d'une application Java ;
- les API cryptographiques (JCA, JCE) et leurs implémentations sous forme de fournisseurs (*providers*).

Ces mécanismes nécessitent une implication du développeur ou de l'utilisateur de l'application Java plus ou moins importante. En théorie, le contrôle d'accès JPSA et la vérification de signatures s'appliquent au code Java sans nécessiter de modification explicite de ce code. Le fournisseur de l'application ou de la bibliothèque Java doit seulement réaliser une opération de signature

[15] `http://java.sun.com/javase/6/docs/api/`

[16] `http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-spec.doc.html`

et spécifier éventuellement la politique de contrôle d'accès nécessaire pour que l'application fonctionne (la politique étant définie par l'utilisateur ou l'administrateur du système). En revanche, les autres mécanismes nécessitent une prise en compte explicite du code Java. Il s'agit essentiellement de fonctionnalités que le développeur choisit d'utiliser. En outre, à la différences des autres mécanismes, le contrôle d'accès JPSA et le chargement de classes nécessitent un support particulier de la JVM. Il ne s'agit pas ici d'analyser en détail ces différents mécanismes qui sont décrits dans les ouvrages de référence, en particulier [10] et [16]. Le mécanisme de contrôle d'accès JPSA est toutefois présenté plus en détail par la suite car il s'agit d'un mécanisme complexe sur lequel repose une part importante de la sécurité de la plateforme Java.

Description du contrôle d'accès JPSA. JPSA constitue un mécanisme de contrôle d'accès « orienté code ». À la différence des mécanismes de contrôle d'accès « orientés utilisateur », utilisés notamment dans les OS, il s'agit de contrôler l'accès à des ressources en fonction des caractéristiques du code exécuté (ici, l'origine des classes auxquelles appartiennent les méthodes exécutées). Initialement, ce mécanisme a été proposé pour implémenter le bac-à-sable permettant de restreindre les accès des applets téléchargées.

Ce mécanisme s'appuie sur des permissions, qui sont des objets Java héritant de la classe `java.security.Permission`. Chaque permission définit une autorisation d'accès à une ressource « critique » pour la sécurité. Il s'agit typiquement d'une ressource de la plate-forme native (fichier, *socket* réseau, etc.) ou d'une méthode critique (par exemple, le constructeur des chargeurs de classes).

L'architecture et le fonctionnement de JPSA est complexe. Ce contrôle est assuré principalement par du code Java exécuté par la JVM. La figure 4 en donne une vue synthétique. Cette figure distingue trois types d'éléments :

- les composants qui participent à l'identification et l'authentification des classes (identifiés en vert sur la figure 4) ;
- les composants qui permettent de spécifier la politique de contrôle d'accès (identifiés en orange sur la figure 4) ;
- les composants qui appliquent les règles de contrôle d'accès (identifiés en rouge sur la figure 4).

La mise en œuvre du contrôle s'appuie sur le gestionnaire de sécurité (une instance de la classe `java.lang.SecurityManager`) qui doit être installé explicitement pour chaque instance de la JVM (via une option de la ligne de commande ou depuis le programme Java). Chaque point de contrôle appelle la méthode `checkPermission` du gestionnaire de sécurité en lui passant en paramètre la permission requise pour réaliser l'accès. Depuis la version 1.4 de Java, l'implémentation du gestionnaire de sécurité utilisée par défaut délègue la vérification au contrôleur d'accès. Celui-ci s'appuie également sur le gestionnaire de la politique de sécurité qui lui fournit l'ensemble des permissions associées à chaque classe selon la politique de sécurité (permissions dynamiques). L'implémentation du gestionnaire de politique utilisée par défaut utilise des fichiers texte permettant de spécifier la politique de contrôle d'accès.

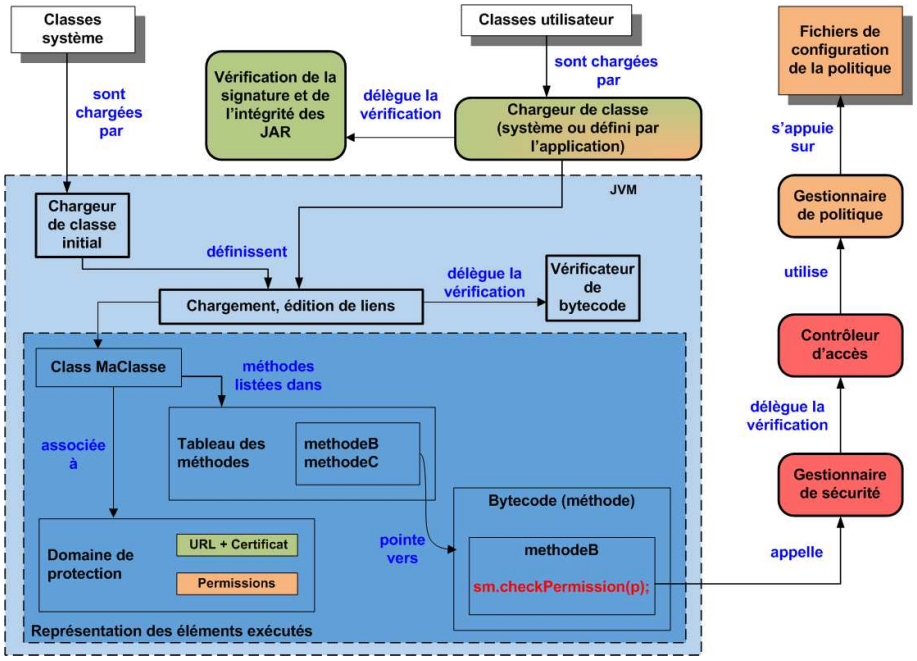

Fig. 4. Contrôle d'accès (JPSA)

Les chargeurs de classes participent également au bon fonctionnement de ce mécanisme. Ils doivent en effet associer un domaine de protection à chaque classe en précisant les éléments qui permettent d'identifier et d'authentifier l'origine de la classe : son URL et les certificats des fournisseurs qui ont signé la classe. Ce deuxième élément est présent uniquement si la classe a été signée et vérifiée (le chargeur de classes déléguant cette vérification).

L'algorithme de contrôle d'accès mis en œuvre par le contrôleur d'accès (et d'autres classes qu'il utilise) consiste à vérifier que **toutes** les méthodes de la pile d'exécution du *thread* ayant réalisé l'accès contrôlé possèdent la permission requise. Pour cela, le code Java s'appuie sur la JVM qui lui fournit l'ensemble des domaines de protection de la pile d'exécution lors du contrôle. Ce comportement très restrictif peut être relâché par l'utilisation des actions privilégiées qui constituent en quelque sorte l'équivalent de la commande UNIX **su**. Elle permettent de limiter le contrôle à un sous-ensemble des méthodes de la pile d'exécution. Seuls les domaines de protection de la « dernière » méthode de la pile, ayant réalisée l'opération contrôlée, ainsi que ceux de ses méthodes appelantes jusqu'à un certain niveau sont pris en compte. Ceux des méthodes appelantes situées au-delà d'un certain niveau ne sont pas considérés (notamment celui de la méthode qui a initié le *thread*).

Concrètement, les intérêts du mécanisme de contrôle d'accès de Java sont les suivants :

- il permet d'appliquer des restrictions différentes en fonction de l'application Java exécutée, notamment lorsque le mécanisme de contrôle d'accès de l'OS ne peut distinguer ces applications (par exemple lorsqu'elles sont exécutées sous la même identité, au sens du contrôle d'accès de l'OS [17]) ;
- il permet de restreindre les droits de l'application Java par rapport à ceux de la JVM (il est par exemple envisageable de permettre à la JVM de lire les fichiers de configuration dont elle a besoin, mais d'en interdire l'accès à l'application Java qu'elle exécute) ;
- il permet également, au sein d'une même application Java, d'adapter les privilèges des différents blocs fonctionnels afin d'appliquer le principe de minimum de privilèges. Ceci permet notamment de limiter l'impact de l'exploitation d'éventuelles vulnérabilités ou de classes malicieuses (ou piégées) fournies par une bibliothèque tierce ;
- outre l'accès aux ressources natives, ce mécanisme permet de restreindre l'utilisation de certains mécanismes ou méthodes Java dont l'utilisation peut s'avérer dangereuse pour la sécurité et qu'il est le seul à pouvoir contrôler : la réflexion (qui permet notamment, en l'absence de restriction du contrôle d'accès, de contourner les règles de visibilité), le chargement de classes (qui peut permettre de piéger l'application), le chargement de bibliothèques natives, etc. Le mécanisme doit également restreindre les accès aux classes participant à la mise en œuvre du contrôle (les chargeurs de classes, le gestionnaire de sécurité, etc.) afin qu'une classe vulnérable, piégée ou malicieuse ne puisse modifier son comportement. Cette forme d'auto-protection est nécessaire du fait que le mécanisme est quasi-entièrement implémenté en Java, la JVM n'offrant qu'un support minimal.

Si, dans la plupart des cas, le développeur doit se contenter de spécifier la politique de contrôle d'accès, il peut parfois être amené à implémenter des points de contrôle ou à modifier des classes. Dans tous les cas, la mise en place d'un tel contrôle nécessite un effort de structuration du code lors de la conception afin de pouvoir distinguer clairement les différents blocs fonctionnels et déterminer pour chacun d'eux le minimum de privilèges requis. Cet effort relève toutefois des bonnes pratiques de programmation. Dans le cadre de la conception d'applications de sécurité à haut niveau de confiance, cet effort paraît nécessaire. De plus, le mécanisme de contrôle d'accès de Java est souple et modulaire. Il offre la possibilité au développeur de redéfinir les différents éléments qui implémentent ce mécanisme. Il s'agit là d'une fonctionnalité intéressante permettant au développeur de mettre en œuvre des contrôles spécifiques non prévus par l'implémentation par défaut. Cependant, la modification de ces éléments n'est pas sans risque, comme nous le verrons dans la section 3.1.

[17] Toutefois, le mécanisme de contrôle d'accès de Java ne peut distinguer différentes exécutions d'une même application Java, puisqu'il repose sur le code de l'application.

3 Les faiblesses de la plate-forme Java

La plate-forme d'exécution Java offre certes des garanties concernant la sécurité des applications Java exécutées mais elle comporte également des faiblesses. Celles-ci sont de différents niveaux :

- elles peuvent être relatives à la difficulté de mise en oeuvre (complexité intrinsèque, pas d'implémentation de services de « haut niveau ») qui implique l'absence d'utilisation (ou la mauvaise utilisation) des mécanismes de sécurité ;
- il peut s'agir de faiblesses intrinsèques à l'architecture (ou à la spécification) ;
- il peut également s'agir de vulnérabilités d'implémentation.

3.1 Mauvaise utilisation des mécanismes de sécurité

Certains problèmes de sécurité rencontrés dans les applications Java sont liés à une mauvaise utilisation des mécanismes de sécurité.

Absence d'utilisation du contrôle d'accès JPSA. En premier lieu, le contrôle d'accès JPSA est rarement utilisé, en particulier pour les applications autonomes « locales » du poste client (ce qui exclut les applets). Ceci s'explique en partie par le fait qu'aucun gestionnaire de sécurité n'est actif par défaut pour ce type d'applications. Or le rôle de celui-ci est important, notamment pour restreindre les privilèges du code non maîtrisé, provenant par exemple de bibliothèques tierces.

En outre, il est difficile pour un humain (un administrateur par exemple), d'appréhender la sémantique du contrôle effectué. En effet, chaque type de permission possède sa propre sémantique définie par la méthode `implies` qui permet de comparer les permissions entre elles. Cette méthode est notamment appelée lors du contrôle afin de s'assurer que les permissions attachées au code exécuté impliquent celles nécessaires pour réaliser l'accès. De plus, il n'existe pas de réelle hiérarchie entre les différents types de permissions.

En pratique, il existe peu d'outils permettant de spécifier la politique et de raisonner sur la sémantique des permissions. Ceux fournis dans l'implémentation de Sun sont largement insuffisants, car ils offrent peu de fonctionnalités supplémentaires par rapport à un simple éditeur de texte.

De manière générale, il s'agit d'un mécanisme complexe à appréhender, notamment pour les développeurs d'application Java. Spécifier une politique de contrôle efficace, qui fournisse à chaque classe les permissions minimales nécessaires à son bon fonctionnement est une tâche complexe et fastidieuse. Il est en outre difficile de s'assurer que la politique spécifiée n'est pas trop laxiste, notamment en raison des actions privilégiées. La bibliothèque standard comprend également un certain nombre de mécanismes qui constituent des exceptions implicites au contrôle d'accès. De plus, certaines permissions comme « accès à tous les fichiers » peuvent entraîner indirectement l'ensemble des permissions, le fichier décrivant les permissions devenant ainsi accessible.

Il est également complexe d'adapter le mécanisme (création de nouveaux points de contrôle, de nouvelles permissions associées, etc.). Or cela est parfois nécessaire, notamment lorsque l'application (ou la bibliothèque) Java comprend du code natif permettant d'accéder à des ressources de la plate-forme native. Il s'agit d'ailleurs d'une limitation importante du contrôle d'accès JPSA : les accès contrôlés doivent faire l'objet d'un point de contrôle spécifié explicitement au sein du code Java. Le développeur qui n'est pas conscient de cette limite peut utiliser une bibliothèque native permettant à du code Java d'accéder à des ressources natives (par exemple, le système de fichier) sans utiliser les classes de la bibliothque standard et en contournant ainsi le contrôle d'accès Java.

Les risques liés à la modification des éléments critiques. Les mécanismes de sécurité proposés par la bibliothèque standard, en particulier le contrôle d'accès, sont très modulaires. Ils s'appuient sur des classes Java dont beaucoup ne sont pas déclarées `final`. Un développeur Java peut donc modifier le comportement de ces mécanismes en redéfinissant ces classes via l'héritage. Ce type de modifications n'est pas conseillé dans la majorité des cas, mais peut parfois être nécessaire. Par exemple, le développeur peut être amené à modifier le gestionnaire de sécurité pour mettre en œuvre des formes de contrôle d'accès particulières (par exemple, afin de prendre en compte d'autres éléments que l'origine des classes dans les décisions de contrôle d'accès).

L'implémentation d'un chargeur de classe *ad hoc* constitue un cas beaucoup plus fréquent de modification d'un élément critique. Cela permet en effet de mettre en place une forme de cloisonnement entre différentes parties de l'application (cette technique est notamment utilisée dans les serveurs d'applications Web type JBoss [18] ou les conteneurs de servlet/JSP comme Tomcat [19]).

Un développeur d'applications ou de bibliothèques Java peut, en modifiant un mécanisme critique, introduire accidentellement des faiblesses dans le mécanisme. Ce risque dépend du niveau des modifications apportées. Il est renforcé par la relative complexité de l'architecture et des interactions entre les différents éléments. C'est notamment le cas du contrôle d'accès qui fait appel à différentes classes dont certaines ne sont pas dédiées à cette fonction. Le développeur qui implémente son propre chargeur de classes n'a pas forcément conscience que cet élément est essentiel au bon fonctionnement du contrôle d'accès. En effet, il participe à l'identification de l'origine des classes en leur associant un domaine de sécurité adéquat. Cette opération doit être spécifiée explicitement dans le code du chargeur de classes. Un développeur peut omettre cette fonctionnalité qui ne remet pas en cause le fonctionnement du chargeur de classes. Cela rend toutefois le contrôle d'accès inopérent pour les classes chargées par ce chargeur de classes, qui possèdent alors toutes les permissions.

Utilisation des mécanismes cryprographiques. L'architecture cryptographique de Java fournit un niveau d'abstraction intéressant pour le développeur.

[18] `http://www.jboss.org/`
[19] `http://tomcat.apache.org/`

Ceci facilite grandement l'utilisation des services cryptographiques par le biais d'une API haut-niveau. Toutefois, il est important d'avoir en tête certaines subtilités afin de mieux contrôler l'exécution de l'application et d'éviter les éventuels piégeages logiciels.

L'instanciation d'une implémentation (par exemple, AES) d'un service cryptographique (par exemple, le chiffrement) est réalisée grâce à l'utilisation de la méthode `getInstance` de la classe de haut-niveau associé au service. Dans l'exemple suivant, il s'agit de la classe `Cipher` :

```
Cipher c = Cipher.getInstance("AES");
Cipher c = Cipher.getInstance("AES", "PROVIDER");
```

La première manière d'instancier le service ne spécifie pas le *provider* cryptographique. Par conséquent, les *providers* enregistrés par JCE sont parcourus et la première d'implémentation d'AES trouvée sera utilisée. Dans un contexte où le choix de l'implémentation est critique (risque de piégeage, efficacité des algorithmes, confiance dans les implémentations, etc.), il est nécessaire de prendre certaines dispositions. Dans un premier temps, le développeur doit utiliser la seconde méthode d'instanciation qui permet d'imposer l'utilisation d'un *provider* donné. Dans un second temps, si le contexte le permet, il est possible de configurer par ordre de préférence la liste des *providers* chargés au démarrage. Cela se fait en configurant le fichier `<JAVA_HOME>/lib/security/java.security`. Ainsi, il est possible de retirer et d'ajouter les *providers* adéquats.

Enfin, nous avons constaté qu'il est primordial de bien connaître les API et de spécifier correctement les paramètres des algorithmes cryptographiques utilisés. A l'instar de la méthode `getInstance`, d'autres méthodes peuvent être appelées avec plus ou moins de paramètres. Certaines versions impliquent des choix réalisés par la bibliothèque standard. Cela peut conduire à des résultats de fonctionnement non désirés par le développeur. Par exemple, nous avons constaté que le générateur de clés DSA du *provider* de SUN utilise un cache d'éléments pré-calculés pour les paramètres p, q et g de la clé :

```
KeyPairGenerator gen;
KeyPair keys;
gen = KeyPairGenerator.getInstance ("DSA", "SUN");
gen.initialize(512, new SecureRandom());
keys = gen.generateKeyPair ();
```

Ainsi, à chaque génération de clé DSA, le même triplet p, q et g est systématiquement utilisé pour la génération des clés. Pour éviter ce piège, le développeur doit utiliser une méthode moins triviale pour générer une clé DSA :

```
KeyPairGenerator gen;
KeyPair keys;
DSAParameterSpec spec;
gen = KeyPairGenerator.getInstance ("DSA", "SUN");
spec = ParameterCache.getNewDSAParameterSpec (1024 ,
  new SecureRandom());
gen.initialize(spec);
keys = gen. generateKeyPair ();
```

3.2 Faiblesses intrinsèques du langage ou de la bibliothèque standard

Des mécanismes « dangereux ». Plusieurs vulnérabilités auxquelles sont sensibles les applications et les bibliothèques Java (notamment la bibliothèque standard) proviennent de mécanismes « dangereux » dont le fonctionnement peut remettre en cause certaines propriétés de sécurité. Ces mécanismes nécessitent, de la part du développeur, une bonne connaissance de leur fonctionnement pour éviter tout problème de sécurité.

Dépassement de capacité des entiers. Tout d'abord, le dépassement de capacité sur les entiers est silencieux en Java : aucune exception n'est levée. Un débordement ne peut *a priori* pas donner directement lieu à une vulnérabilité, car Java utilise une arithmétique modulaire. Par contre, cela peut indirectement donner lieu à une faille de sécurité en combinant ce problème à l'utilisation de méthodes natives utilisant des tampons dont la taille n'est pas contrôlée. Des vulnérabilités de ce type ont notamment été découvertes dans l'implémentation de Sun de la bibliothèque standard [20]. Dans le même ordre d'idée, on peut également noter qu'une division entière par 0 donnera lieu à une exception, alors que du côté des flottants, aucune exception n'est levée.

Sérialisation. La sérialisation est un mécanisme qui permet de créer une représentation binaire enregistrable dans un fichier ou transférable à travers le réseau, à partir d'un objet Java. L'opération inverse, la désérialisation, permet de reconstruire cet objet Java à partir de la représentation binaire. Le mécanisme de sérialisation de Java souffre de plusieurs défauts :

- la sérialisation peut donner accès à des données confidentielles au travers de la représentation binaire. Pour contrer ce problème, il suffit d'empêcher les données confidentielles d'être sérialisées en redéfinissant la méthode de sérialisation des objets contenant ces données confidentielles. Il est également possible d'exclure certains champs de classes en y apposant le modificateur `transient`. Ce modificateur empêche que le champ concerné soit sérialisé par le mécanisme de sérialisation par défaut ;
- il est également possible en ayant accès à la représentation binaire d'une classe de modifier celle-ci afin de briser des invariants de classe. Par exemple, dans [13], l'auteur présente un exemple qui montre la manière de briser l'immuabilité d'une classe en utilisant le mécanisme de sérialisation/désérialisation. Ce problème peut être résolu en traitant la désérialisation de la même manière qu'un constructeur traiterait ses entrées ou en protégeant la représentation binaire (par exemple, à l'aide de mécanismes cryptographiques).
- une troisième problématique concerne le principe de fonctionnement du mécanisme de désérialisation lui-même. Cette problématique a d'ailleurs été exploitée par la faille *Calendar* [7]. La désérialisation d'une donnée se fait de la manière suivante :

[20] `http://sunsolve.sun.com/search/document.do?assetkey=1-66-270474-1`

```
A a = (A) flow.readObject();
```

ce qui revient en fait à réaliser les deux instructions suivantes :

```
Object obj = flow.readObject();
A a = (A) obj;  // Une exception est levee si obj n'est
    //  pas une instance de A
```

La désérialisation ne peut contrôler la classe réellement désérialisée qu'*a posteriori*. En effet, le mécanisme de désérialisation tel qu'il est conçu est prévu pour renvoyer un objet quelconque dont le type est déterminé lors de la lecture de la représentation binaire. Après coup, l'objet retourné est transtypé dans le type que l'on pense récupérer, mais l'objet peut être de type différent et une exception est alors levée. Le problème est que le code de la méthode `readObject()` de la classe désérialisée a été exécuté et ainsi a pu effectuer potentiellement des opérations malicieuses. Dans le cas de la faille *Calendar* [14], l'objet désérialisé est un `ClassLoader` qui permet de charger n'importe quelle classe sans tenir compte des permissions d'accès aux ressources du système. L'instanciation de la classe `ClassLoader` est normalement protégée, mais, pour cette faille, le code d'exploitation utilise une classe où la désérialisation est effectuée dans un contexte privilégié dans lequel toutes les permissions sont données au code exécuté.

Réflexion. La réflexion est la capacité d'un programme à examiner son propre état (introspection) et à modifier son propre état d'exécution ou d'altérer sa propre interprétation ou signification (intercession). Java fournit une API pour analyser une classe (listes des champs, méthodes, constructeurs), un champ (type, modificateurs associés), ou une méthode/constructeur (signature, type de retour, modificateurs associés). Java permet également, au travers de cette API, de lire et modifier des valeurs de champs, d'instancier des classes ou d'invoquer des méthodes. Le problème de ce mécanisme est qu'il permet de contourner les informations de visibilité d'une classe, champ, constructeur ou méthode. En Java, le modificateur `private` indique que l'élément concerné n'est accessible qu'au sein de sa propre classe. Or, en utilisant la réflexion, il devient possible d'accéder à (voire de modifier) des informations confidentielles. La réflexion est par exemple utilisée par le mécanisme de sérialisation par défaut pour récupérer la valeur des champs à sérialiser. La figure 5 montre un exemple commenté de son utilisation.

Toutefois, l'accès au mécanisme de contournement de la visibilité est limité aux classes disposant des permissions adéquates. Par défaut, cela concerne uniquement les classes de la bibliothèque standard. Cependant, il faut prendre garde à mettre en place un gestionnaire de sécurité (ou *security manager*) et à bien configurer la politique de sécurité, car en l'absence de gestionnaire de sécurité, toutes les permissions sont accordées et l'ensemble des méthodes peuvent ainsi contourner la visibilité.

Confiance dans le code de la bibliothèque standard. La plupart des vulnérabilités publiques recensées sur les JRE (notamment, celles concernant l'im-

```
private static ObjectStreamField[]
                getDeclaredSerialFields(Class cl)
                                throws InvalidClassException {
  ObjectStreamField[] serialPersistentFields = null;
  try {
    // permet de recuperer le champ serialPersistentFields
    // de la classe cl si celui-ci existe.
    Field f = cl.getDeclaredField("serialPersistentFields");
    int mask = Modifier.PRIVATE | Modifier.STATIC
                    | Modifier.FINAL;
    // permet de recuperer les modificateurs du champ
    if ((f.getModifiers() & mask) == mask) {
      // permet de contourner la visibilite private du champ
      f.setAccessible(true);
      // recupere la valeur du champ qui est
      // normalement non accessible
      serialPersistentFields=(ObjectStreamField[])f.get(null);
    }
  } catch (Exception ex) {}
  ...
}
```

Fig. 5. Extrait de la classe `java.io.ObjectStreamClass` commentée

plémentation de SUN) sont relatives à la bibliothèque standard. Cette dernière comprend en effet une quantité de code importante susceptible de comporter des vulnérabilités. À titre d'exemple, la bibliothèque standard d'OpenJDK de Sun contient environ 1 900 000 lignes de code (dont les trois quarts sont en Java et le reste en C et C++). En comparaison, l'implémentation de la JVM HotSpot contient environ 450 000 lignes de code (essentiellement du C++). Nous pouvons constater que les vulnérabilités proviennent généralement du code natif de la bibliothèque mais également du code Java (c'est le cas par exemple de la faille *Calendar*). Se pose donc le problème de la confiance en cet élément essentiel du JRE.

En outre, les éléments de la bibliothèque sont très hétérogènes, en termes de fonctionnalités et de qualité d'implémentation. En effet, les éléments essentiels au langage (chaînes de caractères, gestion des *threads*, etc.) ou implémentant des fonctions de sécurité (contrôle d'accès, mécanismes cryptographiques, etc.) côtoient des fonctionnalités plutôt « anecdotiques » (gestion des calendriers, des polices de caractères, etc.). En pratique, pour l'implémentation de Sun, la qualité d'implémentation diffère également sensiblement d'un élément à un autre. Si les fonctions essentielles ou de sécurité ont visiblement fait l'objet de développements au sein de Sun, certaines fonctionnalités « anecdotiques » ont été développées par d'autres sociétés. Le code de ces fonctionnalités n'a pas toujours fait l'objet de bonnes pratiques en termes de développement ni d'audit

de sécurité. En pratique, les vulnérabilités concernent généralement ce type de fonctionnalités.

Cela est d'autant plus problématique qu'un même niveau de confiance est accordé à l'ensemble de la bibliothèque. Bien souvent (c'est le cas par défaut pour HotSpot) le code Java de la bibliothèque standard ne fait pas l'objet d'une vérification de *bytecode*. En outre, le code de la bibliothèque est réputé « de confiance » par le mécanisme de contrôle d'accès qui lui accorde l'ensemble des permissions. Une vulnérabilité présente dans le code de la bibliothèque standard, même s'il concerne une fonctionnalité « anecdotique », aura donc potentiellement un impact important. Certaines vulnérabilités, telle la faille *Calendar*, permettent par exemple une élévation de privilèges.

La bibliothèque standard étant un élément indispensable, elle constitue donc une surface d'attaque importante. De plus, du fait de l'absence de standardisation de certains éléments, l'implémentation de Sun (dont est dérivée celle d'OpenJDK) constitue une implémentation de référence quasi incontournable. Les autres implémentations open source (notamment GNU Classpath) n'offrent pas un niveau fonctionnel équivalent. Certaines implémentations propriétaires concurrentes réutilisent le code de Sun (c'est notamment le cas de Jrockit [21]). Ceci tend à l'uniformisation, ce qui augmente d'autant l'importance des vulnérabilités.

Des interfaces critiques. Le JRE, et la JVM en particulier, comporte des interfaces qui peuvent mettre en péril la sécurité des applications Java.

JNI. JNI offre des fonctionnalités intéressantes et nécessaires au bon fonctionnement de la plate-forme d'exécution. Cette interface permet notamment aux méthodes natives des classes de la bibliothèque d'accéder à la plate-forme native. Toutefois, il s'agit de la principale déviation de Java qui est la cause d'un grand nombre de vulnérabilités des applications et des environnements d'exécution Java. En effet, la plupart des propriétés de Java ne sont plus vérifiées pour le code natif. Ce dernier souffre des problèmes intrinsèques liés à l'utilisation de langages tels que C ou C++ (gestion de la mémoire explicite, absence de vérifications dynamiques, etc.). À l'inverse du code Java, il est vulnérable aux dépassements de tampon, aux erreurs de formatage de chaîne de caractères, etc.

Le risque est d'autant plus important que le code natif implémente des opérations complexes. En cas d'exploitation des éventuelles vulnérabilités, l'attaquant dispose des mêmes privilèges que la JVM (et l'application qu'elle exécute). Si différentes instances de la JVM ne sont pas cloisonnées (par exemple, exécution sous la même identité au sens de l'OS), l'attaquant peut accéder à l'espace mémoire et aux données des autres instances (et donc d'autres applications Java).

La manipulation de données sans vérification au préalable constitue le risque majeur du code natif appelé via JNI. Il existe notamment un risque que des

[21] `http://download.oracle.com/docs/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/aboutjrockit.html`

données acquises par le code Java et transmises au code natif permettent d'exploiter une vulnérabilité de ce dernier. C'est notamment le cas des vulnérabilités de type débordement de tampon, car le passage d'objets de type référence depuis le code Java vers le code natif nécessite une copie pour la conversion entre les types Java et les types natifs.

Pour ces raisons, il est recommandé de n'utiliser JNI que dans les cas qui le nécessitent explicitement (typiquement, l'accès à une ressource native pour laquelle il n'existe pas d'interface avec Java).

Interfaces de mise au point. La plate-forme Java propose plusieurs interfaces de mise au point et d'audit dynamique (JVMTI, JPDA, JMX, etc.). Celles-ci fonctionnement généralement à base d'agents développés en Java ou C/C++. Elles permettent à ces agents, voire à des entités distantes, de contrôler l'exécution d'une application Java. L'interface d'instrumentation à la volée (`java.lang.instrument`) permet même de modifier le code de l'application. Ces interfaces sont certes utiles en phase de développement et de mise au point, mais elles constituent potentiellement un danger en phase de production et devraient alors être désactivées.

Limite du mécanisme de cloisonnement. L'utilisation des chargeurs de classes permet en théorie de cloisonner certains éléments. En pratique, il s'agit simplement de mettre en œuvre des espaces de nommage distincts. Le risque majeur consiste à mettre en œuvre un cloisonnement partiel, qui peut être contourné. La complexité du mécanisme renforce la difficulté de mise en place d'un cloisonnement strict des classes. En effet, il est possible d'échanger de l'information avec les types visibles depuis les différents domaines (notamment, les classes définies par le chargeur initial). Les mécanismes de sérialisation ou d'appel de méthodes à distance (RMI) peuvent également constituer des mécanismes d'échappement. À l'inverse, il n'est pas non plus aisé de définir un mécanisme d'échange souple et paramétrable entre les différents éléments cloisonnés.

Des projets ont été initiés pour inclure dans la spécification de la plate-forme Java un mécanisme dédié au confinement de différentes applications Java s'exécutant sur une unique instance de la JVM. C'est notamment le cas de l'API d'isolation Java (*Java isolate API*) qui a fait l'objet d'une démarche de spécification (JSR 121[22]). Il s'agit de pouvoir exécuter différentes applications sur une même plate-forme Java (appelée *aggregate*), au sein d'environnements confinés (appelés *isolates*). Par défaut, les environnements confinés partagent peu de données Java (ils disposent de leur propre tas), mais ils s'appuient en revanche sur les mêmes services offerts par la JVM. La plate-forme repose également sur une extension de l'API de la bibliothèque standard Java permettant d'implémenter des moyens de communication entre les domaines confinés. Ces échanges peuvent être contrôlés par un mécanisme adéquat reposant sur la définition d'une politique de confinement. Toutefois, la spécification ne précise pas les contraintes qui doivent s'appliquer sur le partage des ressources. De plus, elle ne précise pas la

[22] `http://jcp.org/jsr/detail/121.jsp`

stratégie d'implémentation (même si la plupart des implémentations s'appuient sur une instance unique de la JVM s'exécutant sur un ou plusieurs processus OS).

À l'heure actuelle, cette spécification n'a pas été intégrée à la plate-forme standard. Initialement, il était prévu de l'intégrer à la version 1.5 du JRE, mais cette intégration a été successivement reportée et elle ne semble pas à l'ordre du jour pour la future version standard (1.7). De fait, il n'existe que des implémentations expérimentales, notamment au sein du projet de recherche Joe [23] (ex Barcelona) de Sun.

3.3 Faiblesses de la JVM

Certains problèmes de sécurité des applications Java s'expliquent également par des faiblesses dans la spécification et l'implémentation de la JVM.

Faiblesses de la gestion de la mémoire. Paradoxalement, l'une des premières faiblesses de la JVM est l'impossibilité, pour le programmeur d'application Java, de gérer finement la mémoire qu'il utilise. La gestion automatique de la mémoire est souvent considérée comme un avantage certain de Java. En effet, le développeur n'a pas à s'occuper de cette partie qui conduit généralement, dans les programmes développés à partir de langage ne disposant pas d'une telle fonctionnalité (notamment C et C++), à des bogues ou, dans le pire des cas, à des failles de sécurité (dénis de service par fuite mémoire ou exécution de code par le biais d'un débordement de tampon).

Toutefois, la gestion de la mémoire pose problème lorsqu'il est nécessaire d'assurer l'effacement des données confidentielles stockées en mémoire vive. Certes, il s'agit d'une problématique particulière, mais qui se révéle importante dans certains cas, notamment lorsque l'application doit manipuler des secrets cryptographiques : typiquement, des applications comprenant des opérations de chiffrement/signature (applications bancaires, gouvernementales, etc.). Les bonnes pratiques recommandent de limiter la durée de vie en mémoire vive de ces données confidentielles. Il s'agit de limiter les problématiques de rémanence (et de dissémination).

Les JVM classiques n'implémentent pas de procédures permettant de réaliser l'effacement d'un objet lors de sa libération en mémoire. Actuellement, pour répondre à cette problématique, il est nécessaire pour le développeur d'implémenter des fonctions *ad hoc* réalisant cette tâche. Cette tâche n'est pas toujours aisée. S'il est en théorie possible d'effacer par une mise à zéro les champs publics et les variables locales de type primitif, il peut s'avérer impossible d'effacer certains champs privés ou des objets immuables (par exemple, les chaînes de caractères). Selon le degré de visibilité des objets manipulés, les primitives d'effacement n'auront donc pas forcément accès aux informations sensibles. De plus, en raison des optimisations qui peuvent être réalisées par le compilateur JIT (ou

[23] `http://research.sun.com/projects/joe/`

AOT), le développeur n'a en fait aucune garantie sur la réalisation effective de l'opération d'effacement. En effet, une des optimisations classique dans le domaine de la compilation consiste à supprimer les affectations qui ne sont pas suivies d'une utilisation, ce qui correspond typiquement à une opération d'effacement par mise à zéro.

Parmi les solutions généralement implémentées par les développeurs, on peut citer la surchage de la méthode `finalize`. Cette méthode est appelée par le *garbage collector* avant la libération de la mémoire d'un objet. Toutefois, il n'est pas possible de prévoir à quel instant le processus de collecte est lancé. Cela varie selon la JVM et le paramétrage utilisé. Généralement, le *garbage collector* intervient aux moments suivants :

- lorsque l'allocateur ne trouve plus de blocs de mémoire libres ;
- lorsqu'à l'issue d'une allocation, le taux d'occupation du tas atteint un seuil donné ;
- à intervalle régulier (par exemple, JamVM avec les options adéquates, dispose d'un *garbage collector* qui est lancé à intervalle régulier).

Enfin, il est nécessaire de souligner le problème que constitue les *garbage collector* générationnels, dont le fonctionnement est décrit en annexe A.3), dans l'effacement de la mémoire. L'intérêt de ce type de *garbage collector* est de réaliser des cycles de collecte sur des « générations » ciblées et non sur l'ensemble du tas. Ceci permet notamment de réduire les temps de blocage de la JVM et d'augmenter l'interactivité (au détriment de la bande passante). Toutefois, le *garbage collector* générationnel contribue à dupliquer dans la mémoire des informations contenues dans les objets. En effet, le mécanisme de promotion des objets se traduit par des copies du contenu de la mémoire d'une « génération » à l'autre. Nous avons par exemple constaté que pour HotSpot, la JVM d'OpenJDK, la libération de la mémoire dans la génération « source » de la copie est réalisée par le biais de manipulations de pointeurs. La mémoire libérée ne fait pas l'objet d'une mise à zéro ou d'une procédure d'effacement sécurisée. Par conséquent, des informations sensibles peuvent se voir dupliquées dans la mémoire de l'application. Ceci complexifie d'autant la procédure d'effacement. Quand bien même le développeur Java disposerait de primitives lui garantissant l'effacement de la zone mémoire pointée par une référence, il ne peut avoir la garantie que cette zone n'a pas fait l'objet d'une copie par le GC.

Échappement à la vérification de *bytecode*. Selon la spécification de la JVM [22], l'ensemble des classes et interfaces Java doivent être vérifiées avant leur initialisation. Cependant, en pratique, certaines JVM ne disposent pas de vérificateur de *bytecode* (par exemple JamVM) et, pour d'autres JVM (notamment HotSpot) certaines classes peuvent échapper à la vérification de *bytecode*.

En effet, selon l'implémentation d'OpenJDK6 et la documentation de la bibliothèque standard, les classes héritant de la classe `sun.reflect.MagicAccessorImpl` ne sont pas vérifiées par le vérificateur de *bytecode*. En l'absence de gestionnaire de sécurité ou suite à une mauvaise configuration des permissions d'accès, il peut être possible de créer des classes qui ne seront pas vérifiées et

dont le *bytecode* peut effectuer des opérations contraires à la sémantique de Java (comme l'utilisation de l'arithmétique et de la forge de pointeurs).

Complexification de la JVM. Les modes d'exécution de *bytecode* au sein des implémentations de la JVM, décrits en annexe A.4, apparaissent de plus en plus complexes : interprétation « basique », de type *threaded*, *inline* ou *template* ; compilation de type JIT (*Just In Time*) ou AOT (*Ahead Of Time*) ; compilation dynamique. En conséquence, il est de plus en plus difficile d'évaluer de manière approfondie les implémentations courantes de la JVM. Cela serait pourtant nécessaire avant d'employer l'une d'entre elles dans un environnement critique.

Pour HotSpot, la complexité est également liée aux multiples choix en termes de modes d'exécution. En effet, HotSpot propose deux interpréteurs (mode *inline* et *template*) et deux types de compilateur JIT ou cibles d'exécution (client et serveur), ayant chacune des optimisations qui leur sont propres.

HotSpot propose en outre d'utiliser conjointement ces différents modes d'exécution (compilation dynamique) et de sélectionner le mode le plus adéquat en fonction du *profiling* du code, réalisé lui aussi durant l'exécution. En conséquence, l'exécution de *bytecode* sous HotSpot se traduit par un enchevêtrement de phases d'interprétation et de phases d'exécution de code compilé. La coordination de ces mécanismes nécessite la mise en œuvre d'algorithmes complexes permettant de faire le lien entre les modes interprétés et compilés.

Certaines optimisations ont également un impact sur le fonctionnement globale de la plate-forme d'exécution, soit parce qu'elles font appel à des mécanismes sensibles (la gestion de la synchronisation par exemple), soit parce qu'elles rendent plus difficile la mise en œuvre de mécanismes de sécurité internes. À titre illustratif, voici quelques exemples, détaillés en annexe A.4, relatifs à HotSpot :

- OSR (*On Stack Replacement*) : ce mécanisme complexe nécessite l'intervention de nombreux points de synchronisation entre les *threads* dédiés à la compilation et les *threads* Java.
- Escape Analysis : cette optimisation illustre la difficulté d'analyse de l'implémentation du tas Java, qui peut ainsi être partiellement implémenté sur la pile native. Par ailleurs, le fait d'allouer des objets sur la pile au lieu du tas peut compliquer la réalisation d'un effacement sécurisé réalisé par la JVM ou la mise en place du chiffrement du tas Java.

Pour les compilateurs JIT, beaucoup de manipulations et de transformations sur le code sont réalisées entre l'étape du vérificateur de *bytecode* et l'exécution de code proprement dite. Ces transformations ont notamment lieu lors de la génération de la représentation intermédiaire de code. Certaines implémentations, comme OpenJDK HotSpot en mode client, génèrent même deux représentations intermédiaires.

En outre, les optimisations appliquées sur le *bytecode*, la représentation intermédiaire ou sur le code machine, visent généralement à retirer certaines vérifications, telles que celles faites sur les bornes des tableaux ou sur les références nulles. Ces retraits sont censés être réalisés de manière correcte afin de ne pas

compromettre la sécurité. Toutefois, il est difficile de valider le bon fonctionnement de ces algorithmes d'optimisations sans une étude approfondie.

Ainsi, il n'est donc pas certain que les propriétés de sécurité vérifiées par le vérificateur de *bytecode* soient maintenues jusqu'à l'exécution après de multiples modifications de code et autres optimisations.

Faible utilisation des mécanismes de protection de l'OS. Les différents modes d'exécution (interprétation *switch*, *threaded*, *inline*, *template* ; compilation JIT) incorporent de manière progressive la notion de compilation à la volée. Or, ce type de compilation implique deux aspects complémentaires : la transformation puis l'exécution de code. En conséquence, cetains modes d'exécution de *bytecode* apparaissent incompatibles avec les mécanismes avancés de sécurité de l'OS. En particulier, la protection PaX sous Linux empêche le fonctionnement du JIT et de l'interpréteur dans les modes *inline* et *template*. Il en va de même pour l'optimisation visant à déplier les appels de méthode (*inlining*). En effet, tous ces modes d'exécution impliquent une modification à la volée du code exécutable. Or, la restriction MPROTECT de PaX empêche précisément d'avoir des pages mémoires anonymes avec les bits X (exécution) et W (écriture) activés en même temps. Ainsi, la protection PaX ne permet pas d'exécuter les JVM HotSpot (y compris en mode interpréteur, qui est de type *template*) et cacaovm. Sous JamVM, seule la désactivation du mode *inline* lors de la compilation (option *-disable-int-inline*) permet un fonctionnement compatible avec PaX.

D'autre part, l'étude des JVM a montré une faible utilisation de la programmation sécurisée. En particulier, le verrouillage des pages mémoire, qu'il est possible de mettre en œuvre avec les fonctions systèmes MLOCK() et MLOCKALL(), n'est pas utilisé, de même que les primitives système de la famille des SETUID et SETGID. La protection contre l'écriture dans le fichier d'échange (*swap*) pourrait en effet permettre d'obtenir une JVM renforcée (typiquement, pour se protéger contre les attaques visant la mémoire de masse). Le processus de compilation du code source de ces trois JVM ne fait également pas appel aux options de protection de la pile. Nous avons pu néanmoins obtenir une version stable en recompilant chaque JVM avec l'option -FSTACK-PROTECTOR-ALL de GCC et G++.

L'absence de prise des droits en exécution (bit x sous UNIX) pour les classes Java constitue également une limite. Concrètement, pour interdire l'exécution d'une application Java, il faut interdire la lecture de ses fichiers classes ou l'exécution de la commande `java`. Cette restriction ne permet pas de mettre en œuvre des politiques de type W^X : plus précisément il n'est pas possible de permettre la modification des classes Java sans en permettre l'exécution. Ce type de problématique est présente de manière générale pour les langages interprétés. Toutefois, les particularités de Java (notamment la présence de différents chargeurs de classes) complexifient la prise en compte de ce droit d'exécution. Il ne suffit pas de modifier la commande `java`, mais il faut également s'assurer que tous les chargeurs de classes implémentent cette vérification. Le problème

se pose également pour les contrôles d'accès mandataires comme SELinux qui reposent sur le chemin ou le label de sécurité associé à l'exécutable.

Absence d'un moniteur de référence. Une des principales faiblesses du contrôle d'accès JPSA réside dans l'absence d'un moniteur de référence clairement défini dans l'architecture et implémenté dans la JVM. Le contrôle repose sur différents éléments disséminés dans la bibliothèque standard, le rôle de la JVM étant minimal (elle fournit simplement l'ensemble des domaines de protection relatives à la pile d'exécution).

À la différence des mécanismes de contrôle d'accès implémentés dans les OS, le contrôle d'accès de Java n'est pas protégé par une interface de médiation clairement définie (comme le propose le mode noyau pour les OS). Il s'exécute dans le même contexte que le code qu'il est censé surveiller. De plus, il repose sur des points de contrôle qui doivent être définis explicitement.

Un attaquant ou une application malveillante peut donc modifier intentionnellement les mécanismes de contôle d'accès (par exemple, les désactiver). Il est donc nécessaire, pour se prémunir de ce risque, de protéger le mécanisme. Cela nécessite à la fois de recourir à des mécanismes fournis par l'OS (par exemple, pour protéger les fichiers de configuration) et au mécanisme de contrôle d'accès Java lui-même (par exemple, pour interdir la modification du gestionnaire de sécurité) qui doit donc mettre en oeuvre une forme d'auto-protection. La mise en place de ces protections peut s'avérer fastidieuse (notamment du fait de l'absence de notion d'interdictions) et nécessite que l'administrateur maîtrise le système de contrôle d'accès de Java.

4 Le renforcement de la sécurité de la plate-forme d'exécution Java

Renforcer la sécurité des applications Java implique deux niveaux de confiance :
- confiance dans le code applicatif en lui-même (logique applicative) ;
- confiance dans la plate-forme d'exécution.

Cet article aborde essentiellement le deuxième point. Le premier ne doit cependant pas être négligé, surtout en ce qui concerne les propriétés de sécurité non assurées par Java (cf. SQL injection). Le projet JAVASEC a permis de spécifier un ensemble de règles et recommandations à destination du développeur d'applications Java SE, regroupées sous forme de fiches adressant des problématiques particulières. Ce document s'inspire de travaux existants qu'il complète, notamment le guide du NIST [24] ou celui founi par Sun [25].

Renforcer la sécurité de la plate-forme d'exécution implique la mise en œuvre de techniques plus ou moins coûteuses. Celles-ci permettent de renforcer la sécurité (ou la confiance dans les mécanismes) de la plate-forme d'exécution Java

[24] `https://www.securecoding.cert.org/confluence/display/java/The+CERT+Sun+Microsystems+Secure+Coding+Standard+for+Java`

[25] `http://java.sun.com/security/seccodeguide.html`

à différents niveaux. Il s'agit d'un compromis entre le gain en sécurité (ou en confiance) et la prise en compte d'un certain nombre de problématiques :

- la compatibilité ascendante et descendante ;
- les performances ;
- l'impact global sur l'architecture existante ;
- le coût de mise en œuvre (développement et maintenance de patchs spécifiques, audit de code, etc.).

Le compromis dépend des besoins de l'utilisateur final qui exécute les applications Java. Pour la plupart des utilisateurs, la protection contre les vulnérabilités présentes dans l'implémentation de la bibliothèque standard est suffisante (le nombre d'exploits publics récents contre la JVM étant relativement faible). Pour ces utilisateurs, il est de plus impératif d'assurer la compatibilité avec les applications Java existantes. Les solutions proposées doivent également avoir le moins d'impact possible sur les performances, d'autant que Java n'est pas réputé (parfois à tort) pour être particulièrement efficace en la matière.

Il existe cependant des cas où les besoins en sécurité sont plus forts, notamment dans un contexte gouvernemental ou militaire. Certaines utilisations nécessitent une maîtrise fine du bon fonctionnement des services de la JVM, des besoins de durcissement particuliers, etc. Ces besoins justifient des impacts plus lourds sur les performances et les coûts en général.

Deux approches complémentaires peuvent être suivies :

- renforcer l'efficacité et la robustesse des mécanismes existants de manière transparente pour l'utilisateur ;
- proposer de nouveaux services qui nécessitent un effort de la part du développeur (développement spécifique ou modification du code source existant) ou de l'utilisateur (paramétrage).

Ces modifications portent sur les deux éléments principaux du JRE : la bibliothèque standard et la JVM.

4.1 Renforcement de la bibliothèque standard

La bibliothèque standard constitue un des éléments les plus vulnérables du JRE, en témoignent les nombreuses vulnérabilités publiques qui le concernent. Il s'agit donc d'un élément dont la sécurité doit être renforcée en priorité. Différentes approches peuvent être envisagées.

Audit de sécurité. La première approche consiste à analyser le code de la bibliothèque standard afin d'identifier les vulnérabilités. Cette tâche est coûteuse et fastidieuse. Bien qu'il s'agisse d'une technique intrinsèquement limitée, elle a permis en pratique de mettre à jour nombre de vulnérabilités. Les outils d'analyse statique, comme FindBugs, permettent d'automatiser cette tâche. Toutefois, ils ne permettent de découvrir que des vulnérabilités correspondant à un motif connu.

Cette recherche de motifs de vulnérabilités ou *security anti-patterns* [26] constitue une technique efficace pour trouver des vulnérabilités dans le code Java. Le principe consiste à identifier les méthodes susceptibles de comporter des vulnérabilités, en s'aidant éventuellement d'outils automatiques, puis de vérifier « manuellement » la présence effective des vulnérabilités. Les motifs suivants sont notamment recherchés : utilisation de JNI avec des tampons de taille variable, problématique des débordements d'entiers, utilisation de la désérialisation, utilisation de contextes privilégiés de contrôle d'accès, etc.

Limitation de la surface d'attaque potentielle. La deuxième approche consiste à réduire la surface d'attaque potentielle que constitue la bibliothèque standard. Pour cela, deux solutions sont envisageables. La première consiste, lorsque l'environnement est maîtrisé (l'ensemble des applications Java susceptibles d'être exécutées est connu) à déployer un sous-ensemble de la bibliothèque standard. Les classes inutiles (qui ne sont pas utilisées par les applications) sont supprimées, ce qui limite *in fine* la surface d'attaque. Cette approche nécessite à l'heure actuelle de modifier l'archive `rt.jar`. De plus, il est nécessaire de prendre en compte les dépendances. Cette tâche devrait être simplifiée avec l'introduction des modules Java proposée par la JSR 294 et prévue pour la version 1.7 de Java grâce au système Jigsaw.

La deuxième solution consiste à réduire les privilèges des classes de la bibliothèque standard en fonction de leurs besoins. Cette approche est déjà utilisée ponctuellement de manière *ad hoc* pour résoudre certains problèmes de sécurité (elle est notamment utilisée pour limiter l'impact de la faille *Calendar*). Il paraît nécessaire d'en généraliser l'usage à l'ensemble du code de la bibliothèque standard, de manière à limiter l'impact des vulnérabilités. Cela nécessite cependant de modifier le code du contrôle d'accès Java qui considère par défaut que l'ensemble du code de la bibliothèque Java possède toutes les permissions.

Utilisation du contrôle d'accès. L'absence d'utilisation du mécanisme de contrôle d'accès s'explique en partie par l'absence d'outils adéquats permettant de spécifier la politique de contrôle. Marc Schönefeld a récemment mis à disposition Jchain [27], un outil qu'il a conçu et qui permet de générer automatiquement l'ensemble des permissions requises par une application Java. Bien qu'il s'agisse d'un prototype, il offre des fonctionnalités intéressantes, similaires au mode d'apprentissage des mécanismes de contrôle d'accès SELinux ou AppArmor. Il paraît souhaitable que le JRE inclue ce type d'outils et que les développeurs et les administrateurs soient formés à son utilisation.

4.2 Renforcement de la JVM

L'un des objectifs du projet JAVASEC consiste à proposer et mettre en œuvre des modifications permettant d'obtenir une JVM adaptée à l'exécution d'appli-

[26] `http://www.illegalaccess.org/presentation/bellua.jsp`
[27] `http://www.jchains.org/`

cations de sécurité dans un contexte gouvernemental. Différentes propositions ont été faites, certaines d'entre elles faisant actuellement l'objet d'une implémentation.

Augmenter la confiance dans le mécanisme d'exécution. Le renforcement de la JVM passe en premier lieu par l'augmentation de la confiance dans les mécanismes qu'elle implémente, notamment le mécanisme d'exécution de *bytecode*. Cela suppose un audit approfondi de ces mécanismes. Cette tâche délicate n'est pas facilitée par la complexité croissante des implémentations des JVM, notamment en ce qui concerne les optimisations pour l'exécution du *bytecode*.

Deux solutions sont envisageables. La première consiste à sélectionner, modifier ou ré-implémenter les mécanismes de manière à faciliter les vérifications et l'analyse du code. Cela consiste notamment à garantir que les propriétés de sécurité (notament le typage) soient conservées durant les différentes étapes de transformation et d'optimisation du code.

La deuxième solution consiste à choisir une JVM mettant en œuvre des mécanismes simples et à désactiver ou restreindre les optimisations. Cette solution radicale n'est pas adaptée à tous les contextes d'emploi, car elle a un fort impact sur les performances. Elle possède en outre l'avantage de permettre l'activation des mécanismes de protection comme PaX sous Linux. Une variante moins radicale consiste à paramétrer le niveau d'optimisation suivant la méthode Java exécutée. Le développeur (ou l'administrateur) doit alors identifier les méthodes (ou les classes) dont l'exécution doit faire l'objet de restrictions en ce qui concerne les optimisations réalisées. Différents critères peuvent motiver une restriction :

- l'exécution d'un algorithme implémentant une fonction critique ;
- l'exécution de *bytecode* manipulant des tableaux ou des objets susceptibles de réaliser des dépassements en mémoire (les optimisations pouvant retirer certains contrôles de dépassement) ;
- l'exécution de *bytecode* qui n'est pas de confiance (COTS), etc.

Différentes stratégies d'implémentation peuvent être envisagées, notamment l'utilisation d'un fichier de configuration ou l'utilisation des annotations.

Permettre aux programmeurs un contrôle fin sur la durée de vie et la confidentialité des données sensibles. L'objectif de cette solution est de permettre aux développeurs des applications Java d'avoir une plus grande maîtrise sur la mémoire des objets afin de pouvoir réaliser des effacements de manière sécurisée. La solution répondant à cette problématique s'articule sur plusieurs mécanismes, chaque mécanisme pouvant être implémenté suivant différentes stratégies.

Le premier mécanisme intervenant dans la solution est le mécanisme d'identification des objets sensibles. L'objectif de ce mécanisme est de permettre au développeur d'indiquer à la JVM quels sont les objets sensibles qui feront l'objet d'un effacement sécurisé. Pour réaliser ce mécanisme, plusieurs pistes sont possibles :

- Utiliser les annotations Java pour marquer les classes dont les instances seront définies comme sensibles. L'avantage de cette implémentation est que les annotations peuvent être placées au niveau du code source ou directement sur le fichier `class` après la compilation.
- Utiliser un nouvel allocateur. Ce choix d'implémentation nécessite d'étendre le langage Java avec une nouvelle instruction qui spécifiera que les objets alloués sont définis comme sensibles.

Le second mécanisme intervenant dans la solution intervient dans la gestion des objets préalablement identifiés. Les choix possibles sont les suivants :

- Ajouter un attribut « sensible » à la classe `java.lang.Object` ;
- Utiliser un tas sécurisé différent du tas des objets « normaux ». Cela implique l'utilisation d'un *garbage collector* dédié à ce tas qui réalisera un effacement sécurisé lors de la désallocation des objets.

Enfin, le dernier mécanisme intervenant dans la solution est l'ajout d'une instruction d'effacement explicite d'un objet. En effet, la suppression de la référence vers un objet ne garantit pas son effacement immédiat. Il n'est donc pas possible de prévoir le moment où les cycles de collectes sont réalisés. Cela varie selon l'implémentation de la JVM et de son paramétrage. Par conséquent, il est nécessaire de fournir au développeur une instruction ou une méthode de l'API Java forçant la JVM à réaliser l'effacement d'un objet donné (sans pour autant réaliser l'étape de collecte).

Modifier le lanceur Java pour faciliter l'intégration avec les mécanismes de l'OS. Afin de pallier les limites concernant la prise en compte des droits d'exécution d'une application Java, il peut être envisagé de modifier le lanceur Java. Plus précisemment, deux solutions sont envisageables.

La première solution consiste à définir un lanceur dédié pour chaque application Java à l'aide de JNI. Cela facilite la mise en œuvre du contrôle d'accès au niveau de l'OS, car les applications correspondent à différents exécutables. Il est donc possible de leur associer des droits ou des labels de sécurité distincts. Lorsqu'il est nécessaire de restreindre les droits en exécution, il faut, de plus, limiter l'accès à la commande `java`.

La deuxième solution consiste à définir un lanceur générique qui modifie son identité ou modifie son contexte de sécurité en fonction de l'application Java qu'il exécute. Cela est possible via l'utilisation combinée des API natives de la famille `setuid` et `setgid` et l'activation des bits SUID et SGID. Dans un environnement SELinux, il s'agit de la fonction `int setcon(security_context_t context)`. Cette solution n'est cependant pas satisfaisante pour restreindre l'exécution des applications Java. En effet, à partir du moment où le lanceur est générique, il est difficile de restreindre l'exécution d'une classe (cela nécessite de modifier tous les chargeurs de classes).

Implémenter le contrôle d'accès au niveau de la JVM. Cette solution consiste à implémenter un mécanisme de contrôle d'accès générique et de cloisonnement au sein même de la JVM. Elle permet d'étendre les capacités de

contrôle d'accès et de confinement de la plate-forme d'exécution Java. Elle vise également à assurer un certain niveau de protection de manière à ce que le mécanisme de contrôle d'accès n'ait plus à s'auto-protéger comme c'est le cas pour le mécanisme de contrôle d'accès standard de Java.

Ce mécanisme nécessite d'implémenter différents composants :

- un mécanisme de gestion des identifiants et éléments de sécurité (labels de sécurité, ACL, *credentials*, etc.) associés aux différentes classes et objets Java ;
- un moniteur de référence chargé de mettre en œuvre la politique de sécurité ;
- un mécanisme de gestion de la politique de sécurité.

Afin de proposer un modèle générique de contrôle d'accès implémenté entièrement au sein de la JVM, la gestion des éléments de sécurité doit permettre d'associer à chaque instance (et pas seulement à chaque classe, afin de ne pas restreindre les types de contrôles applicables), un élément de sécurité sous la forme d'un attribut dont l'accès depuis le code Java est limité (par exemple en lecture seule, ou suivant le contexte d'exécution). Ces éléments de sécurité ne définissent *a priori* aucune sémantique de contrôle d'accès particulière. Cette sémantique dépend de l'implémentation du moniteur de référence. Deux stratégies d'implémentation peuvent être envisagées, suivant la granularité de ces éléments de sécurité :

- prévoir un champ dédié pour chaque objet (à la manière de SELinux [1] ou SMACK [2]) ;
- implémenter les *isolates* (JSR 121) et associer les éléments de sécurité à chaque *isolate* (à la manière des *Trusted Extensions* [3] de Solaris).

La première stratégie permet un contrôle plus fin au détriment de la complexité (notamment lors de la spécification de la politique). La seconde est plus simple mais moins souple et nécessite en outre d'implémenter le concept des *isolates*.

Le moniteur de référence surveille les accès aux ressources protégées et décide, en fonction d'un algorithme donné et d'une politique de contrôle, d'autoriser ou non l'accès aux ressources. Dans l'architecture JPSA, ce moniteur est principalement implémenté en Java. Seul l'accès aux objets via l'exécution de méthodes est contrôlé. Pour la solution proposée ici, le moniteur est directement implémenté au sein de la JVM et les ressources sont constituées par les objets (et les classes) au sens de Java. En effet, à chaque ressource native (fichiers, *sockets*, etc.) est associé un objet Java. Le moniteur s'appuie sur les éléments de sécurité qui constituent une composante de la politique de contrôle.

L'algorithme de contrôle est dépendant du type de contrôle d'accès à mettre en œuvre. Il est possible d'envisager une architecture modulaire où différents algorithmes peuvent être utilisés suivant le type de contrôle à mettre en œuvre.

Il reste à définir quels accès doivent faire l'objet d'un contrôle. *A priori*, il est possible de distinguer deux types d'accès :

- la lecture ou la modification des champs de l'objet (ou de la classe) ;
- l'appel d'une méthode de l'objet (ou de la classe).

Ces accès sont réalisés de différentes manières en Java :

- directement, via l'exécution d'un *bytecode* adéquat (par exemple `getfield` pour accéder à un champ) ;
- via l'utilisation du mécanisme d'introspection ;
- depuis le code natif via JNI.

Des points de contrôle doivent être insérés dans le code de la JVM, au niveau du module d'exécution et de l'interface native, pour que ces différents accès aux objets Java soient tous contrôlés. Lorsque l'accès est refusé, il paraît nécessaire d'en avertir le code Java (via l'émission d'une exception de sécurité).

La définition de la politique de sécurité nécessite deux étapes :

1. associer les éléments de sécurité à chaque objet ;
2. définir les autorisations d'accès sous la forme de relations entre les éléments de sécurité.

Cette dernière étape peut être optionnelle si ces relations sont directement intégrées à l'algorithme du moniteur de référence. Dans le cas contraire, l'administrateur doit pouvoir exprimer ces relations de manière simple (par exemple à l'aide de fichiers de spécification de la politique).

4.3 Vérification de *bytecode* étendue

Comme nous l'avons déjà évoqué, le langage Java s'appuie sur des vérifications statiques (*i.e.* qui ont lieu avant l'exécution du programme) relativement poussées tant au niveau du compilateur que de la JVM (vérificateur de *bytecode*). Ces vérifications permettent de garantir l'absence de certains types d'erreurs et de failles de façon automatique. Elles s'appuient sur les annotations de types et de visibilité fournis par le programmeur. Le spectre des propriétés assurées par ce type de techniques peut être élargi en développant des extensions au vérificateur de *bytecode*. Le niveau *bytecode* est privilégié ici, puisqu'il est le seul permettant d'assurer la sécurité des applications fournies sans code source. Le projet JAVASEC étudie actuellement plusieurs extensions du vérificateur de *bytecode* permettant d'assurer des propriétés simples mais importantes pour la sécurité des application Java.

Absence de fuite d'objets partiellement initialisés. Le fait de laisser échapper un objet pendant sa construction (par un champ statique ou même un appel virtuel) est considéré comme une mauvaise pratique de programmation, voire une tentative d'attaque. Cette fuite est cependant inévitable dans de nombreux cas, mais il convient alors de s'assurer qu'un objet partiellement initialisé sera inutilisable par la suite. Le guide de recommandations de Sun [28] évoque cette problématique, mais fournit uniquement un schéma de programmation qu'il conseille de suivre. En s'appuyant sur des annotations spécifiques fournies par le programmeur, il est possible d'affiner le système de type standard de Java afin de traiter par vérification statique du *bytecode* cette problématique.

[28] `http://java.sun.com/security/seccodeguide.html`

Aide à la copie sécurisée d'objet. Une recommandation basique lors de la manipulation d'objets provenant de sources non-sûres (désérialisation, bibliothèques tierces) est de faire une copie profonde préalable de l'objet afin de couper toutes dépendances éventuelles entre le contenu de l'objet et du code malveillant. Cependant, en Java, la copie d'objet est à la charge du programmeur et donc sujette à erreur de programmation, voire à programmation malveillante. Le système d'annotation et de vérification étendue, étudié dans le cadre du projet JAVASEC, permettra de spécifier qu'une certaine méthode de copie assure une séparation mémoire jusqu'à une certaine profondeur entre sa source et son résultat. Cette signature pourra être vérifiée statiquement, donnant ainsi une garantie forte pour l'aide à la copie, même pour les bibliothèques tierces, du moment que celles-ci aient été munies d'une signature adéquate.

5 Conclusion

Java constitue une avancée intéressante pour la sécurité, car certaines propriétés de sécurité sont assurées par la plate-forme d'exécution et il n'est plus nécessaire de faire seulement confiance au code. Ceci permet de limiter le spectre des vulnérabilités donc des attaques (en particulier les problématiques de corruption de la mémoire). Toutefois, en pratique, il convient de garder à l'esprit que cette avancée ne signifie pas l'absence de problème de sécurité pour les applications Java :

- le spectre des propriétés assurées est limité. Certaines classes de vulnérabilités ne sont pas couvertes par la plate-forme (par exemple, les problèmes d'injection SQL) ;
- l'architecture de la plate-forme, la spécification et l'implémentation des mécanismes de sécurité (et des fonctions critiques) présentent des faiblesses qui peuvent remettre en cause les propriétés qui sont censées être garanties.

En particulier, la sécurité apportée par le langage Java se fait aujourd'hui au prix d'une grande complexité tant au niveau de l'implémentation de la plate-forme (mélange de différents modes d'exécution, représentation intermédiaire de code, lieu et moment de vérification) que de sa configuration (politique de visibilité, d'héritage, de contrôle d'accès). La dépendance forte entre tous les différents modules de l'architecture accentue le phénomène « château de carte » où une faille dans un module anodin peut faire s'écrouler la sécurité globale de l'architecture.

Le talon d'Achille de la plate-forme reste ses bibliothèques (notamment la bibliothèque standard) sans lesquelles le language n'aurait certainement pas connu son succès mais qui fournissent autant de services parfois extrêmement sensibles (failles dans le code natif, failles de mutabilité, escalades de privilèges, etc.). Il paraît important de concentrer les efforts de sécurisation de la plate-forme Java sur ce composant, que ce soit par l'augmentation de la qualité intrinsèque de l'implémentation ou par des techniques de cloisonnement et de contrôle d'accès. Dans une moindre mesure, se pose également le problème de confiance dans la JVM. Pour cette dernière, il est nécessaire de faire un compromis entre les per-

formances et les problématiques de sécurité, que ce soit en termes d'audit ou d'intégration avec les mécanismes de l'OS.

6 Remerciements

Les résultats présentés dans cet article sont issus de l'étude JAVASEC financée par l'ANSSI. Les auteurs tiennent particulièrement à remercier Eric Jaeger, Olivier Levillain, Benjamin Morin et Vincent Strubel de l'ANSSI pour leur relecture et leurs remarques constructives.

A Annexes

A.1 Sémantique Java : extrait de la classe `java.lang.Integer` commentée

```
// Declaration du package d'appartenance de la classe
package java.lang;

// Declaration de la classe Integer accessible depuis
// n'importe quelle autre classe (modificateur public)
// final indique qu'aucune classe ne peut heriter d'Integer
public final class Integer
// Heritage d'une et une seule classe (ici Number)
// Par defaut, toute classe herite de java.lang.Object
                        extends Number
// Implementation d'une ou plusieurs interfaces (ici une)
                        implements Comparable<Integer> {

    // private indique que le champ est accessible uniquement
    // par les elements declares dans la classe
    // final indique ici que ce champ ne peut etre ecrit
    // au plus une fois
    private final int value;
    // Declaration d'un champ accessible depuis n'importe
    // quelle classe (modificateur public)
    // static indique un champ de classe et non d'instance
    public final static int MAX_VALUE = 0x7fffffff;

    // Declaration d'un constructeur. Il peut y en avoir
    // plusieurs avec des signatures differentes
    public Integer(int value) {
        this.value = value;
    }

    // Declaration d'une methode locale a la classe
    // throws indique que la methode peut lever une exception
    public static int parseInt(String s)
                        throws NumberFormatException { ... }

    // methode devant etre implementee suite a
    // l'implementation de l'interface Comparable<Integer>
    public int compareTo(Integer anotherInteger)
    { ... }

    // redefinition d'une methode heritee de java.lang.Object
    public String toString() {
        return String.valueOf(value);
    }
}
```

A.2 Spécifications Java

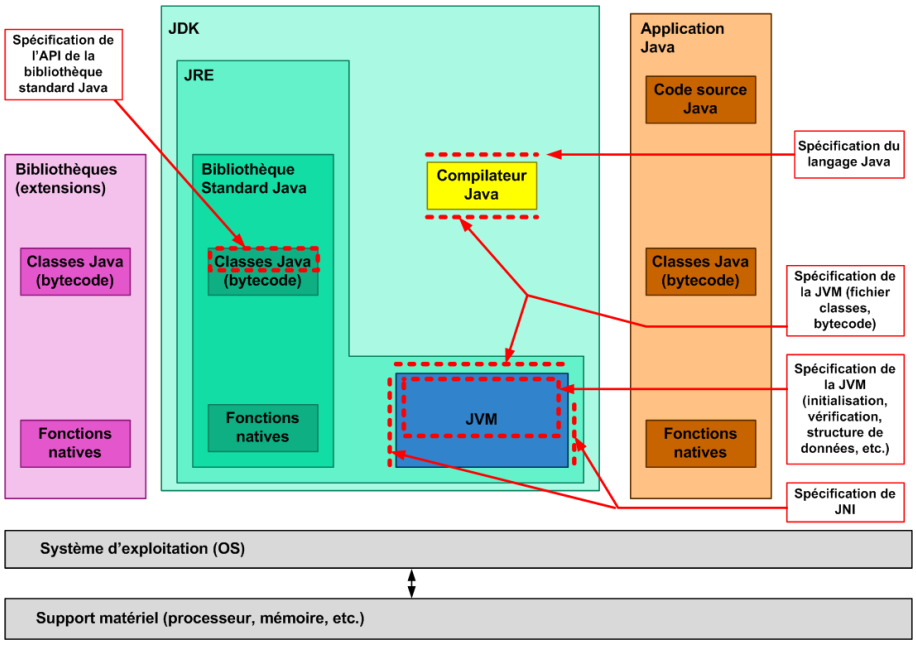

Fig. 6. Spécifications Java

A.3 Gestion de la mémoire

La gestion mémoire de Java est devenue de plus en plus complexe. Les allocations des objets nécessaires à l'exécution d'un programme Java sont faites dans plusieurs zones de données en mémoire. On peut en distinguer deux types :

- la zone mémoire locale à chaque *thread* contient un registre indiquant l'instruction en cours d'exécution pour les méthodes non natives, et une pile d'appels qui contient des *frames* qui interviennent dans la gestion des variables locales, de la pile d'opérandes et des appels et retours de méthodes ;
- la zone mémoire globale (ou le tas) où les instances de classe et les tableaux sont alloués et où sont également stockées les descriptions et *bytecode* de chaque méthode.

Un programme manipule des références vers les objets stockés dans les diverses zones mémoire. Pour l'implémentation de ces références, on distingue généralement deux approches : l'accès direct et la table d'indirections. Dans le second cas, le programme manipule les positions de la table plutôt que les adresses des objets.

Libération de la mémoire. Le langage Java ne fournit aucune primitive pour libérer la mémoire. Cette tâche, automatique, est dévolue à un mécanisme appelé *garbage collector*. Son rôle est de détecter parmi les objets alloués par le programme ceux qui sont devenus inaccessibles. Un objet est accessible si :

- la pile d'appels d'un *thread*, ou « racine » contient une référence vers cet objet au travers d'une variable locale ou d'une pile d'opérandes ;
- un des champs d'un objet accessible pointe vers cet objet.

Il existe plusieurs stratégies pour l'implémentation d'un *garbage collector*. Les deux principales familles sont le comptage de références et l'exploration. Les algorithmes à base d'exploration, plus fréquemment implémentés, reposent sur un parcours de la mémoire permettant de découvrir, à partir des racines, les blocs mémoire accessibles par le programme. Tous ces algorithmes supposent la suspension du programme pendant leur exécution. Les algorithmes utilisés sont le *Mark and Sweep*, le *Copying* et le *Mark and Compact*.

La suspension de l'application pendant toute la durée de l'exécution du *garbage collector* peut être pénalisante pour des applications réactives. Pour limiter l'impact de cette suspension, des optimisations ont été réalisées dont entre autres :

- Le *garbage collector mostly concurrent*. Il est basé sur l'agorithme *Mark and Sweep*, dont une partie importante de l'exécution peut être réalisée en parallèle avec celle du programme. Au cours d'une première phase, le programme est suspendu pour réaliser le marquage des racines. Ensuite, en parallèle de l'exécution du programme, le marquage du reste du tas est effectué. Les modifications du tas faites par le programme au cours de cette phase sont enregistrées. Le programme est suspendu et une nouvelle phase de marquage démarre, cette fois à partir des références enregistrées et ainsi de suite jusqu'à arriver à un état stable. La phase *Sweep* de l'algorithme peut être réalisée en parallèle de l'exécution du programme.
- Les *garbage collectors* générationnels. Le principe du *garbage collector* générationnel est de diviser le tas en un nombre fini de sous-tas appelé « générations ». L'utilisation de ces « générations » permet de trier les objets selon leurs âges dans le cycle de vie d'une application Java :
  - lors de l'allocation d'un objet, la mémoire est allouée dans la « génération des jeunes objets » ;
  - après plusieurs cycles de collecte, si un objet est toujours utilisé, il est alors promu : sa mémoire est déplacée dans « une génération d'objets d'âge supérieur » ;
  - généralement, il existe une « génération » pour les objets dits permanents : *i.e.* ceux dont les cycles de vie correspondent à celui de l'application Java. Par exemple, on y trouve les objets de type `Class`.

A.4 Exécution de *bytecode*

Suivant les JVM, différents modes d'exécution de *bytecode* Java sont implémentés. Ceux-ci s'éloignent plus ou moins de la spécification de la JVM. En effet,

celle-ci décrit une machine à pile dont l'implémentation la plus proche est consituée par un interpréteur de *bytecode* en code machine. Mais ce mode ne permet pas d'obtenir de bonnes performances. De fait, il a été optimisé de plusieurs manières. Ces optimisations sont détaillées ci-dessous :

- mode *switch* : c'est le mode d'interprétation le plus proche de la spécification. Les instructions *bytecode* d'une méthode sont décodées une à une à l'aide d'une boucle *switch/case*. À chaque *opcode* correspond une routine de traitement spécifique qui est alors exécutée.
- mode *threaded* : la boucle *switch/case* est cette fois-ci supprimée à cause de sa lenteur. Un tableau est tout d'abord constitué avec l'ensemble des couples (opcode, adresse de la routine de traitement). Deux cas de figure sont alors possibles :
  - soit une indirection subsiste : une routine se charge d'identifier dans le tableau l'adresse de la routine de traitement associée à chaque instruction *bytecode* ;
  - soit l'instruction *bytecode* est directement remplacée par l'adresse du traitant. L'exécution consiste alors à enchaîner les routines pointées par ces adresses.
- mode *inline* : dans ce mode, les instructions *bytecode* d'une méthode sont remplacées par une liste chaînée contenant le corps des routines de traitement correspondantes. Il s'agit donc d'une forme de compilation simplifiée, réalisée au moment du décodage des instructions.
- mode *template* : similaire au mode *inline*, mais le corps des routines de traitement ne contient pas du code machine, mais une forme de méta-assembleur générique (*i.e.* agnostique de la plate-forme sous-jacente). Au moment de l'exécution, les routines présentes dans la liste chaînée sont compilées en code machine à la volée et exécutées une à une.

Une autre approche en termes d'exécution de *bytecode* consiste à réaliser de la compilation à la volée (également appelée JIT – *Just In Time*). Ce mode d'exécution se rapproche grandement du mode d'interprétation *template*, à ceci près que l'opération de compilation à la volée est réalisée sur un bloc d'instructions successives. La compilation à la volée est organisée en plusieurs étapes :

1. Optimisations globales :
   - INLINING : déplie un appel de méthode au sein du *bytecode* d'une méthode. Dans le cas du JIT, l'appel de méthode est ainsi remplacé par le *bytecode* de la méthode appelée.
   - OPTIMISATION DES BOUCLES : permet de diminuer la complexité d'exécution des boucles en réalisant différentes améliorations (détection des bornes d'un tableau et élimination des tests triviaux de dépassement de tableau, déroulement des boucles lorsque celles-ci itèrent peu de fois, fusion des boucles ayant un en-tête commun, etc.).
2. Création d'une ou de plusieurs représentations intermédiaires du code. Plusieurs algorithmes existent à ce sujet :
   - SSA (*Single Static Assignement*) : algorithme utilisé par certaines JVM lors de la création d'une réprésentation intermédiaire d'une portion de

code. Le SSA, comme son nom l'indique, permet de générer un code où chaque variable est assignée de manière unique.
  - GVN (*Global Value Numbering*) : optimisation d'une représentation intermédiaire de code de type SSA, à ceci près que les expressions sont également rendues uniques au sein d'une portion de code.

3. Application d'optimisations sur ce code intermédiaire, dont par exemple :
  - IF-CONVERSION : permet de fusionner plusieurs branches conditionnelles, en utilisant des instructions conditionnelles avec prédicat présentes sur certaines architectures telles que ARM et IA-64.
  - RÉORGANISATION DES INSTRUCTIONS : réordonnancement, fusion ou encore suppression d'instructions du code intermédiaire.
  - ESCAPE ANALYSIS : permet d'optimiser l'allocation mémoire des objets et de supprimer les verrous inutiles en fonction de leur visibilité (visibilité au sein d'une méthode, d'un *thread* ou globale). L'élimination des verrous permet notamment de réduire la charge due aux mécanismes de synchronisation. À noter que ce mécanisme s'applique sur une représentation intermédiaire de code de type GVN.

4. Traduction en code machine avec allocation spécifique des registres du processeur.
  - LSRA (*Linear Scan Register Allocation*) : permet de réordonner des portions de code de manière à rendre linéaire l'étape d'allocation des registres du processeur en fonction de leur fréquence d'utilisation.

Le JIT permet de gagner en performance à l'exécution mais il génère un surcoût lors de la compilation (donc lors de la première exécution d'une méthode), surtout lorsque des optimisations « lourdes »sont appliquées. Afin de tirer parti des avantages des deux principaux modes d'exécution la compilation dite « dynamique » a été implémentée sur certaines JVM (notamment HotSpot de Sun). Concrètement, il s'agit de modifier dynamiquement le mode d'exécution courant afin de choisir le plus adapté au contexte. Ainsi, lors de l'exécution d'un programme Java, les méthodes les moins sollicitées vont être interprétées et les méthodes les plus fréquemment appelées seront compilées à la volée en code machine puis exécutées. Une analyse permanente de l'usage de chaque méthode (*profiling*) est alors nécessaire pour identifier le mode d'exécution le plus efficace.

L'OSR (*On Stack Replacement*) joue un rôle primordial en cas de compilation dynamique. L'OSR intervient lorsque la JVM décide de modifier, en cours de route, le mode d'exécution d'une méthode. Lors d'un tel basculement, le mécanisme d'OSR recopie les valeurs des variables locales ou les valeurs de la pile dans l'environnement compilé ou interprété, et continue l'exécution de la méthode dans ce nouvel environnement.

D'autres modes d'exécution existent, comme la compilation en avance de phase (AOT – *Ahead Of Time*), mais ne sont pas détaillés puisqu'ils sont peu mis en œuvre actuellement.

Références

1. SELinux. `http://www.nsa.gov/research/selinux/index.shtml`.
2. SMACK. `http://www.schaufler-ca.com/`.
3. Solaris Trusted Extensions. `http://hub.opensolaris.org/bin/view/Community+Group+security/tx`.
4. Bloch,, Joshua. *Effective Java Programming Language Guide*. Sun Microsystems, Inc., Mountain View, CA, USA, 2001.
5. Campione,, Mary and Walrath,, Kathy and Huml,, Alison. *The Java Tutorial : A Short Course on the Basics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
6. Claude Delannoy. *Programmer en Java (3ème édition)*. Eyrolles, 2008.
7. Common Vulnerabilities and Exposures website. CVE-2008-5353. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-5353`, 2008.
8. Eckel,, Bruce. *Thinking in Java*. Prentice Hall Professional Technical Reference, 2002.
9. Flanagan,, David. *Java In A Nutshell, 5th Edition*. O'Reilly Media, Inc., 2005.
10. Gong,, Li and Ellison,, Gary. *Inside Java(TM) 2 Platform Security : Architecture, API Design, and Implementation*. Pearson Education, 2003.
11. Horstmann, Cay S and Cornell, Gary. *Core Java. Revised and Updated for Java SE 6 ; 8th ed.* Prentice-Hall, Upper Saddle River, NJ, 2008.
12. James Gosling and Bill Joy and Guy Steele and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.
13. Joshua Bloch. *Effective Java, 2nd edition*. Addison Wesley, 2008.
14. Sami Koivu. Calendar bug. 2008. `http://slightlyrandombrokenthoughts.blogspot.com/2008/12/calendar-bug.html`.
15. LSD. Java and java virtual machine security vulnerabilities and their exploitation techniques. In *Black Hat Asia*, 2002.
16. Oaks,, Scott. *Java Security Second Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.
17. Philippe Prados. La sécurité de java. *MISC*, 2009.
18. Philippe Prados. Porte dérobée dans les serveurs d'applications javaee. *MISC*, 2009.
19. Jérémy Renard and Alexandre Ahmim-Richard. Vulnérabilités liées aux serveurs d'applications j2ee. *MISC*, 2009.
20. Fabrice Rossi. L'architecture de sécurité de java. *MISC*, 2003.
21. Marc Schönefeld. Hunting flaws in jdk. In *Black Hat Europe*, 2003.
22. Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
23. Julien Tinnes. One bug to rule them all : la faille calendar.java. *MISC*, 2009.