SILICOM
RÉGION OUEST

AMOSSYS

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE
centre de recherche RENNES - BRETAGNE ATLANTIQUE

INRIA

SGDN

Projet: JAVASEC

Type : rapport d'étude

Guide de règles et de recommandations relatives au développement d'applications de sécurité en Java

Référence	:	JAVASEC_NTE_002
Version	:	1.3
Nb pages	:	106
Date	:	6 novembre 2009

TABLE DES MATIÈRES

1 INTRODUCTION

1.1 Objet du document

Ce document est réalisé dans le cadre du Projet JAVASEC, relatif au marché 2008.01801.00.2.12.075.01 notifié le 23/12/2008. Il correspond au deuxième livrable technique contractuel émis au titre du poste 1 (en version finale) : Guide de développement en version finale (identifiant 1.3.3 dans le CCTP).

Ce guide présente des règles et recommandations à destination du développeur d'applications Java. Il s'appuie sur le « Rapport d'étude sur le langage Java » [Con09c].

1.2 Présentation du contexte

Java est un langage de programmation orienté objet développé par Sun. En plus d'un langage de programmation, Java fournit également une très riche bibliothèque de classes pour tous les domaines d'application de l'informatique, d'Internet aux bases de données relationnelles, des cartes à puces et téléphones portables aux superordinateurs. Java présente des caractéristiques très intéressantes qui en font une plateforme de développement constituant l'innovation la plus intéressante apparue ces dernières années.

Dans le cadre de ses activités d'expertise, de définition et de mise en œuvre de la stratégie gouvernementale dans le domaine de la sécurité des systèmes d'information, l'ANSSI souhaite bénéficier d'une expertise scientifique et technique sur l'adéquation du langage Java au développement d'applications de sécurité, ainsi que d'une démarche permettant d'améliorer la confiance vis-à-vis de la sécurité de ces applications.

Ce document constitue la partie « guide méthodologique » de cette étude, document ayant pour objectif de permettre le développement d'applications Java avec un meilleur niveau de sécurité.

1.3 Organisation du document

Chapitre	Intitulé	Contenu
2	Glossaire, Acronymes	Description des différents acronymes du document.
3	Définitions et présentation de la démarche	Rappels sur le langage et l'environnement d'exécution Java, présentation de la démarche.
4	Recommandations relatives au langage Java	Règles et recommandations relatives au cœur du langage Java.
5	Recommandations relatives à la bibliothèque standard	Règles et recommandations relatives à l'utilisation de la bibliothèque standard.
6	Annexes	

2 GLOSSAIRE, ACRONYMES

Acronyme	Définition
API	Application Programming Interface : interface de programmation
CSP	Cryptographic Service Provider
JAAS	Java Authentication and Authorization Service : *package* de la bibliothèque standard pour l'authentification et le contrôle d'accès
JCA	Java Cryptography Architecture : *package* de la bibliothèque standard pour la cryptographie
JCE	Java Cryptography Extension : *package* de la bibliothèque standard pour la cryptographie
JCP	Java Community Process ou Java CertPath (suivant le contexte)
JDK	Java Development Kit : environnement de développement Java
JIT	Compilation Just In Time : compilation à la volée
JNI	Java Native Interface : interface de programmation pour l'intégration des fonctions natives
JPDA	Java Platform Debugger Architecture : interface de mise au point
JRE	Java Runtime Environment : environnement d'exécution Java
JSASL	Java Simple Authentication and Security Layer
JSR	Java Specification Request
JSSE	Java Secure Socket Extension
JVM	Java Virtual Machine : machine virtuelle Java
JVMTI	Java Virtual Machine Tool Interface : interface de mise au point
PKI	Public Key Infrastructure
PRNG	Pseudorandom Number Generator
RMI	Remote Method Invocation
TPM	Trusted Platform Module

3 DÉFINITIONS ET PRÉSENTATION DE LA DÉMARCHE

Le but de cette section est de rappeler brièvement les concepts de Java, de préciser la définition de certains termes utilisés et de présenter la démarche adoptée lors de la rédaction des règles et recommandations. Ce document ne constitue ni une introduction à Java ni une référence sur la spécification du langage ou de la machine virtuelle Java (JVM). Il est supposé que le lecteur est déjà familier avec les principaux concepts de la programmation Java. Le lecteur désirant s'initier à la programmation Java pourra se référer aux documents [Hor08, Eck02, Cam00, Cla08]. Les ouvrages [Fla05, Hor08, Blo01, Jam05] constituent des références sur le langage et sa bibliothèque standard. La spécification de la JVM est précisée dans l'ouvrage [Tim99]. La problématique de la sécurité en Java est abordée dans les ouvrages [Gon03, Oak01].

Ce document constitue un ensemble de bonnes pratiques de développement en ce qui concerne les aspects liés à la sécurité informatique. Les autres types de problématiques, notamment les problèmes de performance, de lisibilité et de structuration du code source ne sont pas évoqués. Ces recommandations portent sur le cœur du langage et sur les fonctionnalités de la bibliothèque standard Java (Java SE). Elles ont été élaborées à partir de l'analyse du langage et de l'architecture de la plate-forme d'exécution Java, évoquées dans les différents rapports du projet JAVASEC [Con09c, Con09d, Con09a, Con09b]. Des démarches similaires ont été suivies par différents organismes :

- Sun fournit un guide de développement concernant les aspects liés à la sécurité, accessible à l'adresse `http://java.sun.com/security/seccodeguide.html` ;
- le CERT de l'université Carnegie Mellon met en place des standards pour le développement sécurisé d'applications Java. À la date de rédaction de ce présent rapport, ces standards sont en cours de spécification. Ils sont accessibles via un site de type Wiki à l'adresse `https://www.securecoding.cert.org/confluence/display/java` ;
- l'OWASP fournit également des informations de sensibilisation des développeurs d'applications Java EE aux problématiques de sécurité. Bien que la plupart de ces informations sortent du périmètre de l'étude JAVASEC qui se concentre sur Java SE, ce document constitue un complément intéressant. Il est accessible à l'adresse `http://www.owasp.org/index.php/OWASP_Java_Table_of_Contents#J2EE_Security_for_Developers`.

Le présent document s'inspire en partie des recommandations fournies par les documents cités précédemment.

3.1 Les principaux concepts de Java

Java désigne à la fois un langage de haut niveau et une plate-forme d'exécution, tous deux issus des travaux de recherche menés dans les années 1990 par Sun Microsystems dans le cadre du projet *Stealth* (renommé *Green project* par la suite).

La philosophie de Java permet de répondre à plusieurs objectifs :

1. utiliser un langage de haut niveau et orienté objet ;
2. faciliter le développement des applications en proposant un langage simple et inspiré de C++ ;
3. faciliter le déploiement des applications en s'assurant qu'une même application puisse s'exécuter sur des environnements différents (UNIX, Windows, Mac, etc.) ;
4. permettre l'utilisation de manière native des réseaux informatiques ;
5. permettre l'exécution de code distant de manière sécurisée.

Le langage Java, à proprement parler, permet de répondre aux deux premiers objectifs. Il s'agit en effet d'un langage orienté objet fortement inspiré de C++ de par sa syntaxe. Les principales différences par rapport à C++ sont les suivantes :

- la gestion de la mémoire est automatisée via l'utilisation d'un ramasse-miettes (*garbage collector*). Le programmeur ne peut gérer directement les allocations mémoires et manipule les objets via les références [1] qui permettent d'abstraire la notion de pointeur utilisée en C++ ;
- l'héritage multiple n'est pas possible en Java qui propose en revanche la notion d'interface. À la différence d'une classe, une interface ne contient pas de code, mais seulement des descriptions de méthodes et des constantes. Une classe Java hérite au plus d'une classe et peut implémenter plusieurs interfaces.
- l'intégralité du code doit être implémentée sous la forme de méthodes de classes (statique ou non) ;
- le compilateur Java n'utilise pas de préprocesseur. Il n'est donc pas possible d'utiliser des macros ou des définitions de constantes (`#define`, etc.) ;
- le type primitif `boolean` n'est pas compatible avec les types numériques (`int`, `long`, etc.) (à l'inverse du type `bool` de C++) ;
- l'instruction `goto` a été supprimée.

L'environnement standard de Java définit, en plus du langage, un certain nombre de classes de base regroupées dans des *packages* de la bibliothèque standard. Ces classes offrent différents

1. La notion de référence en Java est différente de la notion de référence en C++. En Java, une même référence peut, durant une exécution, référencer différents objets distincts, ce qui n'est pas possible en C++. En revanche, à l'inverse des références C++, les références Java ne peuvent référencer que les objets et les tableaux (et non les variables de type primitif).

services, tels que l'accès aux ressources du système (fichiers, base de données, etc.). Elles permettent notamment de programmer des applications communiquant via un réseau informatique grâce à la gestion des *sockets* ou des applications utilisant la programmation concurrente (gestion de plusieurs files d'exécutions ou *threads*). Elles répondent donc en partie aux objectifs 2 et 4.

Afin de répondre au troisième objectif, le modèle standard d'exécution des programmes Java repose sur l'utilisation d'une machine virtuelle (JVM). Le compilateur Java effectue la traduction du code source Java de haut niveau vers un langage de niveau intermédiaire ou *bytecode* Java. Ce dernier est, par la suite, exécuté sur la machine virtuelle. Celle-ci constitue une interface standard qui permet d'abstraire les différentes plates-formes d'exécution. Elle assure notamment l'exécution du *bytecode* sur l'architecture de la plate-forme d'exécution (x86, Sparc, PPC, etc.). Historiquement, l'exécution est réalisée par un interpréteur mais, pour des raisons d'optimisation, les JVM actuelles intègrent généralement un compilateur « à la volée » (ou *JIT*). L'utilisation d'une machine virtuelle permet de déployer les applications Java sous forme compilées (bytecode) tout en garantissant que l'application s'exécutera, dans le cas idéal, de la même manière sur les différentes plates-formes natives.

Les spécifications du langage de haut niveau [Jam05], de l'API fournie par la bibliothèque standard [2] et du *bytecode* [Tim99] sont fournies par Sun. Différents fournisseurs (dont Sun) proposent leur propre implémentation de la JVM et de la bibliothèque standard. L'ensemble des éléments nécessaires pour exécuter une application Java fournie sous la forme de fichiers de classes (contenant le *bytecode* de l'application) est appelé *Java Runtime Environment* (JRE). Afin de développer des applications Java, un certain nombre d'outils supplémentaires est nécessaire (dont un compilateur Java permettant d'effectuer la traduction du langage Java de haut niveau vers le *bytecode*). L'ensemble des outils nécessaires au développement d'applications Java (comprenant ceux présents dans le JRE) constitue le *Java Development Kit* (JDK). Depuis la version 2 de Java, l'environnement d'exécution standard est décliné en trois versions :

- Java SE (*Standard Edition*), l'édition de référence ;
- Java EE (*Entreprise Edition*), dédiée aux applications Web (EJB, Servlet, JSP, etc.) ;
- Java ME (*Micro Edition*), dédiée aux applications embarquées.

Le présent document s'applique aux versions 1.5 et 1.6 de l'environnement Java SE. De plus, les aspects liés aux problématiques de code mobile (*applet*), ne sont pas traités en profondeur.

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 en parallèle. 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. Cette définition correspond notamment à la configuration de l'implémentation de Sun sous Linux et sera retenue dans la suite de ce document.

2. `http://java.sun.com/javase/6/docs/api/`

3.2 Définitions

Cette section vise à rappeler la définition de certains termes propres à Java et qui sont utilisés dans la suite du document. Ce document utilise autant que faire se peut la traduction française de ces termes.

Terme anglais	**Traduction francaise**	**Définition**
Class	Classe	Une classe représente une catégorie d'objets et contient un ensemble de propriétés définissant cette catégorie d'objets. La classe déclare des attributs représentant l'état des objets et des méthodes représentant leur comportement.
Inner class	Classe interne	Une classe interne est une classe dont la déclaration est faite au sein d'une autre classe.
Subclass	Sous-classe	Considérant le mécanisme d'héritage, on dit que `B` est une sous-classe de `A` si la classe `B` hérite de la classe `A`.
Object	Objet	En Java, un objet est de type (ou une sous-classe de) `java.lang.Object`. Un objet est donc soit une instance d'une classe, soit un tableau. En effet, en Java, tout tableau hérite de la classe `java.lang.Object` et ainsi est donc considéré comme un objet.
Package	Paquetage	Un *package* est un regroupement de classes, d'interfaces et de *sous-package* correspondant à certaines fonctionnalités.
Reference	Référence	Une référence est une valeur qui permet l'accès en lecture et en écriture à une donnée située soit en mémoire principale soit ailleurs. Une référence n'est pas la donnée elle même, mais seulement une information de localisation.
Serialization	Sérialisation	La sérialisation est un mécanisme servant à encoder l'état d'un objet se trouvant en mémoire sous la forme d'une suite d'octets. Cette suite peut par exemple être utilisée pour la sauvegarde ou le transport sur le réseau.
Deserialization	Désérialisation	La désérialisation est l'opération inverse de la sérialisation. Il permet de reconstruire un objet à partir d'une suite d'octets représentant cet objet.

Terme anglais	Traduction francaise	Définition
Reflexion	Réflexion (ou réflexivité)	La réflexion est la capacité d'un programme à examiner (introspection), et éventuellement à modifier (intercession), la structure de ses objets lors de son exécution.

3.3 La démarche

Ce document aborde la problématique du développement d'applications sécurisées en Java en proposant des règles et recommandations à suivre lors des processus de spécification, de conception et de développement d'applications ou de bibliothèques externes en Java. Ces règles et recommandations portent principalement sur l'utilisation du langage Java et de sa bibliothèque standard. Certains éléments connexes tels que la configuration de l'environnement d'exécution, l'interface avec la plate-forme d'exécution native ou l'utilisation de l'interface de mise au point sont également évoqués dès lors qu'ils présentent un risque pour la sécurité et qu'ils possèdent un impact sur le processus de développement. Les règles et recommandations se veulent génériques et doivent s'appliquer quel que soit le mode d'exécution (exécution standard sur JVM, compilation native ou processeur Java) et l'implémentation de l'environnement d'exécution utilisé. Toutefois, lorsqu'il paraît nécessaire d'illustrer par un exemple dépendant de l'implémentation, l'hypothèse est faite que l'application Java est exécutée sur le JRE fourni par Sun sur une plate-forme native de type Linux x86.

Les règles et recommandations sont présentées par la suite sous forme de fiches récapitulatives. Chaque fiche correspond à une problématique de sécurité particulière identifiée notamment lors de l'étude du langage Java et des modèles d'exécution. Les différentes fiches sont regroupées par thèmes dans différentes sections. Les problématiques liées au cœur du langage sont notamment dissociées de celles résultant de l'utilisation de mécanismes de la bibliothèque standard. Chaque fiche comprend les champs suivants :

- un identifiant de fiche ;
- un nom de fiche indiquant l'objet de la fiche (éléments Java impliqués ou résumé de la problématique) ;
- un indice du risque lié à la non prise en compte de la recommandation. Cet indice peut prendre les valeurs **F** (faible), **M** (modéré) et **E** (élevé) ;
- une description de la problématique abordée dans la recommandation ;
- une évaluation des impacts potentiels sur le processus de développement entraînés par l'application de la règle ;
- des références externes éventuelles ;
- des références internes entre fiches (champs « Voir également »).

Le champ **Recommandations** peut comprendre des règles, indiquées en gras, qui doivent *a priori* être appliquées. Ceci permet de les distinguer des recommandations qui constituent des indications et qui peuvent être modulées en fonction des impératifs fonctionnels. Lorsqu'il paraît nécessaire d'attirer l'attention du développeur sur une caractéristique ou une limite particulière, la fiche peut comporter des avertissements, indiqués en italique gras. Nous insistons sur le fait que les règles et recommandations présentées dans ce document constituent de bonnes pratiques qu'il convient de suivre afin de se prémunir des risques liés à la sécurité informatique. Elles doivent compléter les règles génériques de développement (annotation du code source, choix de noms de variables explicites, etc.). Le langage Java étant susceptible d'évoluer, ce document sera susceptible d'être mis à jour afin de prendre en compte les modifications apportées au langage.

L'illustration de la problématique ou des recommandations nécessite parfois de présenter un ou plusieurs exemples de codes conformes ou non conformes. Dans la mesure du possible, les fiches sont illustrées par des exemples concis de code conforme, inclus au sein des fiches. Lorsque un exemple plus conséquent est nécessaire, il est renvoyé en annexe. Les codes non conformes sont uniquement présentés lorsqu'ils présentent un apport significatif pour la compréhension de la problématique. Ils sont systématiquement renvoyés en annexe et sont clairement identifiés par un fond de couleur rouge.

4 RECOMMANDATIONS RELATIVES AU LANGAGE JAVA

4.1 Héritage

Identifiant : 1	**Nom :** Héritage	**Risque :** M

Problématique :

Une classe déclarée comme une sous-classe d'une autre classe hérite de l'ensemble des membres de sa super-classe, y compris les membres `private` même si ces derniers ne lui sont pas directement accessibles. La classe a la possibilité de redéfinir une méthode héritée, en fournissant sa propre définition d'une méthode avec les mêmes noms et paramètres. L'héritage et la redéfinition de méthodes peuvent permettre de contourner des contrôles de sécurité. Si une classe `B` héritant de la classe `A` redéfinit une méthode `foo` faisant des vérifications de sécurité, un appel virtuel `x.foo()` va appeler la méthode définie dans `B` dès lors que la variable `x` contient un objet de classe `B` ou d'une des sous-classes de `B` qui ne redéfinit pas la méthode `foo`.

Recommandations :

De manière générale, interdire la re-définition d'une classe qui n'a pas vocation à être étendue, en déclarant la classe `final`.

Si cela n'est pas possible, privilégier d'interdire la re-définition d'une méthode effectuant des contrôles de sécurité en la déclarant `final`. À défaut, le développeur doit effectuer une analyse du flot de contrôle du code pour assurer que ces contrôles ne sont pas contournés lors des appels virtuels.

Ne pas masquer les champs hérités en déclarant des champs du même nom.

Exemple :

```
class A {
  protected final void securityCheck(){
    ...
  }
  public final int foo(){
    securityCheck() ;  // acces control checks
    return ...;
  }
}
```

Impacts potentiels sur le processus de développement :

Modéré. Le développeur peut être amené à revoir l'architecture de son code afin de regrouper et d'isoler les contrôles de sécurité dans des méthodes qui ne sont pas redéfinissables.

Références :

[1] `https://www.securecoding.cert.org/confluence/display/java/OBJ33-J.+Limit+the+extensibility+of+non-final+classes+and+methods+to+only+trusted+subclasses`

[2] `http://java.sun.com/security/seccodeguide.html` *Guideline 1-2 : Limit the extensibility of classes and methods.*

Voir également :

- Fiche 2, page 14

Identifiant : 2	**Nom :** Héritage implicite de `java.lang.Object`	**Risque :** M

Problématique :

La class `Object` comporte plusieurs méthodes non-finales (`equals`, `hashCode`, `toString`, `clone` et `finalize`) dont toute classe hérite automatiquement. Le comportement par défaut proposé par ces méthodes est parfois inadapté et il convient alors de redéfinir certaines de ces méthodes.

Il est aussi important de remarquer que la méthode `clone`, qui est héritée de la classe `java.lang.Object`, ne pourra être utilisée que si la classe implémente l'interface vide `Cloneable`. Dans ce cas, le fait d'implémenter l'interface `Cloneable` active la fonction de clonage.

Recommandations :

Redéfinir les méthodes `equals`, `hashCode`, `toString` en respectant la spécification générique imposée par Sun [1].

Dans une classe implémentant l'interface `Cloneable`, il faut redéfinir la méthode `clone` en respectant la spécification générique imposée par Sun [1].

Ne redéfinir la méthode `finalize` que dans les cas appropriés (voir Fiche 14, page 34).

Impacts potentiels sur le processus de développement :

Modéré.

Références :

[1] *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.*

[2] [Jos08], Item 8, *Obey the general contract when overriding equals*.

[3] [Jos08], Item 9, *Always override hashCode when you override equals*.

[4] [Jos08], Item 10, *Always override toString*.

[5] [Jos08], Item 11, *Override clone judiciously*.

[6] [Jos08], Item 7, *Avoid finalizers*.

Voir également :

- Fiche 1, page 12
- Fiche 14, page 34

4.2 Accessibilité

Identifiant : 3	**Nom :** Visibilité `private`	**Risque :** E

Problématique :

L'accès aux membres d'un objet depuis un autre objet est contrôlé lors de la compilation et lors du chargement d'un *bytecode* par les modificateurs de visibilité associés à chaque membre d'une classe. Les modificateurs de visibilité définissent depuis quelles classes il est possible de faire référence à un membre (champ ou méthode). Ils sont au nombre de quatre :

- `public` : le membre est visible depuis toutes les autres classes quel que soit leur *package* ;
- `protected` : le membre est visible par toutes les classes du *package* qui contient la classe définissant ce membre et par toutes les classes qui héritent de cette classe. Dans ce dernier cas, il faut noter que la visibilité pour une classe `B` héritant de `A` sur les membres `protected` de la classe `A` est limitée aux instances de `B` et aux instances des sous-classes de `B`. L'utilisation de ce modificateur implique qu'une sous-classe peut accéder aux données, voire la divulguer à toute autre classe ;
- (package) le modificateur par défaut. Les membres sont visibles par toutes les classes du *package* qui contient la classe définissant le membre. Le risque ici est qu'une autre classe peut se déclarer membre du *package* et ainsi gagner l'accès aux membres déclarés (package) ;
- `private` : le membre est uniquement visible dans sa classe. Le modificateur `private` ne veut pas dire que le membre est uniquement accessible depuis son objet d'appartenance. Tous les objets de la même classe peuvent accéder à un champ privé d'un objet de cette classe s'ils possèdent une référence à l'objet. La compilation des classes internes peut effacer ce modificateur.

Un risque potentiel est de donner une trop large visibilité aux membres d'une classe (surtout les champs) qui permet à d'autres applications d'accéder à des données à protéger et potentiellement de les modifier.

Il faut noter que le mécanisme de réflexion permet de contourner les vérifications concernant la visibilité d'une classe, d'un champ ou d'une méthode.

Recommandations :

Réduire au maximum la visibilité des champs d'une classe. Privilégier des champs privés avec des méthodes pour y accéder (des *getters* et *setters*) qui peuvent effectuer des vérifications dynamiques de contrôle d'accès.

Ne pas donner la visibilité `protected`, `public` ou la visibilité par défaut (package) sur des champs qui doivent contenir des données à protéger.

Déclarer les champs non confidentiels et les méthodes (en particulier les *getters* et les *setters*) `final` quand c'est possible. Pour les champs contenant des données confidentielles, voir la fiche « Gestion de la mémoire » dans la section 4.6.

Il faut se prémunir de l'utilisation des mécanismes de réflexion afin de garantir que les modificateurs de visibilité sont bien pris en compte (voir fiche 9, page 25).

Impacts potentiels sur le processus de développement :

Modéré. La prise en compte des recommandations nécessite une analyse préalable des niveaux de sécurité des données manipulées, mais n'a par ailleurs pas d'impact sur le processus de développement. De plus, il faut configurer la politique de sécurité de manière à empêcher entre autre l'utilisation de la réflexion. La configuration complète de la politique de sécurité est assez lourde.

Références :

[1] `https://www.securecoding.cert.org/confluence/display/java/OBJ00-J.+Declare+data+members+private`

[2] `http://java.sun.com/security/seccodeguide.html` *Guideline 1-1 Limit the accessibility of classes, interfaces, methods, and fields*

[3] `http://java.sun.com/developer/JDCTechTips/2001/tt0130.html` *Controlling Package Access With Security Permissions and Sealed JAR Files*

Voir également :

- Fiche 5, page 19
- Fiche 4, page 17
- Fiche 9, page 25
- Fiche 14, page 34

Identifiant : 4	**Nom :** Visibilité au sein d'un *package*	**Risque :** M

Problématique :

Par défaut, un membre (champ ou méthode) d'une classe est visible dans le *package* où la classe est définie. Les membres déclarés `protected` sont également visibles par les classes se trouvant dans le même *package*.

Un *package* est par défaut ouvert et d'autres classes peuvent se déclarer membre du *package* et ainsi gagner une meilleure visibilité sur les membres des classes du *package*. Il est possible de configurer la politique de sécurité afin de spécifier un accès plus restreint aux *packages*, voir [1].

Recommandations :

Considérer les méthodes `protected` comme des méthodes publiques qui peuvent être redéfinies par des sous-classes externes (ce qui entre autre permet de contourner des contrôles d'accès effectués dans ces méthodes).

Sceller les *packages* pour éviter que des classes malveillantes puissent se déclarer membre du *package* et ainsi gagner une meilleure visibilité sur les membres (déclarés `protected` ou (*package*)) des classes du *package*. Ainsi, aucune autre classe ne peut se déclarer du même *package* et être chargée sans lever une exception de sécurité (`java.lang.SecurityException: sealing violation`). Pour sceller un *package*, il faut placer les fichiers `.class` dans une archive JAR et y adjoindre un fichier `Manifest.MF` où les *packages* sont déclarés comme étant scellés. Par exemple, pour le *package* `com.develop.impl`, il faudra :

1. Créer un fichier `Manifest` dans lequel on ajoute les 2 lignes suivantes (s'il y a plusieurs *packages*, il faudra répéter cette étape pour chacun d'entre eux) :

   ```
   Name: com/develop/impl/
   Sealed: true
   ```

2. Créer l'archive JAR, par la commande `jar cvmf Manifest archive.jar com`

Privilégier la définition de méthodes pour accéder aux champs au lieu de rendre ces derniers directement visibles à l'extérieur du *package*.

Impacts potentiels sur le processus de développement :

Modéré. Les méthodes qui doivent être visibles par des sous-classes extérieures aux *packages* peuvent être déclarées en utilisant le modificateur de visibilité `protected`. Les classes doivent être placées au sein d'archives JAR qui peuvent être scellées et signées.

Références :

[1] `http://java.sun.com/developer/JDCTechTips/2001/tt0130.html` *Controlling Package Access With Security Permissions and Sealed JAR Files.*

Voir également :

- Fiche 31, page 71

Identifiant : 5	**Nom :** Classes internes	**Risque :** F

Problématique :

Il est possible de définir une classe à l'intérieur d'une autre classe, soit comme un champ de classes soit à l'intérieur d'un bloc de code d'une méthode. Dans ce dernier cas, la classe n'est ni visible à l'extérieur de la méthode, ni accessible (en terme de langage) par une autre classe, car celle-ci est déclarée de manière anonyme. Pour une classe définie dans une méthode d'une classe `C`, un fichier `C$1.class` est alors créé.

Les classes internes ont été signalées comme une source de problèmes de sécurité lors de la compilation vers du *bytecode* Java. La notion de classe interne n'existe pas au niveau *bytecode* et il est donc nécessaire de rendre globale une classe interne lors de sa compilation en *bytecode*. Or, pour donner accès aux membres internes de la classe englobante par la classe interne, il est parfois nécessaire d'augmenter la visibilité des membres privés et de les rendre accessibles à l'intérieur du *package* où sont définies les deux classes.

Recommandations :

Ne pas accéder à un champ déclaré private d'une classe englobante à partir d'une classe interne.

Éviter si possible d'utiliser des classes internes.

Impacts potentiels sur le processus de développement :

Faible.

Références : S.O.

Voir également :

- Fiche 7, page 21

Identifiant : 6	**Nom :** Utilisation des champs `static`	**Risque :** E

Problématique :

Un champ d'une classe peut être déclaré statique (mot-clé `static`). Les champs statiques sont plus faciles d'accès que les champs d'un objet puisqu'ils ne nécessitent pas d'avoir une référence vers l'objet (mais juste de connaître le nom de la classe). Par exemple, le champ statique `f` défini dans la classe `C` est appelé par l'expression `C.f`. Un tel niveau d'accès est uniquement souhaité pour des constantes (par exemple les constantes mathématiques utilisées dans une application). Ce champ est alors partagé et accessible par toutes les instances de cette classe ou d'autres classes en fonction du modificateur de visibilité associé. Le champ fonctionne comme une variable partagée.

À noter que le modificateur `static` ne signifie pas qu'il est impossible de modifier la valeur du champ. Pour éviter la modification d'un champ, il faut le déclarer `final` [2].

Un champ déclaré `public static` s'apparente à une variable globale, parce qu'il est visible et accessible par toutes les classes.

Recommandations :

En raison de la faiblesse du contrôle d'accès à ces champs, il est en général déconseillé de stocker des données à protéger dans des champs statiques.

Déclarer les constantes (mathématiques ou autres) d'une application comme `static final` et effectuer leur initialisation lors de la déclaration.

Remplacer les autres champs statiques par des champs statiques privés avec des méthodes d'accès (*getters* et *setters*) qui contrôlent l'accès à cette variable.

Impacts potentiels sur le processus de développement :

Faible.

Références :

[1] `https://www.securecoding.cert.org/confluence/display/java/OBJ31-J.+Do+not+use+public+static+non-final+variables`

[2] `http://java.sun.com/security/seccodeguide.html` *Guideline 3-1 Treat public static fields as constants*

Voir également :

- Fiche 3, page 15
- Fiche 14, page 34

4.3 Sérialisation

Identifiant : 7	**Nom :** Sérialisation	**Risque :** E

Problématique :

Java offre la possibilité de construire une représentation concrète d'un objet sous forme d'une suite d'octets. Une telle sérialisation d'un objet permet d'améliorer la persistance d'une application et de communiquer des objets à travers un réseau.

Pour sérialiser un objet, il suffit que la classe de l'objet implémente l'interface `java.io.Serializable`. Cette interface est vide, elle ne contient aucune signature de méthodes, mais il est nécessaire de l'implémenter pour que la méthode `writeObject` de la classe `ObjectOutputStream` puisse créer la représentation binaire de l'objet. Par défaut, la sérialisation d'un objet contient tout l'état interne de l'objet. Il est possible d'indiquer explicitement qu'un champ ne doit pas faire partie de la sérialisation d'un objet en lui donnant le modificateur `transient`.

Du point de vue de la sécurité, la sérialisation rend accessible les données qui étaient encapsulées dans des objets (même les champs déclarés `private`) à travers sa représentation comme une suite d'octets. Pour accéder à ces champs, la méthode `writeObject` de la classe `ObjectOutputStream` n'utilise pas des *getters* qui peuvent protéger la donnée par un contrôle d'accès, mais elle utilise les mécanismes de la réflexion (voir Fiche 9, page 25).

Il faut également noter que si une classe `B` sérialisable hérite d'une classe `A` non-sérialisable, les champs hérités de `A` ne seront pas sérialisés, ce qui peut poser des problèmes sur les invariants de la classe.

Recommandations :

La possibilité de rendre une classe sérialisable ou désérialisable doit être utilisée avec parcimonie et doit être accompagnée d'une analyse des parties d'un objet à sérialiser.

Éviter que des champs avec des données confidentielles ou dont le besoin en intégrité est important apparaissent dans la sérialisation d'un objet, en leur donnant les modificateurs `transient`. Les champs `static` ne sont pas sérialisés, excepté le champ `serialVersionUID` servant à vérifier la version des classes lors de la désérialisation et qui doit être déclaré avec les modificateurs `final` et `static`, et être de type `long`.

Si un champ fait partie de la sérialisation d'un objet et si l'accès à sa valeur par un *getter* est protégé par un contrôle d'accès, la classe contenant le champ doit définir sa propre méthode `writeObject` et s'assurer que le même contrôle d'accès est effectué avant d'écrire cette valeur dans une représentation binaire.

Il est également possible de définir explicitement l'intégralité de la sérialisation d'un objet en implémentant l'interface `java.io.Externalizable` à la place de l'interface `java.io.Serializable`. Cette interface oblige la définition d'une méthode `public void writeExternal(ObjectOutput out) throws IOException` dans laquelle tout doit être sérialisé manuellement et où la réflexion n'intervient pas.

Il ne faut pas rendre les classes internes sérialisables. La sérialisation d'un objet dont la classe est une classe interne entrainera également la sérialisation de l'instance de la classe englobante qui lui est attachée.

Enfin, si une classe sérialisable est suceptible d'évoluer au sein d'une API, il est conseillé de définir la méthode `writeObject` afin de rester compatible avec les anciennes versions de sérialisation de la classe.

Impacts potentiels sur le processus de développement :

Modéré. Il faut analyser l'ensemble des classes pour lesquelles on veut pouvoir utiliser le mécanisme de sérialisation et redéfinir proprement la méthode `writeObject` de ces classes.

Références :

[1] `https://www.securecoding.cert.org/confluence/display/java/13.+Serialization+(SER)`

[2] `http://java.sun.com/security/seccodeguide.html` *5 Serialization and Deserialization*

Voir également :

- Fiche 8, page 23
- Fiche 9, page 25
- Fiche 23, page 53

Identifiant : 8	**Nom :** Désérialisation	**Risque :** E

Problématique :

À l'inverse de la sérialisation, la désérialisation en Java permet de reconstuire un objet à partir d'une suite d'octets obtenue par une sérialisation. Plus précisément, la classe `ObjectInputStream` propose une méthode `readObject` qui permet de récupérer une représentation interne de l'objet sérialisé à partir d'une suite d'octets. Par défaut, pour effectuer cette désérialisation, la méthode `readObject` de la classe `ObjectInputStream` utilise la réflexion.

La méthode `readObject` fournie par défaut n'effectue pas de vérification sur les invariants d'un objet reconstruit à partir d'un flux binaire (par exemple, si un objet représente bien une date ou une heure valide). Une classe sérialisable peut définir sa propre méthode `readObject` pour effectuer des vérifications supplémentaires.

La désérialisation permet à un attaquant de créer des objets par un moyen qui n'utilise pas les constructeurs de classes et qui, en conséquence, contourne les contrôles effectués au sein de ces constructeurs. De plus, la version par défaut de `readObject` ne vérifie pas que la suite d'octets correspond à la sérialisation d'un objet valable. Un attaquant peut ainsi forger une suite d'octets qui mène à la création d'objets partiellement initialisés. Pour réaliser une désérialisation, on appelle l'instruction suivante sur un objet `in` de type `ObjectInputStream` :

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

En exécutant cette instruction, on commence d'abord par désérialiser l'objet contenu dans la suite d'octets et ensuite, on vérifie si cet objet est du type attendu (ici `A`). Au sein de l'opération de désérialisation, un attaquant peut enregistrer l'objet désérialisé dans un champ static d'une classe même si celui-ci ne correspond pas à l'objet que l'on pensait désérialiser.

Enfin, il faut également noter que si on désérialise un objet dont la classe hérite d'une super-classe non-sérialisable, le constructeur sans argument de cette super-classe sera appelé pour initialiser les champs hérités de la super-classe, car ceux-ci n'ont pas été sérialisés.

Recommandations :

Éviter que des champs privés dans un objet obtenu par désérialisation ne contiennent des références issues de cette désérialisation. Il n'y a pas de garantie que ces champs privés ne donnent pas accès à des éléments partagés. On préférera travailler sur une copie de l'objet obtenu par désérialisation (voir fiche 12, page 30).

Traiter tout objet obtenu par désérialisation comme non-fiable et vérifier son état et les propriétés supplémentaires supposées être valables pour des objets de cette classe. **En particulier, répliquer les vérifications de permission effectuées dans les constructeurs (ou dans une méthode *setter*) dans la méthode `readObject`.**

Ne pas désérialiser un objet dans un contexte privilégié (voir Fiche 29, page 66). Cela peut permettre à la méthode `readObject` d'une classe malicieuse d'instancier des objets auxquels la classe n'aurait pas eu accès en dehors de ce contexte privilégié.

Comme pour la sérialisation, il est possible de définir explicitement l'intégralité de la désérialisation d'un objet en implémentant l'interface `java.io.Externalizable` à la place de l'interface `java.io.Serializable`. Cette interface nécessite de définir la méthode `public void readExternal(ObjectInput in) throws IOException` dans laquelle tout doit être désérialisé manuellement (il faut utiliser le même schéma que celui de la sérialisation réalisée dans la méthode `writeExternal`) et où la réflexion n'intervient pas. En utilisant cette interface, il faut également que la classe dispose d'un constructeur public sans argument.

Puisque la méthode `readExternal` est `public`, **il faut vérifier qu'elle n'est pas appelée sur un objet déjà initialisé ce qui écraserait les données d'un objet**. Pour ce faire, il suffit d'utiliser un drapeau d'initialisation comme présenté dans la fiche 13, page 32.

Enfin, si une classe sérialisable est susceptible d'évoluer au sein d'une API, il est conseillé de définir la méthode `readObject` afin de rester compatible avec les anciennes versions de sérialisation et désérialisation de la classe.

Impacts potentiels sur le processus de développement :

Modéré. Il faut analyser l'ensemble des classes pour lesquelles on veut pouvoir utiliser le mécanisme de sérialisation et redéfinir proprement la méthode `readObject` de ces classes.

Références :

[1] `https://www.securecoding.cert.org/confluence/display/java/13.+Serialization+(SER)`

[2] `http://java.sun.com/security/seccodeguide.html` *5 Serialization and Deserialization*

Voir également :

- Fiche 7, page 21
- Fiche 9, page 25
- Fiche 12, page 30
- Fiche 13, page 32
- Fiche 29, page 66

4.4 Programmation réflexive

Identifiant : 9	**Nom :** Programmation réflexive	**Risque :** F

Problématique :

La programmation réflexive permet à une application d'accéder et potentiellement de modifier son code lors de son exécution. Lors de l'exécution, Java garde une représentation de chaque classe comme un objet de classe `java.lang.Class`. Via des méthodes disponibles dans les classes `java.lang.Object`, `java.lang.Class` et dans toutes celles du *package* `java.lang.reflect`, une application peut entre autre :

- obtenir une classe, soit en récupérant celle d'un objet chargé (par un appel comme `o.getClass()`), soit en chargeant une classe par la méthode `Class.forname` qui prend en paramètre le nom d'une classe à charger et rend comme résultat l'objet de type `java.lang.Class` ;
- à partir d'un objet de type `java.lang.Class`, récupérer le nom, le *package*, les modificateurs apposés sur la classe, les interfaces implémentées et la superclasse étendue, la liste des champs (de type `java.lang.reflect.Field`), des méthodes (de type `java.lang.reflect.Method`), des constructeurs (de type `java.lang.reflect.Constructor`) et, depuis Java 1.5, des annotations (de type `java.lang.Annotation`) ;
- à partir des objets de type `Field`, `Method` et `Constructor`, obtenir les noms des membres d'une classe, le type des champs, la liste des paramètres d'une méthode et leur type, etc.
- instancier de nouveaux objets soit en utilisant la méthode `Class.newInstance()` qui va appeler le constructeur `public` et sans argument de la classe, soit à partir des objets de type `Constructor` en appelant leur méthode `newInstance(Object... initargs)` ;
- obtenir et modifier la valeur d'un champ (méthodes `get*` et `set*`) ou exécuter les méthodes à partir des objets de type `Method` en utilisant la méthode `invoke(Object obj, Object... args)` ;
- et surtout, suspendre les contrôles liés aux modificateurs de visibilité en utilisant la méthode `setAccessible` accessible depuis chacun des membres.

Un autre problème est que l'accès aux membres d'une classe par des méthodes réflexives peut contourner le contrôle d'accès effectué par inspection de pile parce que seule l'identité de l'appelant immédiat est vérifiée. Par exemple, un appel d'une méthode réflexive `java.lang.reflect.Field.set*` sur un objet `Field` va permettre de modifier la valeur d'un champ si l'appelant possède les droits d'effectuer cette modification, même si l'objet a été fourni par un code malveillant qui ne possède pas ces droits.

Il faut également noter qu'il est possible d'utiliser directement les classes du *package* `sun.reflect` qui fournissent les moyens de contourner les politiques de sécurité. De plus, selon les implémentations et la documentation de la bibliothèque standard, les vérifications faites par le vérificateur de *bytecode* ne s'appliquent à aucune classe héritant de la classe `sun.reflect.MagicAccessorImpl`. Ainsi, il est possible de créer une classe héritant de la classe `sun.reflect.MagicAccessorImpl` en déclarant que cette classe fait partie du *package* `sun.reflect` (pour contourner la visibilité (*package*) de la classe `sun.reflect.MagicAccessorImpl`).

Recommandations :

La programmation réflexive n'offre pas la vérification statique offerte par la programmation sans réflexion et doit donc être limitée au maximum. Le livre de Bloch [Jos08] en explique les inconvénients du point de vue du génie logiciel et propose des techniques de programmation qui permettent de l'éviter.

Ne pas appeler des méthodes réflexives sur des instances d'objets de provenance non sûre.

Mettre en place un gestionnaire de sécurité (fiches 23 et 24) dès le début de l'application et le configurer (fiches 25 et 27) pour interdire de passer outre les contrôles liés à la visibilité d'une classe ou d'un membre de classe, que ce soit pour des classes dont la provenance est sûre ou non. Le gestionnaire de sécurité permet également d'empêcher l'accès aux classes du *package* `sun.reflect` et donc d'hériter de la classe `sun.reflect.MagicAccessorImpl`.

Impacts potentiels sur le processus de développement :

Faible. Il faut inclure au sein de la politique de sécurité des règles qui empêchent le contournement de la visibilité des classes, champs et méthodes.

Références :

[1] `http://java.sun.com/security/seccodeguide.html` *Guideline 6-4 Be aware of standard APIs that perform Java language access checks against the immediate caller*

Voir également :

- Fiche 23, page 53
- Fiche 24, page 55
- Fiche 25, page 57
- Fiche 27, page 61

4.5 Exceptions

Identifiant : 10	**Nom :** Exceptions non levées sur dépassement de capacité	**Risque :** F

Problématique :

Le dépassement de capacité (*overflow*) des entiers est silencieux en Java. Un tel événement ne lève pas d'exception.

Le programmeur doit être conscient de ce phénomène. Un débordement ne peut *a priori* pas donner directement lieu à une vulnérabilité, car Java utilise une arithmétique modulaire. Un débordement étant généralement non souhaité par le programmeur, il peut compromettre les invariants internes d'un objet.

Combiné à l'utilisation de méthodes natives (écrites en C ou C++), un débordement peut indirectement donner lieu à une faille de sécurité [2]. C'est par exemple le cas si un débordement entraîne le passage d'un argument négatif à une méthode native qui attend implicitement une valeur positive et qui ne valide pas ses entrées. La méthode native peut alors avoir un comportement non prévu, donnant lieu à un débordement d'entier, voire de tampons.

Dans l'exemple suivant qui représente un cas de code non conforme, la méthode `main` effectue une itération qui n'est « censée » que faire croître l'entier (initialement positif) indiquant la taille d'un tableau à allouer dans la fonction native implémentée dans le fichier `IntegerOverflowExample.c`. Or, dans cette fonction native, il n'y a pas de test effectué sur la valeur de l'entier reçu qui doit être positif. L'exécution de ce programme entraînera un arrêt brutal de la JVM puisque la partie native tentera d'allouer un tableau de taille négative.

Fichier `IntegerOverflowExample.java` :

```
public class IntegerOverflowExample {
  ... // Chargement des fonctions natives

  // Méthode native allouant un tableau de taille size
  private native void allocate(int size);

  public static void main(String[] args) {
    IntegerOverflowExample ioe = new IntegerOverflowExample();
    // A chaque itération, le tableau est 4 fois plus grand
    for(int i = 0, size = 100000; i < 10; i++, size*=4)
      ioe.allocate(size);
  }
}
```

Fichier `IntegerOverflowExample.c` :

```
JNIEXPORT void JNICALL
  Java_jni_IntegerOverflowExample_allocate(JNIEnv * env,
      jobject obj, jint value)
{
  int[] tab = new int[size];
}
```

Recommandations :

L'utilisation de méthodes natives peut compromettre les propriétés intrinsèques du langage Java. Les paramètres entiers passés à ces méthodes doivent être validés, pour détecter un éventuel débordement.

Une analyse statique peut être utilisée pour assurer statiquement que toutes les manipulations numériques ne débordent jamais, si le débordement n'est pas souhaité.

Impacts potentiels sur le processus de développement :

Faible. Il est nécessaire de comprendre quelles sont les données valides attendues par les méthodes natives employées.

Le programmeur doit pouvoir aussi spécifier si un débordement arithmétique est souhaité ou considéré comme une erreur.

Références :

[1] `https://www.securecoding.cert.org/confluence/display/java/INT34-J.+Perform+explicit+range+checking+to+ensure+integer+operations+do+not+overflow`

[2] G. Tan and J. Croft. An Empirical Security Study of the Native Code in the JDK. In *U*SENIX Security 2008, July 2008, pages 365-377.

Voir également :

- Fiche 16, page 38
- Fiche 40, page 86

Identifiant : 11	**Nom :** Filtrage des exceptions	**Risque :** M

Problématique :

Les exceptions Java comportent des informations qui peuvent être confidentielles. Un mauvais filtrage de ces exceptions peut conduire à une fuite d'information au delà du périmètre de sécurité d'une application. C'est par exemple le cas des exceptions lancées en cas d'échec durant l'ouverture d'un fichier. L'exception lancée peut alors dévoiler un nom de fichier ou un chemin d'accès censé rester confidentiel.

Recommandations :

Une exception à contenu confidentiel doit être filtrée afin d'effacer tout contenu confidentiel. Il est donc nécessaire d'utiliser un bloc `try {...} catch {...}` pour rattraper les exceptions, éventuellement nettoyer leur contenu, puis afficher un message d'erreur ou relancer une nouvelle exception.

```
try {
  FileInputStream fis = new FileInputStream(...);
  ...
} catch(FileNotFoundException e) {
  System.err.println("erreur␣interne");
}
```

Impacts potentiels sur le processus de développement :

Modéré. Le programmeur doit pouvoir distinguer les exceptions comportant un contenu confidentiel.

Références :

[1] `http://java.sun.com/security/seccodeguide.html` *Guideline 3-4 Purge sensitive information from exceptions*

[2] `https://www.securecoding.cert.org/confluence/display/java/EXC01-J.+Do+not+allow+exceptions+to+transmit+sensitive+information`

[3] `https://www.securecoding.cert.org/confluence/display/java/EXC05-J.+Use+a+class+dedicated+to+reporting+exceptions`

Voir également :

- Fiche 17, page 40

4.6 Gestion de la mémoire

Identifiant : 12	**Nom :** Copie des objets	**Risque :** M

Problématique :

Lorsqu'un objet contient des données à protéger, et que celui-ci est potentiellement partagé, il peut être préférable de réaliser une copie. Trois cas sont à envisager :

- Une portion de code qui reçoit des objets en paramètre ; par exemple, dans le cas d'une classe immuable, un objet passé en paramètre d'un constructeur et servant à initialiser un des champs de la classe devra être copié si l'on veut être sûr de garantir l'immuabilité de la classe.
- Une portion de code qui communique un objet (valeur de retour d'une méthode) ; par exemple, une méthode de type *getter* d'une classe considérée comme immuable ne doit pas retourner l'objet référencé par son champ, mais une copie de celui-ci pour être sûr que la méthode qui a invoqué ce *getter* ne modifie pas l'état interne du champ.
- Un objet obtenu par désérialisation. Lorsqu'un objet est le résultat d'une désérialisation, il n'y a pas de garantie que les adresses contenues dans cet objet ne soient pas accessibles (voir [2]) par ailleurs. Il peut donc être nécessaire, là aussi, de réaliser une copie.

Le langage Java met à disposition de l'utilisateur un mécanisme de copie appelé clonage. Selon les cas, par l'utilisation du mécanisme de clonage de Java, une copie en surface ou en profondeur sera transmise au (ou reçu du) tiers.

Recommandations :

Lorsqu'un objet peut contenir des données confidentielles ou ayant un besoin d'intégrité important, il est préférable de travailler sur une copie (en surface ou en profondeur selon les cas). Celle-ci peut permettre de masquer certaines informations et d'éviter des interactions non souhaitées.

Lorsque l'on veut garantir l'immuabilité d'une classe, il faut réaliser une copie profonde (de préférence) des objets passés au constructeur et servant à initialiser les champs de la classe et également des objets communiqués à l'extérieur de la classe.

Impacts potentiels sur le processus de développement :

Modéré. Il peut s'avérer nécessaire de redéfinir certaines méthodes de clonage, mais l'impact reste localisé.

Références :

[1] `https://www.securecoding.cert.org/confluence/display/java/MSC37-J.+Make+sensitive+classes+noncloneable`

[2] `http://java.sun.com/security/seccodeguide.html` *Guideline 2-1 Create a copy of mutable inputs and outputs*

Voir également :

- Fiche 8, page 23

Identifiant : 13	**Nom :** Objets partiellement créés	**Risque :** M

Problématique :

Un attaquant peut accéder à des objets partiellement créés si une exception est lancée lors de la construction de l'objet. En particulier, cela pourrait permettre de contourner des contrôles de sécurité.

De manière générale, un objet (son adresse) ne devrait pas être « visible » hors de son constructeur tant que celui-ci n'a pas terminé normalement (sans levée d'exception) et retourné l'adresse de cet objet à l'appelant.

En pratique, si le constructeur d'une classe réalise des appels à des méthodes pouvant être surchargées, il est possible en héritant de cette classe de redéfinir ces méthodes et de faire échapper l'adresse de l'objet avant que le constructeur ne termine et ce indépendamment des vérifications de sécurité.

Une situation plus problématique concerne la redéfinition de la méthode `finalize` par l'attaquant. En effet, même si une exception de sécurité est levée et que l'objet n'est pas totalement initialisé, la méthode `finalize` peut être appelée par le ramasse-miettes. La fuite de l'objet partiellement initialisé peut alors être réalisée dans cette méthode. Interdire la surcharge de la méthode `finalize`, contrairement aux méthodes appelées par le constructeur, peut être problématique. Le comportement de cette méthode dépend effectivement beaucoup des modifications apportées par l'héritage de la classe de base. Le même type de problème peut se produire avec les méthodes de clonage et de désérialisation si celles-ci appellent des méthodes pouvant être surchargées.

Recommandations :

Éviter qu'un constructeur n'appelle des méthodes pouvant être surchargées. Les méthodes appellées par un constructeur devraient donc être soit `private` soit `final`. La même recommandation s'applique aux méthodes appelées par les méthodes `clone` et `readObject`.

Pour les classes considérées comme ayant un impact important sur la sécurité, on pourra se référer à la recommandation de [2] qui suggère d'utiliser un « drapeau » d'initialisation pour représenter l'état d'initialisation de l'objet et conditionner son utilisation. Le drapeau sera de la forme :

```
private volatile boolean initialized = false;
```

et ne devra passer à la valeur `true` qu'à la fin de l'exécution de son constructeur. Toutes les méthodes pouvant être utile à un attaquant et qui pourraient être appelées à partir de la méthode `finalize` devront alors commencer par vérifier la valeur de ce drapeau avant toutes exécutions.

Note : ces recommandations n'empêchent pas encore totalement l'échappement d'objets partiellement initialisés. Une recommandation d'évolution de la JVM est proposée dans [Con09e] pour renforcer ce point.

Impacts potentiels sur le processus de développement :

Faible. Il suffit d'associer aux méthodes appelées par un constructeur les bons modificateurs. Dans le pire des cas, s'il n'est pas possible d'appliquer les bons modificateurs sur les méthodes appelées par le constructeur, il faudrait alors créer une méthode privée contenant le code à réaliser. Cette méthode serait celle appelée directement par le constructeur, et la méthode publique qui pourrait alors être surchargée ne ferait qu'appeler cette méthode privée.

Références :

[1] `https://www.securecoding.cert.org/confluence/display/java/OBJ32-J.+Do+not+allow+partially+initialized+objects+to+be+accessed`

[2] `http://java.sun.com/security/seccodeguide.html` *Guideline 4-2 : Defend against partially initialized instances of non-final classes*

Voir également :

- Fiche 3, page 15
- Fiche 8, page 23
- Fiche 12, page 30

Identifiant : 14	**Nom :** Désallocation et effacement	**Risque :** M

Problématique :

Sous Java, la gestion de la mémoire dynamique est réalisée automatiquement. En particulier, la désallocation est réalisée implicitement par le mécanisme de « ramasse-miettes » (ou *garbage collector*) de la JVM. De manière générale, l'interface avec le mécanisme de gestion de la mémoire est limitée en Java, ce qui permet de réduire les erreurs de gestion de la mémoire par le développeur, mais peut s'avérer problématique pour la gestion des données confidentielles (par exemple, une clé cryptographique). Se pose notamment le problème de la persistance des données dans la mémoire de l'application (dès lors qu'elles ne sont plus nécessaires) et de la rémanence de ces données dans la mémoire (une fois l'application terminée).

Recommandations :

Le déchargement des classes étant difficile à mettre en œuvre, le stockage de données confidentielles au sein des classes (champs déclarés `static`, constantes, etc.) n'est pas recommandé.

Dans la plupart des cas, le développeur ne doit pas interagir avec le mécanisme de gestion de la mémoire. La libération de la mémoire doit être favorisée uniquement par la libération des références sur les objets à effacer.

Pour les données nécessitant un effacement sécurisé, il peut être nécessaire de définir une méthode d'effacement sécurisée. Cette méthode est notamment appelée à la destruction de l'objet à l'aide de la méthode `public void finalize()`. L'utilisation d'une telle méthode permet d'espérer [3] que les champs de l'objet contenant des données confidentielles (clés cryptographiques, etc.) seront bien remis à zéro avant sa libération par le *garbage collector*. Toutefois, seules les valeurs numériques contenues dans les types primitifs (`int`, `byte`, `long`, etc.) peuvent êtres effacées de manière efficace. Pour les types référence (tableaux, objets, classes), il est nécessaire d'itérer de manière récursive sur les différents champs de type primitif accessibles depuis la référence (par exemple, les différents éléments d'un tableau en utilisant éventuellement la méthode `java.util.Arrays.fill`). Il n'est pas possible d'appliquer cette recommandation pour les champs correspondant à des objets immuables (par exemple, les chaînes de caractères de type `String`).

L'implémentation de Sun de la classe `com.sun.crypto.provider.DESKey` illustre l'application de cette recommandation.

3. La spécification de la JVM et les implémentations existantes ne permettent pas de garantir l'absence de persistance des données confidentielles. En effet, outre le problème des objets immuables, certains GC dit « générationnels » peuvent être amenés à copier un objet dans différentes zones de la mémoire. Seule la dernière copie peut être effacée via les techniques évoquées ici. En outre, le développeur ne peut s'assurer, de manière fine, que certains objets ne seront pas copiés sur le fichier ou la partition d'échange (SWAP). Seul l'utilisateur peut restreindre l'utilisation de ce mécanisme, et ce pour l'ensemble de l'application.

Pour limiter les problématiques de persistance, il convient d'appeler la méthode d'effacement au plus tôt, dès lors que la donnée n'est plus utilisée. Il n'est en revanche pas recommandé de forcer l'étape de libération de la mémoire grâce à un appel à la méthode `System.gc()`. En effet, ceci peut avoir un impact sur les performances. De plus, le développeur doit garder à l'esprit que le comportement du *garbage collector* dépend de l'implémentation de la JVM et que l'effacement n'est pas garanti lors du retour de la méthode `System.gc()`.

Impacts potentiels sur le processus de développement :

Modéré. Le développeur doit identifier les données confidentielles nécessitant un traitement adéquat (utilisation d'une méthode d'effacement). Les autres types de données doivent être libérés au plus tôt en s'assurant qu'aucune référence ne pointe sur l'objet à libérer.

Références :

[1] [Jam05] *Chapter 12 - Execution, Section 7 - Unloading of Classes and Interfaces*

[2] `http://java.sun.com/javase/6/docs/api/java/lang/System.html#gc()`

[3] `http://java.sun.com/javase/6/docs/api/java/util/Arrays.html`

[4] `https://www.securecoding.cert.org/confluence/display/java/MSC31-J.+Never+hardcode+sensitive+information`

Voir également :

- Fiche 6, page 20

4.7 Problématique de la décompilation

Identifiant : 15	**Nom :** Décompilation	**Risque :** F

Problématique :

Les applications Java sont généralement déployées sous forme de *bytecode* contenu dans des fichiers `class`. Le *bytecode* est un langage de niveau intermédiaire entre le code machine et le langage source Java. Le fossé sémantique entre ce dernier et le *bytecode* reste faible. Il est donc relativement aisé de décompiler le *bytecode* et de reconstruire en grande partie le code source original, les fichiers de classes contenant beaucoup plus d'information sur le code source original que le code machine exécuté par la plate-forme native. Des outils comme Mocha [3], JAD [4] ou JOD [5] permettent d'automatiser le processus. L'utilisation de ce type d'outil peut constituer une menace dès lors que des informations confidentielles (algorithme protégé, mot de passe, etc.) peuvent être retrouvées.

De manière générale, le développeur doit être conscient que toute personne disposant du *bytecode* de l'application peut aisément retrouver le code source. En particulier, les hypothèses suivantes doivent être faites :

- le flux de contrôle de l'application peut être déterminé par une personne ayant accès au *bytecode* ;
- tous les champs, y compris ceux dont la visibilité est restreinte, peuvent potentiellement être lus par une personne ayant accès au *bytecode*.

Recommandations :

Afin de limiter les risques liés à la décompilation, il convient de s'assurer que le *bytecode* déployé en environnement de production ne contient pas d'information de mise au point. La commande suivante illustre cette recommandation pour le compilateur de Sun : `javac -g:none <program_name>`.

Pour les mêmes raisons, il convient de ne pas laisser en clair des informations à caractère confidentiel (par exemple les clés de chiffrement).

S'il est nécessaire d'embarquer de telles informations, les approches suivantes peuvent être mises en œuvre :

- utiliser des techniques d'offuscation de *bytecode* ;
- chiffrer les classes et utiliser un *class loader* approprié.

Toutefois, ces techniques ne constituent pas un moyen de protection sûr contre la rétro-conception, dès lors qu'elles ne dépendent pas d'un élément secret non présent en clair dans le code. Elles permettent généralement de se protéger des outils automatiques mais un attaquant peut, en utilisant des techniques *ad hoc*, accéder aux informations protégées.

Impacts potentiels sur le processus de développement :

Faible.

Références :

[1] `https://www.securecoding.cert.org/confluence/display/java/SEC06-J.+Assume+that+all+Java+clients+can+be+reverse+engineered%2C+monitored%2C+and+modified`

[2] [Kal04] *2, Decompiling classes ; 3, Obfuscating Classses et 19, Protecting Commercial Applications from Hacking*

[3] `http://www.brouhaha.com/~eric/software/mocha/`

[4] `http://mguessan.free.fr/java/jad/index.html`

[5] `http://jode.sourceforge.net/`

Voir également :

- *Sans objet.*

4.8 Gestion des entrées/sorties

Identifiant : 16	**Nom :** Vérification des données en entrée	**Risque :** E

Problématique :

Les données d'entrée d'une application constituent un vecteur d'attaque potentiel. En effet, un attaquant peut utiliser les entrées de l'application pour y injecter des données susceptibles de provoquer un comportement anormal (dénis de service, exécution de code).

L'environnement d'exécution Java effectue automatiquement certaines vérifications (variable d'index d'accès à un tableau inférieur à la taille du tableau, typage des données, etc.) ce qui limite le spectre des attaques (notamment celles entraînant la corruption de la mémoire). Toutefois, certaines vulnérabilités ne sont pas traitées, en particulier lorsque des données sont fournies à du code natif via JNI, ce dernier n'étant pas soumis aux contrôles évoqués précédemment. De plus, aucune vérification n'est effectuée par défaut afin de détecter les « dépassements » d'entiers (*integer overflow*). Lorsqu'une variable entière désigne la taille d'un tableau passé en paramètre à du code natif, ce type de vulnérabilité peut faciliter l'exploitation d'un débordement de tampon dans le code natif.

De manière générale, le problème se pose lorsque des données en entrée, organisées suivant une structure complexe, sont fournies à un bloc fonctionnel qui analyse et interprète ces données. C'est par exemple le cas des données interprétées en tant que requête SQL, de l'analyse des fichiers XML ou de données qui sont utilisées pour forger une commande exécutée sur le système.

Recommandations :

Ce type d'attaque n'est possible que si l'attaquant arrive à exploiter des cas particuliers de données d'entrée non prévues par la routine de traitement. Par conséquent, le développeur doit prévoir tous les cas possibles dans la routine de traitement des données d'entrée. Les cas jugés non-conformes pour l'utilisation du produit doivent être rejetés et faire l'objet d'un traitement d'erreur approprié (par exemple, lever une exception).

Le développeur doit être particulièrement vigilant sur les points suivants :

- la taille des données d'entrée ;
- la valeur des paramètres qui sont directement utilisés par l'application pour accéder à des ressources (indice de tableau, nom de fichier, paramètre de requêtes, etc.) ou par des mécanismes critiques (désérialisation, appel de code natif via JNI, chargement de classes, etc.)

Impacts potentiels sur le processus de développement :

Faible.

Références :

[1] `http://www.owasp.org/index.php/OWASP_Java_Table_of_Contents#Input_Validation_Overview`

Voir également :

- Fiche 10, page 27
- Fiche 40, page 86

Identifiant : 17	**Nom :** Filtrage des données affichées	**Risque :** E

Problématique :

L'affichage des données confidentielles (notamment les mots de passe saisis par l'utilisateur) à l'écran constitue une fuite possible d'informations.

Recommandations :

Le programme ne doit pas comporter de code permettant l'affichage de données confidentielles.

La saisie de données confidentielles par l'utilisateur ne doit pas s'afficher à l'écran. Dans le cas d'une application graphique, le développeur doit utiliser une fenêtre de saisie permettant de masquer les frappes de clavier. Il peut utiliser par exemple :

- la classe `javax.swing.JPasswordField` ;
- la classe `java.awt.TextField` avec la méthode `setEchoChar` qui permet de configurer ce qui est affiché à chaque frappe.

Concernant la saisie dans une application de type console, Java 1.6 introduit une méthode (`char[] java.io.Console.readPassword()`) permettant à l'utilisateur de saisir son mot de passe sans afficher les caractères à l'écran. Le code suivant montre comment utiliser cette méthode :

```
Console console = System.console();
private char[] passwd = console.readPassword();
```

Enfin, concernant la saisie dans une application de type console pour un environnement d'exécution Java 1.5 ou antérieur, il n'existe pas de moyens simples fournis par la bibliothèque standard permettant d'éviter l'affichage des caractères saisis par l'utilisateur. L'article présenté à cette adresse [1] montre une méthode permettant de résoudre ce problème. Il est également possible d'utiliser un mécanisme natif via JNI.

Impacts potentiels sur le processus de développement :

Faible.

Références :

[1] `http://java.sun.com/developer/technicalArticles/Security/pwordmask/`

[2] `http://java.sun.com/javase/6/docs/api/java/io/Console.html`

Voir également :

- Fiche 11, page 29

4.9 Utilisation correcte du langage et de la bibliothèque

Identifiant : 18	**Nom :** Ambiguïté sur le nom des classes	**Risque :** F

Problématique :

Certaines classes possèdent un nom « court » identique bien que leur nom complet soit distinct. En effet, il est possible de définir des noms de classes identiques pour des *packages* différents. Ceci peut parfois mener à des ambigüités si le développeur utilise le nom « court » des classes qu'il utilise.

Par exemple, la classe `Certificate` est sujette à une ambigüité susceptible de provoquer des erreurs à la compilation. En effet, dans Java 1.1, une interface appelée `java.security.Certificate` a été introduite et utilisée par les utilitaires `javakey` et `appletviewer`. Dans Java 2, une nouvelle classe appelée `java.security.cert.Certificate` a été introduite afin de traiter toutes les interactions avec les certificats. Depuis, `java.security.Certificate` est devenue obsolète. Enfin, dans JSSE, la classe `javax.security.cert.Certificate` apparaît. L'utilisation simultanée des imports `import java.security.*`, `import java.security.cert.*` et `import javax.security.cert.*` peut provoquer des erreurs à la compilation si la classe `Certificate` est utilisée.

Outre les problèmes d'incompatibilité et d'erreur de compilation, une désignation ambigüe d'une classe peut favoriser le piégeage d'une application (chargement et utilisation d'une classe différente de celle souhaitée par le développeur).

Recommandations :

Afin d'éviter toute ambigüité, le développeur doit utiliser le nom complet de la classe.

Impacts potentiels sur le processus de développement :

Faible.

Références :

[1] [Oak01] Section 9, Keys and Certificates

Voir également :

- *Sans objet.*

5 RECOMMANDATIONS RELATIVES À LA BIBLIOTHÈQUE STANDARD

5.1 Gestion des *threads* et programmation concurrente

Afin de s'assurer que son programme offre un comportement prévisible, le programmeur Java doit s'assurer que son programme est correctement synchronisé. Cette tâche est en grande partie manuelle et le langage Java n'offre que peu de facilités pour garantir cette propriété de manière automatique.

Le programmeur Java doit donc :

- soit s'assurer que son programme est correctement synchronisé en utilisant les outils du langage conçus à cet effet ;
- soit renoncer à l'utilisation des *threads*.

Dans le cas d'une application nécessitant une forte sûreté de fonctionnement, la programmation concurrente ne devrait être utilisée que si un gain indispensable en termes de performances est clairement identifié, ou si la modélisation du concept à traiter se résout naturellement sous forme de *threads*.

Si l'utilisation des *threads* se révèle indispensable, les échanges *inter-threads* doivent être clairement définis et maîtrisés.

Les recommandations suivantes ne sont pas toutes au même niveau. Certaines d'entre elles ne prennent leur sens qu'en fonction de la prise en compte ou non d'autres recommandations de plus haut niveau. Le diagramme suivant présente les dépendances entre recommandations. Un losange représente une recommandation qui doit être étudiée. Si cette recommandation est respectée, la branche « Oui » doit être suivie, et si elle n'est pas respectée, c'est la branche « Non » qui doit l'être.

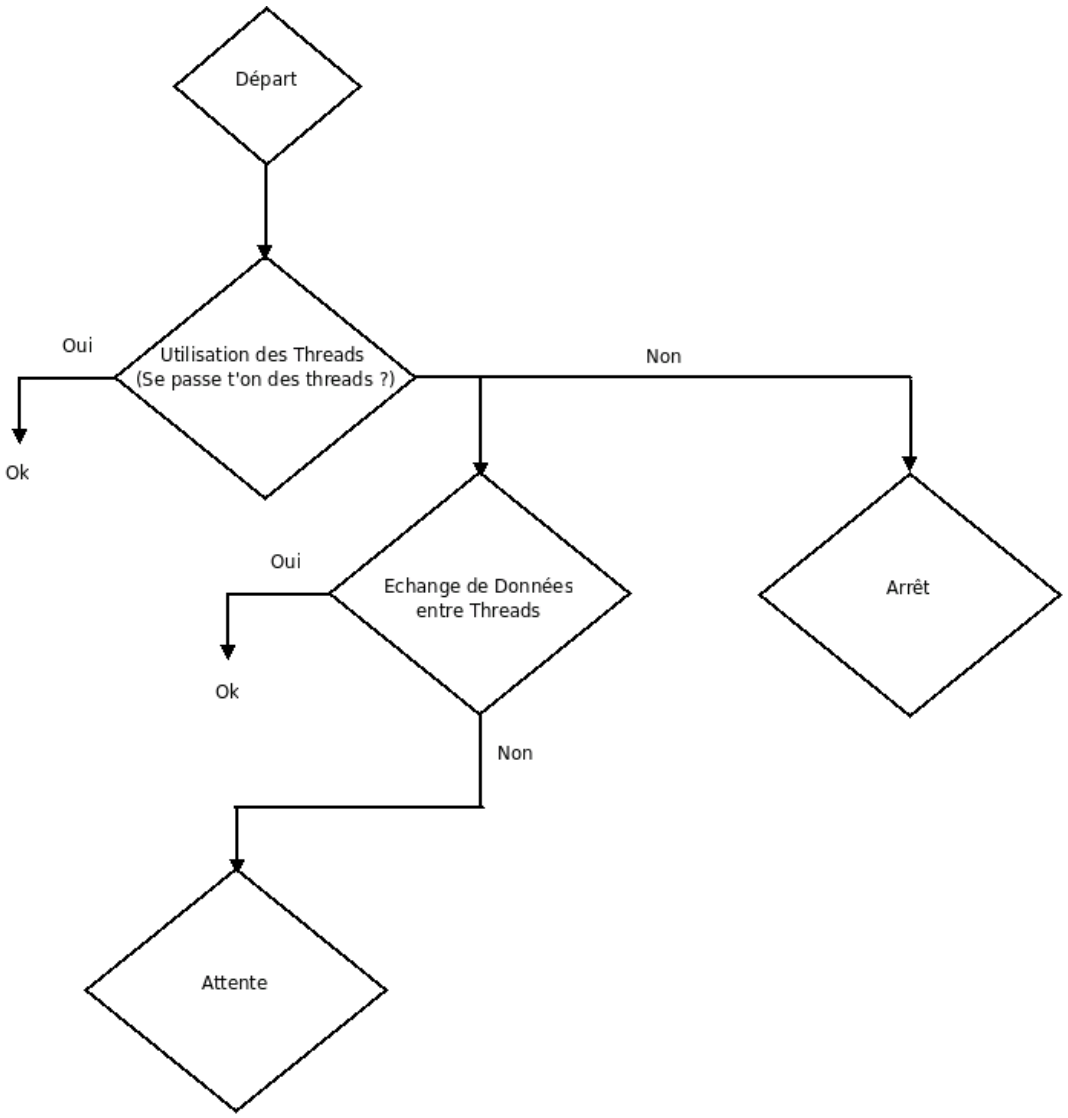

FIGURE 1 – Cartographie des recommandations relatives aux *threads* Java et à la programmation concurrente

Identifiant : 19	**Nom :** Utilisation des *threads*	**Risque :** M

Problématique :

Sans prise de dispositions particulières, l'écriture d'un programme concurrent correct en Java impose de fortes contraintes sur le développeur Java. En effet, en plus d'avoir à s'assurer que son programme est algorithmiquement correct, le programmeur doit en plus s'assurer que son programme est correctement synchronisé.

Recommandations :

L'utilisation des *threads* Java est dangereuse, et dans l'idéal devrait être évitée. La phase de conception doit faire apparaître quel est le besoin d'utilisation des *threads* dans l'architecture logicielle choisie, et devrait en limiter l'utilisation aux sous-ensembles en ayant vraiment besoin (dans l'interface homme-machine par exemple).

À titre d'exemple, l'utilisation du package `java.nio` permet de se passer de l'utilisation de *threads* dans les cas où l'on doit gérer des entrées/sorties sur plusieurs fichiers, *sockets* ou tubes nommés (*pipe*) simultanément. A cette fin `java.nio` fournit un mécanisme d'entrées sorties non-bloquantes, permettant le multiplexage de plusieurs sources d'entrées sorties dans un seul thread. Cette méthode a l'avantage d'être plus performante (pas de coût annexe de création/destruction de threads), et l'inconvénient d'être un peu plus lourde à l'usage qu'une implémentation basée sur les threads.

Un cas d'utilisation implémentant cette technique est présenté dans les exemples 1 et 2, donnés en annexe (page 95).

Impacts potentiels sur le processus de développement :

Modéré. L'impact porte principalement sur la phase de conception.

Références :

[1] `http://java.sun.com/javase/6/docs/api/java/util/concurrent/package-summary.html`

Voir également :

- Fiche 20, page 45

Identifiant : 20	**Nom :** Echange de données entre *threads*	**Risque :** M

Problématique :

Il existe de multiples mécanismes en Java permettant les échanges de données *inter-threads*. Certains mécanismes sont plus sûrs d'utilisation que d'autres, car ils n'imposent pas au développeur de gérer manuellement la synchronisation *inter-threads*.

Recommandations :

Tous les échanges *inter-threads* devraient être faits en utilisant des mécanismes « automatiques », c'est-à-dire qui ne nécessitent pas de gestion manuelle de la synchronisation de la part du développeur. Dans la mesure du possible, on devrait privilégier l'utilisation des *Concurrent collection* de Java 1.5 en lieu et place de l'utilisation du mot-clé `synchronized` ou l'utilisation des classes de verrous manuels tels que `Lock` et `ReadWriteLock`. Un exemple de code basé sur l'API Swing et conforme avec cette recommandation est donné en annexe page 99.

Impacts potentiels sur le processus de développement :

Faible. L'impact porte sur la phase d'implémentation. Il n'y a pas de surcoût *a priori* lié à la prise en compte de cette recommandation, si ce n'est la prise de connaissance de la part du développeur des nouvelles possibilités de Java 1.5.

Références : S.O.

Voir également :

- *Sans objet.*

Identifiant : 21	**Nom :** Attente	**Risque :** M

Problématique :

La méthode `wait()` de la classe `java.lang.Object` est sujette à des réveils aléatoires sur certaines JVM [4], c'est-à-dire qu'elle rend parfois la main sans que les méthodes `notify()` ou `notifyAll()` n'aient été appelées. De plus, rien n'empêche un *thread* mal intentionné d'essayer de réveiller un autre *thread* de manière non sollicitée.

Recommandations :

La méthode `wait()` doit toujours être utilisée dans une boucle comme présenté ci-dessous. Dans cet exemple, C est la condition de réveil du *thread*.

```
synchronized(obj) {
  while (<C est faux>) {
    obj.wait();
  }
}
... // Faire les actions nécessaires quand C est avérée
```

Impacts potentiels sur le processus de développement :

Faible.

Références :

[1] `http://java.sun.com/javase/6/docs/api/java/lang/Object.html#wait()`

Voir également :

- *Sans objet.*

4. Il s'agit en fait d'un détail technique d'implémentation des JVM qui a été rémonté jusque dans la spécification, probablement dû au fait que la fonction utilisée sous UNIX pour implémenter le `wait()` peut rendre la main de façon prématurée.

Identifiant : 22	**Nom :** Arrêt	**Risque :** M

Problématique :

Java proposait originellement une fonction destinée à arrêter un *thread* qui était en cours d'exécution, ainsi que des fonctions pour suspendre/reprendre l'exécution d'un *thread*.

Il fut découvert plus tard que ces APIs n'étaient pas sûres de fonctionnement. Par conséquent, elles ne devraient jamais être utilisées.

Recommandations :

Les méthodes `stop()`, `suspend()` et `resume()` de la classe `java.lang.Thread` sont marquées comme étant obsolètes (*deprecated*) dans le JDK. Elles ne devraient jamais être utilisées.

Corollaire : l'unique méthode appropriée pour arrêter un *thread* est de lui demander de s'arrêter en utilisant un mécanisme de communication *inter-threads*. Dans le cas où le *thread* est bloqué sur un appel de méthode Java, il est également parfois possible de l'arrêter en utilisant la méthode `interrupt()`.

Impacts potentiels sur le processus de développement :

Faible.

Références :

[1] `http://java.sun.com/javase/6/docs/api/java/lang/Thread.html`

[2] `http://java.sun.com/javase/6/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html`

[3] `https://www.securecoding.cert.org/confluence/display/java/CON35-J.+Do+not+try+to+force+thread+shutdown`

Voir également :

- *Sans objet.*

5.2 Utilisation du contrôle d'accès

La plate-forme d'exécution Java fournit un service de contrôle d'accès orienté code (JPSA). Ce mécanisme permet de s'assurer que le code exécuté possède les permissions nécessaires pour réaliser certaines opérations jugées critiques (accès à une ressource native, utilisation d'un mécanisme critique, interaction avec le mécanisme de contrôle d'accès, etc.).

Ce contrôle est assuré principalement par du code Java exécuté par la JVM. L'architecture de JPSA est illustrée par la figure 2. Cette figure distingue trois types d'éléments :

- les éléments de spécification de la politique de contrôle (en orange) ;
- les éléments d'identification et d'authentification du code (en vert) ;
- les éléments d'application du contrôle (en rouge).

Les principales classes de contrôle d'accès sont les suivantes :

- le gestionnaire de sécurité (`java.lang.SecurityManager`), qui est appelé par les différents points de contrôle ;
- le contrôleur d'accès (`java.security.AccessController`), dont l'implémentation par défaut du gestionnaire de sécurité délègue les vérifications ;
- les permissions, qui permettent de définir des autorisations (accès à une ressource native, exécution d'une méthode critique, etc.) ;
- le gestionnaire de la politique (`java.security.Policy`), qui fournit au contrôleur d'accès l'ensemble des permissions associées à une classe. L'implémentation par défaut s'appuie sur des fichiers texte permettant à l'utilisateur de spécifier la politique de contrôle.

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'utilisation du gestionnaire de sécurité peut paraître redondante (au vue des fonctionnalités offertes par le contrôleur d'accès), toutefois celui-ci a été maintenu pour deux raisons :

- il assure la compatibilité descendante (le contrôleur d'accès n'ayant été introduit qu'à partir de la version 2 de Java) ;
- il permet de modifier la mise en œuvre du contrôle d'accès. En effet, il n'est pas possible de modifier le contrôleur d'accès en définissant une classe héritant de la classe `java.security.AccessController`, celle-ci étant déclarée `final`.

Il est possible d'allouer différentes instances de gestionnaire de sécurité. Toutefois, à un instant donné, une seule de ces instances est active (l'instance active étant celle qui est appelée lors des vérifications de contrôle d'accès). Pour modifier le gestionnaire de sécurité, c'est-à-dire spécifier quelle est l'instance active, le code applicatif doit posséder la permission `RuntimePermission("setSecurityManager")` (l'installation de la première instance ne nécessite pas de droit particulier, puisque le contrôle d'accès n'est pas effectif à ce moment).

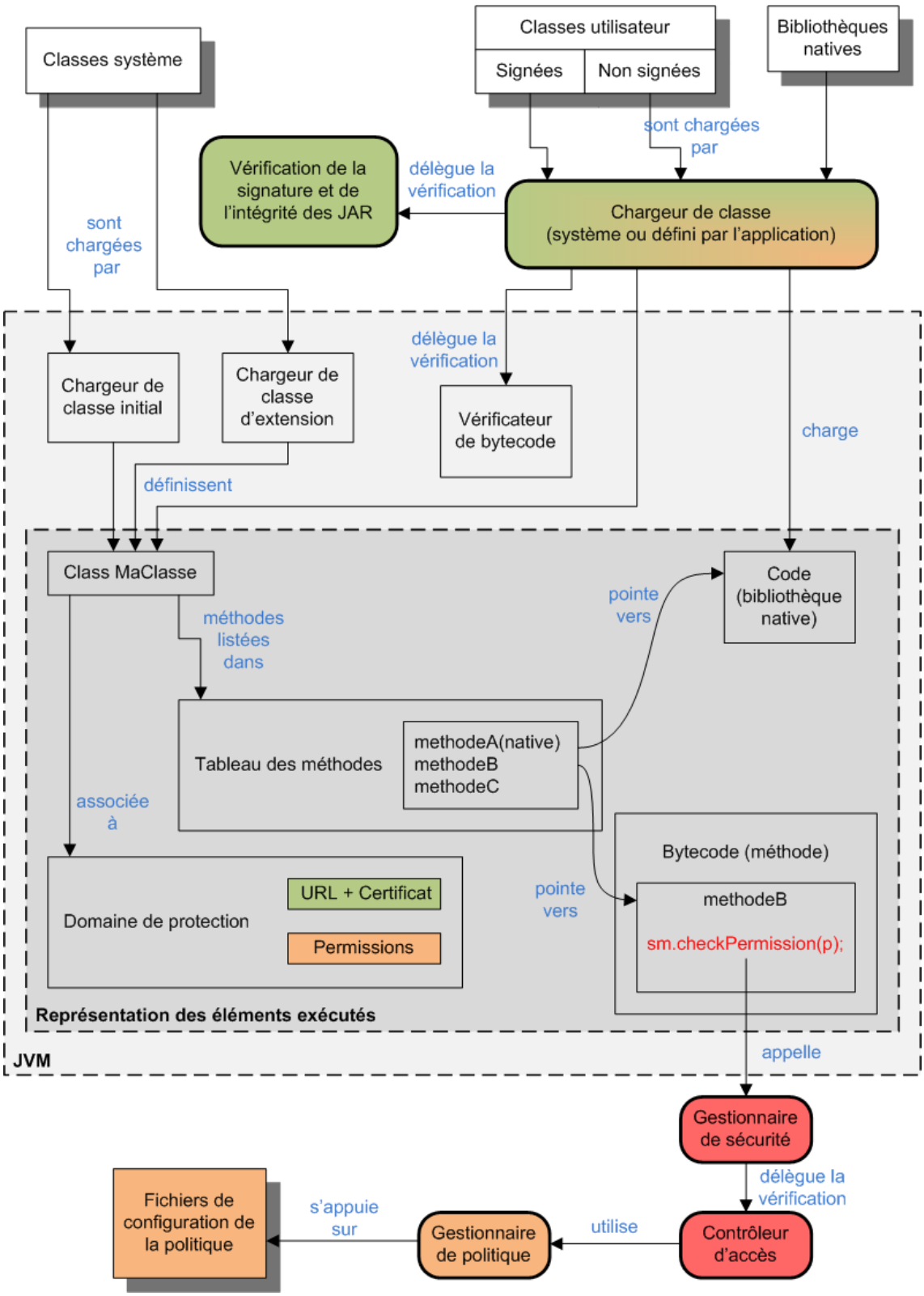

FIGURE 2 – Architecture générique du système de contrôle d'accès de Java.

Il en va de même pour le gestionnaire de politique. Le fournisseur de l'environnement d'exécution (JRE) propose généralement une implémentation par défaut de cette classe. Par exemple, le gestionnaire de politique par défaut de l'environnement d'exécution fourni par Sun est implémenté par la classe `sun.security.provider.PolicyFile` qui repose sur la spécification de la politique à l'aide de fichiers au format ASCII. Il est également possible d'utiliser d'autres implémentations, permettant de mettre en œuvre une politique spécifique ou reposant sur d'autres supports de spécification (bases de données, fichiers XML, etc.). Il est possible d'allouer différentes instances du gestionnaire de la politique de sécurité. Toutefois, à un instant donné, une seule de ces instances est active (l'instance active étant celle qui est appelée lors des vérifications de contrôle d'accès).

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 chaque méthode présente sur la pile d'exécution courante du *thread* lors de la vérification possède la permission de réaliser l'accès qui fait l'objet du point de contrôle. 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 par défaut peut être modifié par l'utilisation des actions privilégiées qui limitent le contrôle à un sous-ensemble de la pile d'exécution.

Considérons, par exemple, une application Java dont la classe principale est `MonAppli`. La méthode `lectureFichier` de cette classe tente d'accéder en lecture à un fichier du système (instanciation d'un objet de type `FileInputStream`). Supposons qu'une exécution de cette application conduise à la pile d'appels de méthodes illustrée par le tableau 23 (cette pile correspond au point de contrôle implémenté dans le constructeur de `FileInputStream`)).

Classe	Méthode	Domaine de protection
`AccessController`	`getStackAccessControlContext`	`null` (DS)
`AccessController`	`checkPermission`	`null` (DS)
`SecurityManager`	`checkPermission`	`null` (DS)
`SecurityManager`	`checkRead`	`null` (DS)
`FileInputStream`	`FileInputStream`	`null` (DS)
`MonAppli`	`lectureFichier`	DA
`MonAppli`	`Main`	DA

TABLE 23: Exemple de pile d'appels

La méthode `getStackAccessControlContext` renvoie un objet `AccessControlContext` qui englobe l'ensemble des domaines de protection associé à chaque classe des méthodes de la pile d'appel. Dans cet exemple, il est possible de distinguer deux types de domaines de protection :

– les classes système n'ont pas d'instance de domaine de protection associée (le do-

maine vaut `null`), ce qui caractérise le domaine de protection système, auquel est associé implicitement la protection `AllPermission` ;

– la classe `MonAppli` est associée à un domaine DA.

Supposons que la politique de sécurité soit la suivante :

```
grant codebase "file:/home/user/MonAppli*" {
permission java.io.FilePermission "/tmp/*", "read, write";
};
```

Supposons que le domaine de protection DA soit associé (par le chargeur de la classe `MonAppli`) à l'URL `/home/user/MonAppli`. L'algorithme de contrôle d'accès parcourt l'ensemble des domaines de protection de la pile d'exécution et détermine, pour chacun d'eux, les permissions associées via la politique de sécurité. Ici, seul le domaine de protection DA est associé explicitement à une permission (`java.io.FilePermission("/tmp/*", "read, write")`). Par la suite, l'algorithme vérifie si l'ensemble des permissions associé à chaque domaine de protection implique la permission vérifiée (ici, `java.io.FilePermission(<nom_fichier "read")`).

Si l'application tente d'accéder à un fichier du répertoire `/tmp/` (par exemple `/tmp/exemple.txt`), l'accès sera autorisé. En effet, la permission `java.io.FilePermission("/tmp/*", "read, write")` implique la permission `java.io.FilePermission("/tmp/exemple.txt", "read")`. Les accès en dehors de ce répertoire sont en revanche refusés et provoquent la levée d'une exception de sécurité.

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 [5]) ;

– 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 malveillantes (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

5. 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.

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. Toutefois, la modification de ces éléments n'est pas sans risque.

Les recommandations suivantes portent donc sur le paramétrage et la mise en place du contrôle d'accès Java ainsi que sur la modification des éléments du contrôle d'accès.

Identifiant : 23	**Nom :** Installation du gestionnaire de sécurité	**Risque :** E

Problématique :

Le gestionnaire de sécurité constitue la pièce centrale du mécanisme de contrôle d'accès de Java. Cette classe permet de mettre en œuvre une politique de sécurité en autorisant ou non les accès aux ressources natives et aux méthodes sensibles du système. Elle implémente les méthodes qui sont appelées par les différents points de contrôle. Toutefois, aucun gestionnaire de sécurité n'est installé par défaut pour les applications Java locales, à la différence des *applets*.

Recommandations :

Un gestionnaire de sécurité doit être installé pour toute application Java. L'installation du gestionnaire de sécurité peut être réalisée de deux manières :

- le gestionnaire de sécurité peut être spécifié dans la ligne de commande permettant d'exécuter l'application (par exemple, `java -Djava.security.manager MonAppli`);
- le gestionnaire de sécurité peut être installé par l'application elle-même via l'appel de la méthode `setSecurityManager` de la classe `java.lang.System`.

Le développeur doit indiquer le type de gestionnaire de sécurité que l'utilisateur doit utiliser en le spécifiant dans la ligne de commande Java.

Afin de s'assurer qu'un gestionnaire de sécurité approprié est mis en place, il est conseillé d'effectuer les traitements suivants au sein d'une méthode d'initialisation exécutée au début de l'application :

- obtenir une référence sur l'instance active du gestionnaire de sécurité (celle appelée par les points de contrôle) via un appel à la méthode `System.getSecurityManager`;
- vérifier le type du gestionnaire de sécurité si l'application nécessite un gestionnaire de sécurité spécifique :
  - si aucun gestionnaire de sécurité n'est installé ou si le type est incorrect, tenter d'installer une instance correcte de gestionnaire de sécurité via la méthode `System.setSecurityManager`,
  - en cas d'échec d'installation d'un gestionnaire de sécurité approprié, quitter l'application.

Il peut s'avérer nécessaire d'octroyer à ce code d'initialisation la permission `RuntimePermission("setSecurityManager")` si ce dernier est susceptible de modifier le gestionnaire de sécurité installé au préalable. Il convient dans ce cas d'isoler le code d'initialisation (par exemple au sein d'une classe dédiée) afin d'appliquer les principes de minimum de privilège et de séparation des tâches.

En dehors de ce code d'initialisation, la plupart des applications ne nécessitent pas de modifier le gestionnaire de sécurité utilisé lors de l'exécution. Il est dans ce cas impératif de s'assurer qu'aucune classe (en dehors du code d'initialisation) ne possède la permission de modifier le gestionnaire de sécurité.

Impacts potentiels sur le processus de développement :

Faible.

Références :

[1] `https://www.securecoding.cert.org/confluence/display/java/SEC30-J.+Always+use+a+Security+Manager`

[2] [Gon03] Section *6.1, Security Manager*

[3] `http://java.sun.com/javase/6/docs/api/java/lang/SecurityManager.html`

[4] `http://java.sun.com/javase/6/docs/api/java/lang/System.html`

Voir également :

- Fiche 24, page 55
- Fiche 25, page 57
- Fiche 26, page 59
- Fiche 27, page 61
- Fiche 28, page 63

Identifiant : 24	**Nom :** Utilisation du gestionnaire de sécurité	**Risque :** F

Problématique :

La mise en œuvre de règles de contrôle d'accès spécifiques (par exemple, des règles dépendant de la date ou de l'heure de l'accès) justifie parfois l'implémentation d'un gestionnaire de sécurité spécifique à l'aide d'une classe héritant de la classe `java.lang.SecurityManager`. Toutefois, l'implémentation d'un gestionnaire de sécurité ou la modification de l'algorithme par défaut sont des tâches délicates qui peuvent compromettre l'ensemble du mécanisme de contrôle d'accès implémenté par la classe `java.security.AccessController`.

En outre, lorsque le développeur d'application est amené à implémenter des points de contrôle, il peut être tenté d'appeler directement le contrôleur d'accès au lieu du gestionnaire de sécurité.

Recommandations :

Depuis la version 2 de Java, il n'est plus nécessaire de redéfinir les méthodes du gestionnaire de sécurité pour mettre en œuvre une politique de contrôle d'accès. Le développeur privilégiera donc la modification de la politique de sécurité (fichiers de configuration de la politique ou éventuellement implémentation d'une classe héritant de `java.security.Policy`). L'implémentation d'un gestionnaire de sécurité *ad hoc* doit être limitée aux cas qui le nécessitent expressément, comme par exemple :

- la mise en œuvre d'une forme de contrôle d'accès différente de celle implémentée par le contrôleur d'accès ;
- la modification de l'algorithme implémenté par le contrôleur d'accès.

Le dernier cas est le plus courant. Il s'agit par exemple de relâcher ou de contraindre les vérifications effectuées par le contrôleur d'accès en fonction du contexte (par exemple, la date et l'heure de l'accès). Le développeur devra s'assurer dans ce cas que les modifications apportées s'intègrent au mieux dans le mécanisme natif de contrôle d'accès. En particulier, l'implémentation du gestionnaire de sécurité doit, dans la mesure du possible, déléguer la vérification au contrôleur d'accès.

Les appels directs aux méthodes du gestionnaire d'accès sont à proscrire sauf dans le cas spécifique où le contrôle doit porter sur un contexte d'exécution différent du contexte courant. Ce cas de figure correspond notamment à la vérification d'un accès réalisé par le *thread* courant qui nécessite de prendre en compte le code exécuté par d'autres *threads*. Ce type d'appel doit être utilisé uniquement en cas de stricte nécessité, car il contourne les vérifications additionnelles éventuellement mises en œuvre par le gestionnaire de sécurité.

Le développeur privilégiera la forme générique de contrôle reposant sur l'utilisaton de la méthode `checkPermission()` et l'utilisation ou la définition d'une permission *ad-hoc*. Les méthodes du gestionnaire de sécurité implémentant un type de vérification particulier (par exemple, `checkRead()`) sont présentes pour des raisons de compatibilité mais font elles-mêmes appel à la méthode `checkPermission()`.

L'exemple suivant illustre l'implémentation d'un point de contrôle lors de l'accès en lecture à un fichier :

```
try {
  SecurityManager sm = System.getSecurityManager();
  if(sm != null) {
    sm.checkPermission( new FilePermission("file1", "read"));
  }
}
catch (SecurityException se) {
  System.out.println("Not allowed");
}
```

Impacts potentiels sur le processus de développement :

Faible.

Références :

[1] https://www.securecoding.cert.org/confluence/display/java/SEC30-J.+Always+use+a+Security+Manager

[2] [Gon03] Section *6.1, Security Manager*

[3] http://java.sun.com/javase/6/docs/api/java/lang/SecurityManager.html

[4] http://java.sun.com/javase/6/docs/api/java/security/package-summary.html

[5] http://java.sun.com/j2se/1.5.0/docs/guide/security/spec/security-spec.doc6.html#28195

Voir également :

- Fiche 23, page 53
- Fiche 28, page 63
- Fiche 27, page 61

Identifiant : 25	**Nom :** Mise en place du gestionnaire de la politique de contrôle	**Risque :** E

Problématique :

Le gestionnaire de la politique de contrôle d'accès fournit l'ensemble des permissions associées aux différentes classes susceptibles d'être chargées par la JVM.

Le choix de l'implémentation du gestionnaire de la politique de sécurité utilisé par le mécanisme de contrôle d'accès est un paramètre important qu'il convient de spécifier ou de vérifier. Le développeur et l'administrateur responsable du déploiement de l'application doivent notamment s'assurer qu'un attaquant ne puisse spécifier un gestionnaire de politique qui lui soit favorable, afin de contourner le contrôle d'accès.

Recommandations :

L'implémentation du gestionnaire de la politique de sécurité utilisé par le mécanisme de contrôle d'accès de Java doit être spécifiée et paramétrée explicitement. Lorsqu'il est réalisé depuis la plate-forme d'exécution native, ce paramétrage doit être protégé par un mécanisme adéquat (contrôle d'accès de l'OS, etc.).

L'instance de la politique utilisée par le système de contrôle d'accès peut être spécifiée de deux manières :

- le fichier de configuration `<JAVA_HOME>/lib/security/java.security` (avec `<JAVA_HOME>` le répertoire d'installation du JRE utilisé) permet de spécifier, via la propriété de sécurité `policy.provider`, la classe du gestionnaire de la politique utilisée par défaut ;
- lors de l'exécution, le code de l'application Java peut modifier l'instance de la politique utilisée à l'aide de la méthode statique `setPolicy` de la classe `java.security.Policy`. Pour cela, le code de l'application doit posséder la permission `SecurityPermission("setPolicy")`.

Le développeur doit indiquer le type de gestionnaire de politique que l'utilisateur doit spécifier au sein du fichier de configuration (`<JAVA_HOME>/lib/security/java.security`).

Afin de s'assurer qu'un gestionnaire de politique approprié est mis en place [6], il est conseillé d'effectuer les traitements suivants au sein d'une méthode d'initialisation exécutée au début de l'application :

- obtenir une référence sur l'instance active du gestionnaire de politique (celle appelée par les points de contrôle) via un appel à la méthode `java.security.Policy.getPolicy` ;

6. Pour l'implémentation de Sun, un gestionnaire de politique est toujours activé. Par défaut, en l'absence de paramétrage, il s'agit d'une instance de la classe `sun.security.provider.PolicyFile`.

- vérifier le type du gestionnaire de politique si l'application nécessite un gestionnaire de politique spécifique :
  - si le type est incorrect, tenter d'installer une instance correcte de gestionnaire de politique via la méthode `System.setSecurityManager`,
  - en cas d'échec d'installation d'un gestionnaire de politique approprié, quitter l'application.

Il est nécessaire d'octroyer les permissions `SecurityPermission("getPolicy")` et `SecurityPermission("setPolicy")` à ce code d'initialisation. Ces permissions doivent être spécifiées dans la politique de contrôle. Il convient dans ce cas d'isoler le code d'initialisation (par exemple au sein d'une classe dédiée) afin d'appliquer les principes de minimum de privilège et de séparation des tâches.

En dehors de ce code d'initialisation, la plupart des applications ne nécessitent pas de modifier le gestionnaire de politique utilisé lors de l'exécution (ni même d'obtenir une référence sur cet objet). Il est dans ce cas impératif de s'assurer qu'aucune classe (en dehors du code d'initialisation) ne possède la permission d'obtenir ou de modifier le gestionnaire de politique.

Impacts potentiels sur le processus de développement :

Faible.

Références :

[1] `https://www.securecoding.cert.org/confluence/display/java/SEC30-J.+Always+use+a+Security+Manager`

[2] [Gon03] Section *6.1, Security Manager*

[3] `http://java.sun.com/javase/6/docs/api/java/security/Policy.html`

[4] `http://java.sun.com/javase/6/docs/api/java/security/package-summary.html`

Voir également :

- Fiche 23, page 53
- Fiche 26, page 59
- Fiche 27, page 61

Identifiant : 26	**Nom :** Paramétrage de la politique de contrôle	**Risque :** E

Problématique :

La politique de contrôle d'accès spécifie les permissions qui sont associées aux différentes classes susceptibles d'être chargées par la JVM. Certaines implémentations du gestionnaire de la politique de sécurité permettent de préciser, depuis l'environnement d'exécution natif, les supports utilisés pour spécifier et paramétrer la politique. Par exemple, l'implémentation par défaut fournie par Sun repose sur des fichiers de configuration au format texte ASCII.

Le choix des supports (fichiers, etc.) de spécification de la politique de sécurité utilisée par le mécanisme de contrôle d'accès est un paramètre important qu'il convient de spécifier ou de vérifier. Le développeur et l'administrateur responsable du déploiement de l'application doivent notamment s'assurer que les supports utilisés pour spécifier la politique (par exemple, des fichiers texte) sont effectivement pris en compte par le gestionnaire de la politique utilisée par le système de contrôle d'accès.

Recommandations :

L'emplacement des supports de spécification de la politique de sécurité doit être configuré explicitement. L'accès à ces supports doit être protégé à l'aide de mécanismes adéquats (contrôle d'accès, etc.) au niveau de la plate-forme d'exécution native.

Pour l'implémentation de Sun, la localisation de ces fichiers de configuration peut être spécifiée de deux manières :

- dans le fichier de configuration `<JAVA_HOME>/lib/security/java.security` à l'aide des propriétés de sécurité `policy.url.i` ($i \in \mathbb{N}$) ;
- à l'aide d'une option de la ligne de commande (exemple : `-Djava.security.policy=somefile`) si la propriété `policy.allowSystemProperty` est positionnée à `true` dans le fichier de configuration précédent.

Si la même spécification de la politique de sécurité doit s'appliquer sur les différentes applications susceptibles d'être exécutées sur le système, il est préférable de spécifier l'URL du fichier de spécification de la politique via le fichier de configuration en s'assurant que ce paramètre ne peut être modifié par la ligne de commande. Dans le cas contraire, la politique peut être ajustée à l'aide de l'option `java.security.policy` de la ligne de commande.

Impacts potentiels sur le processus de développement :

Faible.

Références :

[1] `https://www.securecoding.cert.org/confluence/display/java/SEC30-J.+Always+use+a+Security+Manager`

[2] [Gon03] Section *6.1, Security Manager*

[3] `http://java.sun.com/javase/6/docs/api/java/security/Policy.html`

[4] `http://java.sun.com/javase/6/docs/api/java/security/package-summary.html`

Voir également :

- Fiche 23, page 53
- Fiche 25, page 57
- Fiche 27, page 61

Identifiant : 27	**Nom :** Spécification de la politique de contrôle	**Risque :** E

Problématique :

La politique de contrôle d'accès spécifie les permissions qui sont associées aux différentes classes susceptibles d'être chargées par la JVM. Elle consiste à attribuer des permissions aux différentes classes en fonction de leur origine exprimée en termes d'URL ou de certificat du fournisseur (pour les classes signées).

L'étape de spécification de la politique est cruciale :

- un attaquant peut profiter d'une politique trop permissive ;
- une politique trop restrictive peut perturber le fonctionnement des applications en interdisant des accès légitimes.

Recommandations :

Une politique de contrôle d'accès adaptée aux applications susceptibles d'être exécutées doit être spécifiée et fournie à l'utilisateur.

Cette spécification consiste à associer des permissions (donc des autorisations d'accès à des méthodes sensibles ou à des ressources du système) en fonction de l'origine du code. Les recommandations suivantes doivent être suivies lors de cette étape :

- appliquer les principes de minimum de privilèges et de séparation des tâches en octroyant à chaque classe les permissions qui lui sont strictement nécessaires ;
- limiter l'utilisation de la permission `AllPermission` qui correspond au code privilégié. Octroyer uniquement cette permission aux classes de confiance signées par un fournisseur de confiance ;
- utiliser conjointement les URL et les certificats (et éventuellement les *principals* de JAAS) pour associer les permissions aux différentes classes. Les certificats permettent de différencier les classes en fonction de leurs fournisseurs (ou de vérifier qu'il s'agit bien de classes légitimes) tandis que l'URL permet de différencier les différents blocs fonctionnels (si le découpage en termes de classes/*packages* a été réalisé correctement). Utiliser uniquement les certificats conduit à attribuer des privilèges excessifs (une classe, même si elle provient d'un fournisseur de confiance, peut être utilisée par un attaquant ou un code malveillant) ;

– tenir compte des permissions impliquées et des effets de bords. La relation `implies` définit les permissions impliquées au niveau de chaque permission. Il existe également une deuxième forme d'implication liée aux effets de bords de certaines permissions. Par exemple, la permission de modifier l'intégralité des fichiers de la plate-forme native implique potentiellement la possibilité de modifier le fichier de spécification de la politique. Cette permission implique donc indirectement toutes les permissions si aucune mesure de protection supplémentaire n'est prise au niveau de l'OS.

Impacts potentiels sur le processus de développement :

L'impact est élevé. Le développeur doit analyser les besoins de chacune des classes de l'application afin de définir l'ensemble minimum des permissions qui doit être attribué à chaque classe.

Références :

[1] [Gon03] Section *6.1, Security Manager*

[2] `http://java.sun.com/javase/6/docs/api/java/lang/SecurityManager.html`

[3] `http://java.sun.com/javase/6/docs/api/java/security/package-summary.html`

[4] `http://java.sun.com/j2se/1.5.0/docs/guide/security/spec/security-spec.doc6.html#28195`

[5] `http://java.sun.com/javase/6/docs/technotes/guides/security/permissions.html`

Voir également :

- Fiche 23, page 53
- Fiche 25, page 57
- Fiche 26, page 59

Identifiant : 28	**Nom :** Points de contrôle	**Risque :** M

Problématique :

Le mécanisme de contrôle d'accès de Java repose sur la spécification explicite de points de contrôle dans les méthodes Java réalisant des accès sensibles (appels de méthodes sensibles, accès aux ressources du système, etc.). Les classes de la bibliothèque standard implémentent un certain nombre de ces points de contrôle afin de restreindre les accès aux ressources de la plate-forme d'exécution native (par exemple, l'accès aux fichiers via la classe `FileInputStream`) ou aux méthodes sensibles (par exemple, celles permettant de modifier le gestionnaire de sécurité). Les applications et bibliothèques additionnelles peuvent également comprendre des méthodes sensibles. Afin de restreindre l'accès à ces méthodes sensibles, le programmeur doit spécifier explicitement des points de contrôle additionnels.

Recommandations :

Les méthodes de l'application implémentant des opérations sensibles (accès aux ressources natives, chargement de bibliothèques, etc.) doivent faire l'objet de points de contrôle implémentés explicitement par le développeur. Lorsque cela s'avère nécessaire, le développeur peut être amené à définir une permission adéquate permettant de mettre en œuvre un contrôle pertinent avec l'accès réalisé.

L'exemple suivant illustre l'implémentation d'un point de contrôle lors de l'accès à une ressource native (un tube relié à un processus fils). L'accès est réalisé par la méthode native `nativePipeOpen`. Ce point de contrôle repose sur une permission `IOPermission` définie par le développeur de l'application.

```
public NativePipe pipeOpen(String command) throws
    FileNotFoundException {
  SecurityManager sm = System.getSecurityManager();
  if (sm != null) {
    sm.checkPermission(new IOPermission("OpenPipe"));
  }
  if (command == null) {
    throw new NullPointerException();
  }
  return nativePipeOpen(command);
}
```

Ce code s'assure de la présence d'un gestionnaire de sécurité avant l'appel à `checkPermission` ce qui est impératif. Si aucun gestionnaire de sécurité n'est installé, le contrôle n'est pas effectué. Cet exemple suppose que les recommandations concernant l'installation d'un gestionnaire de sécurité soient respectées. Il est souhaitable que l'application vérifie ce point. Toutefois, il paraît préférable que cette vérification soit effectuée au début de l'application et non pour chaque point de contrôle afin d'éviter les redondances.

La définition d'une nouvelle permission repose sur l'implémentation d'une classe héritant de la classe abstraite `java.security.Permission`. Il est notamment nécessaire de définir la méthode `implies` qui définit la sémantique de la permission [7]. Plusieurs recommandations doivent être prises en compte lors de la définition d'une telle classe :

- la redéfinition de permissions (notamment celles fournies par la bibliothèque standard) doit être évitée en raison des effets de bords sur l'algorithme de contrôle d'accès (contraintes supplémentaires sur l'implémentation de la méthode `implies`, etc.) ;
- les classes implémentant les permissions doivent être déclarées `final` ;
- la factorisation de code, partagé par plusieurs classes définissant des permissions, doit être réalisée à l'aide d'une classe abstraite dont héritent ces différentes classes. La classe `java.security.BasicPermission` de la bibliothèque standard constitue un exemple d'une telle classe abstraite ;
- il peut être nécessaire d'implémenter également une classe `PermissionCollection` suivant la complexité de l'algorithme du contrôle de la permission.

Impacts potentiels sur le processus de développement :

L'impact est modéré. Le développeur doit identifier les accès nécessitant l'implémentation de points de contrôle. Il peut être amené à implémenter des permissions spécifiques pour réaliser le contrôle.

Références :

[1] `http://java.sun.com/javase/6/docs/technotes/guides/security/permissions.html`

[2] `http://java.sun.com/javase/6/docs/api/java/security/package-summary.html`

[3] `http://java.sun.com/security/seccodeguide.html#3-3`

[4] `https://www.securecoding.cert.org/confluence/display/java/SEC30-J.+Define+wrappers+around+native+methods`

[5] `https://www.securecoding.cert.org/confluence/display/java/SEC08-J.+Define+custom+security+permissions+for+fine+grained+security`

7. En effet, cette méthode est appelée par le contrôleur d'accès afin de s'assurer que les permissions associées au code « impliquent » la permission vérifiée.

Voir également :

- Fiche 23, page 53
- Fiche 24, page 55

Identifiant : 29	**Nom :** Méthodes privilégiées	**Risque :** M

Problématique :

La méthode statique `doPrivileged` permet d'invoquer des méthodes privilégiées pour lesquelles les permissions des méthodes appelantes sont ignorées par le mécanisme de contrôle d'accès. Seules les permissions des méthodes privilégiées sont prises en compte, permettant ainsi aux méthodes des applications d'accéder à des ressources pour lesquelles elles ne possèdent pas d'autorisation, via l'appel de méthodes privilégiées.

Dans ce contexte, il existe un risque d'élévation de privilège pour le code appelant. Celui-ci peut utiliser le code privilégié pour accéder librement à des ressources auxquelles il n'a pas accès directement si ce dernier n'implémente pas de filtrage correct sur les données échangées entre le code appelant et le code privilégié. Il s'agit d'une problématique similaire à celle résultant de l'utilisation du bit SUID pour le mécanisme de contrôle d'accès discrétionnaire des systèmes de type UNIX. L'exemple 4 de code non conforme donné en annexe (page 102) illustre cette problématique.

Recommandations :

Comme indiqué dans la documentation de l'API Java [1], une attention toute particulière doit être portée sur l'utilisation de la méthode `doPrivileged`. Le développeur doit notamment s'assurer que les règles suivantes sont respectées :

- le code non privilégié de l'application ne doit pas pouvoir spécifier directement la ressource lue (par exemple en spécifiant l'URL de la ressource via une variable globale ou un paramètre). Le cas échéant, le code privilégié doit appliquer un filtrage sur l'URL spécifiée (afin par exemple de limiter les répertoires ou les domaines lus) ;
- le code privilégié ne doit pas fournir directement, au code non privilégié de l'application, des références sur les ressources auxquelles il accède ;
- de manière générale, l'ensemble des opérations exécutées de manière privilégiée (suite à un appel à la méthode `doPrivileged`) doit être limité au strict nécessaire. Toute opération ne nécessitant pas d'être exécutée de manière privilégiée doit être extraite du bloc d'instructions exécuté à l'aide de `doPrivileged`, excepté les cas où ce retrait contredit une des deux règles précédentes ;
- si des exceptions générées par le code privilégié doivent être transmises au code de l'application, il est nécessaire de s'assurer qu'elles ne contiennent pas d'informations confidentielles. Les exceptions doivent le cas échéant être traitées au sein du code privilégié ou être filtrées pour éviter toute fuite d'information ;
- l'ensemble des vérifications décrites précédemment doivent êtres fournies par la même classe que celle fournissant le code privilégié (ou dans une classe mère). L'accès au code privilégié doit obligatoirement provoquer l'exécution des méthodes implémentant les vérifications, afin d'éviter les contournements (par exemple, en déclarant le code privilégié `private`).

Les exemples 5 et 6 de code conforme donnés en annexe (page 103) illustrent la prise en compte de ces règles.

Impacts potentiels sur le processus de développement :

Faible.

Références :

[1] `https://www.securecoding.cert.org/confluence/display/java/SEC31-J.+Guard+doPrivileged+blocks+against+untrusted+invocations`

[2] `http://java.sun.com/javase/6/docs/api/java/security/AccessController.html#doPrivileged(java.security.PrivilegedAction)`

[3] [Gon03] Sections *6.4, AccessController* et *9.5 Privileged Code*

Voir également :

- Fiche 30, page 69
- Fiche 1, page 12

Identifiant : 30	**Nom :** Contournement du contrôle d'accès	**Risque :** M

Problématique :

Certaines méthodes de l'API standard de Java peuvent, suivant le contexte, contourner les vérifications effectuées par défaut par le mécanisme de contrôle d'accès. Par exemple, l'appel de la méthode `newInstance` de la classe `java.lang.Class<T>` nécessite en théorie de vérifier que l'ensemble du code appelant possède la permission d'accéder au *package* de la classe en question. Toutefois, lorsque `newInstance` est invoquée sur une classe `C` au sein d'une méthode `m`, cette vérification n'est pas effectuée dans les cas suivants :

- si le *class loader* de la classe de `m` est identique à celui de `C` ;
- si le *class loader* de la classe de `m` est un ancêtre du *class loader* de `C` ;
- si la classe de `m` ou `C` sont des classes système (chargées par le *bootstrap class loader*).

Contrairement à l'algorithme de contrôle d'accès classique implémenté par la classe `java.security.AccessControler`, la vérification sur le *class loader* ne prend en compte que la méthode appelante directe et non l'intégralité des méthodes de la pile d'exécution du *thread* courant. Implicitement, cela signifie que l'ensemble des méthodes qui passent le test du *class loader* avec succès contournent la vérification du contrôle d'accès. Ces méthodes constituent donc implicitement des méthodes privilégiées (sans qu'il soit nécessaire de faire appel à la méthode `doPrivileged`). Ces méthodes doivent donc s'assurer qu'elles ne peuvent être utilisées par du code non privilégié afin de contourner le mécanisme de contrôle d'accès.

Les méthodes suivantes ont un comportement similaire à `Class.newInstance` et peuvent potentiellement être utilisées pour contourner le contrôle d'accès :

- `java.lang.Class.newInstance;`
- `java.lang.Class.getClassLoader;`
- `java.lang.Class.getClasses;`
- `java.lang.Class.getField(s);`
- `java.lang.Class.getMethod(s);`
- `java.lang.Class.getConstructor(s);`
- `java.lang.Class.getDeclaredClasses;`
- `java.lang.Class.getDeclaredField(s);`
- `java.lang.Class.getDeclaredMethod(s);`
- `java.lang.Class.getDeclaredConstructor(s);`

- `java.lang.ClassLoader.getParent`;
- `java.lang.ClassLoader.getSystemClassLoader`;
- `java.lang.Thread.getContextClassLoader`.

Recommandations :

Ces méthodes ne doivent être invoquées que pour des instances de classes `Class`, `ClassLoader` ou `Thread` fournies par du code de confiance. De plus, il est recommandé de filtrer les données utilisées lors de l'appel de ces méthodes si ces dernières ne proviennent pas de code de confiance. De même, il est nécessaire de s'assurer que les références sur les instances de classes obtenues à l'aide de ces méthodes ne sont propagées que vers du code de confiance.

Impacts potentiels sur le processus de développement :

Faible.

Références :

[1] `https://www.securecoding.cert.org/confluence/display/java/SEC02-J.+Do+not+expose+standard+APIs+that+may+bypass+Security+Manager+checks+to+untrusted+code`

[2] `http://java.sun.com/security/seccodeguide.html#gcg`

[3] [Gon03] Section *4.3.2, Class Loader Delegation Hierarchy*

Voir également :

- Fiche 29, page 66
- Fiche 42, page 93

Identifiant : 31	**Nom :** Signature de classes	**Risque :** F

Problématique :

Le format d'archive JAR est courament utilisé lors du déploiment d'applications Java. Il s'agit d'un format d'archive compressée dérivé du format ZIP qui supporte le contrôle d'intégrité et la signature multiple des différents fichiers de l'archive. Celle-ci comprend notamment des fichiers *class* regroupés sous forme de *packages*.

Le fournisseur de l'application peut ainsi déployer ou mettre à jour tout ou partie de l'application à l'aide d'archives JAR. Le mécanisme de signature peut être utilisé pour se prémunir des risques de piégeage de l'application dans la chaîne de distribution en permettant de vérifier l'intégrité et l'authenticité du code fourni.

Recommandations :

Le déploiement et la mise à jour des applications devraient privilégier l'utilisation du mécanisme de signatures intégrées aux archives JAR. Cela implique de mettre en œuvre les étapes suivantes :

- le fournisseur doit signer les classes de l'archive JAR qui le concernent, par exemple à l'aide de l'outil `jarsigner` ;
- la plate-forme d'exécution doit mettre en place les vérifications de la signature des classes. Cela suppose notamment que les différents *class loader* utilisés par l'application implémentent la vérification de signatures et que le contrôle d'accès tienne compte des signatures de classes. La politique de sécurité doit par exemple imposer des restrictions sur les classes non signées. Les chargeur de classe doivent interdire le chargement de classes dont la signature n'est pas valide.

Cela implique également que l'utilisateur mette en place une gestion correcte des certificats (éventuellement via une infrastructure de clés publiques ou PKI).

Toutefois, si la plate-forme n'utilise pas de code mobile, ce service peut être implémenté par un mécanisme de la plate-forme native (par exemple, un outil du système d'exploitation).

La signature peut porter sur les différents contenus de l'archive, dont les fichiers `class`. Elle porte également sur les méta-données associées à chaque fichier de l'archive et qui sont présentes dans le fichier `MANIFEST.MF`. En revanche, certaines méta-données, telles que le nom de l'archive JAR, ne sont pas protégées.

Impacts potentiels sur le processus de développement :

L'impact est modéré. Le développeur de l'application principale doit s'assurer que le chargement de classes implémente de manière correcte la vérification des signatures et mettre en place une gestion des certificats. Le fournisseur doit signer les archives JAR.

Références :

[1] `https://www.securecoding.cert.org/confluence/display/java/SEC06-J.+Sign+and+seal+sensitive+objects+before+transit`

[2] [Oak01] Chapitre *12 : Digital Signatures, Signed Classes*

[3] [Gon03] Section *12.8.3 : jarsigner*

[4] `http://java.sun.com/docs/books/tutorial/security/sigcert/index.html`

[5] `http://wiki.eclipse.org/JAR_Signing`

[6] *Common Weakness Enumeration* (`http://cwe.mitre.org/`) : CWE-347, CWE-494

Voir également :

- Fiche 27, page 61

Identifiant : 32	**Nom :** Utilisation de JAAS	**Risque :** M

Problématique :

L'API JAAS complète le contrôle d'accès orienté code de Java (JPSA). Elle permet d'identifier et d'authentifier les utilisateurs d'une application Java et d'accorder des permissions en fonction de l'utilisateur qui exécute le code de l'application.

Toutefois, ce mécanisme est fortement dépendant du mécanisme de contrôle d'accès utilisé par défaut et implémenté par la classe `java.security.AccessControler`. Il est donc nécessaire de respecter certaines recommandations afin de s'assurer du bon fonctionnement des services proposés par l'API.

Recommandations :

Lorsque l'application nécessite d'authentifier des utilisateurs, il est recommandé d'utiliser l'API JAAS. Celle-ci offre en effet une interface standard permettant d'utiliser différents moyens d'authentification (il s'agit d'un mécanisme similaire à PAM pour les OS de type UNIX). Ce mécanisme peut également être utilisé pour restreindre les privilèges de l'application (par exemple contrôler l'accès à certaines ressources) en fonction de l'utilisateur authentifié.

Lorsque l'API JAAS est utilisée, le développeur doit s'assurer des points suivants :

- JAAS s'appuyant sur le mécanisme de contrôle d'accès par défaut de JPSA, celui-ci doit être actif et configuré correctement. En particulier, le développeur doit éviter autant que faire se peut de modifier les classes permettant de mettre en œuvre le contrôle d'accès (par exemple, le gestionnaire de sécurité ou le gestionnaire de politique). S'il est amené à modifier ces classes, il doit s'assurer que ces modifications ne perturbent pas l'exécution de l'algorithme de contrôle d'accès implémenté notamment par la classe `java.security.AccessControler` ;
- l'utilisation de JAAS permet uniquement d'ajouter des permissions (donc des autorisations). Il convient donc de s'assurer que la politique mise en œuvre par le contrôle d'accès de JPSA, et qui s'applique à tous les utilisateurs, n'est pas trop permissive ;
- il convient de séparer le code de l'application effectuant des opérations critiques (authentification, appel à la méthode `doAs`, etc.) du code de l'application exécuté pour le compte d'un utilisateur. Cette séparation se concrétise par l'utilisation d'espaces de noms distincts (*packages* différents).

Impacts potentiels sur le processus de développement :

L'impact est élevé. Le développeur doit analyser les besoins de chacune des classes de l'application et des utilisateurs de ces classes afin de définir l'ensemble minimum des permissions qui doit être attribué à chaque classe en fonction de l'utilisateur qui l'exécute.

Références :

[1] [Gon03] Section 8.4, User-Centric Authentication and Authorization Using JAAS

[2] [Oak01] Section 15, Authentication and Authorization

[3] `http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html`

[4] `http://java.sun.com/j2se/1.5.0/docs/guide/security/jaas/tutorials/index.html`

[5] `http://java.sun.com/javase/6/docs/api/java/lang/SecurityManager.html`

Voir également :

- Fiche 23, page 53
- Fiche 25, page 57
- Fiche 26, page 59

5.3 Utilisation des mécanismes cryptographiques

Les recommandations suivantes concernent l'utilisation des services cryptographiques fournis par la bibliothèque standard de Java. Celle-ci spécifie uniquement des interfaces normalisées, l'implémentation des mécanismes étant fournies par des *providers*.

Identifiant : 33	**Nom :** Instanciation d'un service cryptographique	**Risque :** M

Problématique :

L'utilisation d'un service cryptographique de JCE suppose en premier lieu l'instanciation d'une implémentation particulière, fournie par un *provider*, par le biais de la méthode `getInstance` : par exemple, `Cipher c = Cipher.getInstance("RSA", "SUN")`. Cette méthode peut prendre en entrée les paramètres suivants :

- l'algorithme souhaité (dans l'exemple, RSA) ;
- le nom du fournisseur du service ou *provider* (dans l'exemple, SUN).

Le dernier paramètre est facultatif. En cas d'omission, l'ensemble des *providers* disponibles sur l'environnement d'exécution est parcouru par ordre de priorité, et la première implémentation correspondant à l'algorithme demandé est utilisée. L'ordre de priorité est configurable dans le fichier `<JAVA_HOME>/lib/security/java.security`. Cet ordre est spécifié par l'utilisateur (ou l'administrateur) pour l'ensemble des applications Java susceptibles d'être exécutées sur la plate-forme d'exécution.

Par conséquent, si ce paramètre est omis, l'application n'utilisera pas obligatoirement l'implémentation souhaitée à l'origine par le développeur. L'impact de cette omission est très variable selon le type de service concerné : utilisation d'un algorithme d'aléa non sûr, utilisation d'un algorithme de chiffrement moins performant, etc.

Recommandations :

Comme indiqué dans le rapport d'étude sur le langage Java [1], il est conseillé de toujours préciser le *provider* que l'on souhaite utiliser. Par exemple, pour une plate-forme native Windows, si le *provider* n'est pas précisé, le moteur de chiffrement retourné est celui du *provider* "SUN". Si le développeur souhaite utiliser un autre *provider*, il doit le préciser : `SecureRandom gen = SecureRandom.getInstance("RSA", "SunMSCAPI")`. Ce code permet d'utiliser le moteur de chiffrement RSA de Windows du *provider* `SunMSCAPI`. Pour Linux, seul le *provider* de Sun est disponible par défaut (mais l'installation d'autres *providers* est possible).

Impacts potentiels sur le processus de développement :

Faible.

Références :

[1] [Con09c] 6.4 Cryptographie

[2] `http://www.ibm.com/developerworks/java/jdk/security/60/FIPShowto.html`

Voir également :

- Fiche 34, page 77
- Fiche 35, page 80

Identifiant : 34	**Nom :** Configuration de la liste des *providers*	**Risque :** M

Problématique :

L'implémentation de certains services cryptographiques fait elle-même appel à d'autres services sans préciser les algorithmes et/ou le *provider* souhaités. S'agissant de composants ré-utilisés, le développeur de l'application n'a pas la maîtrise du code source de ces composants et ne peut appliquer les recommandations évoquées dans la fiche 33 (page 75). Ceci peut conduire à l'utilisation d'implémentations de services moins performantes ou moins sûres en raison de la liste de préférence des *providers* configurée par l'utilisateur ou l'administrateur de la plate-forme d'exécution.

Recommandations :

Le développeur doit fournir un manuel permettant à l'utilisateur ou l'administrateur de configurer l'ordre de préférence des *providers*.

Cette configuration est réalisée pour le JRE de Sun en éditant le fichier `<JAVA_HOME>/lib/security/java.security`. Les *providers* de confiance doivent être placés en premier et ceux non sûrs doivent être supprimés.

Le retrait d'un* provider *de la liste ne garantit pas que l'application ne l'utilisera pas. En effet, le* provider *n'est plus accessible par l'interface de JCA, mais il peut être chargé et utilisé manuellement à la demande de l'application.

La suppression du fichier `<JAVA_HOME>/lib/security/java.security` provoque l'attribution par défaut d'une liste de* providers *définie par la classe `java.security.Security`. Avec la bibliothèque standard de Sun ou d'OpenJDK, par ordre de priorité, les* providers *définis par défaut sont les suivants :

- ***sun.security.provider.Sun ;***
- ***sun.security.rsa.SunRsaSign ;***
- ***com.sun.net.ssl.internal.ssl.Provider ;***
- ***com.sun.crypto.provider.SunJCE ;***
- ***sun.security.jgss.SunProvider ;***
- ***com.sun.security.sasl.Provider.***

La seule manière de modifier la priorité d'un* provider *depuis une application Java est de le retirer et de le réinsérer de la liste des* providers *(voir la fiche suivante).

Par conséquent, dans le cas où l'administrateur du poste client souhaite maîtriser l'environnement d'exécution Java, il est recommandé de ne pas accorder aux applications Java les droits permettant de retirer ou de réinsérer des* providers *(voir la fiche suivante). Il doit donc configurer le gestionnaire de sécurité en tant que tel et ne pas accorder les permissions suivantes :

- ***java.security.SecurityPermission "insertProvider.*<nom du provider>*";***
- ***java.security.SecurityPermission "removeProvider.*<nom du provider>*";***
- ***java.security.SecurityPermission "removeProviderProperty.*<nom du provider>*";***
- ***java.security.SecurityPermission "loadProviderProperties.*<nom du provider>*".***

Impacts potentiels sur le processus de développement :

Faible.

Références :

[1] Rapport d'étude sur le langage Java, 6.4 Cryptographie

[2] `http://www.ibm.com/developerworks/java/jdk/security/60/FIPShowto.html`

Voir également :

- Fiche 33, page 75
- Fiche 35, page 80

Identifiant : 35	**Nom :** Configuration des *providers*	**Risque :** M

Problématique :

Dans le cas où l'application serait déployée dans un environnement non maîtrisé, le développeur n'a pas forcément la maîtrise de la configuration de l'environnement Java et notamment de l'ordre de préférence des *providers*. D'après le problème identifié dans la fiche 34 (page 77), des implémentations de services cryptographiques moins performantes ou moins sûres que celles souhaitées par le développeur peuvent être utilisées lors de l'exécution de l'application.

Recommandations :

Dans le cas où la configuration de l'environnement d'exécution n'est pas maîtrisée par le développeur de l'application, celui-ci doit s'assurer que cette configuration est conforme aux exigences de l'application en termes d'utilisation des implémentations des services cryptographiques. Pour cela, le développeur dispose de méthodes Java permettant de tester quels sont les *providers* utilisés et de déterminer l'ordre de préférence de la plate-forme d'exécution Java. Il peut également modifier dynamiquement cet ordre de préférence, sous réserve que le code de l'application dispose des droits nécessaires. Par conséquent, dans le cas où une application nécessite l'utilisation d'un *provider* particulier, il convient de placer ce dernier en tête de la liste des préférences. Pour cela, l'application doit retirer ce *provider*, puis le réinsérer en tête de liste via l'utilisation d'une classe privilégiée.

Ces méthodes sont :

- **`static Provider[] Security.getProviders();`**
- **`static void Security.removeProvider(String name);`**
- **`static void Security.insertProviderAt(Provider provider, int position).`**

Les méthodes `Security.insertProviderAt` et `Security.removeProvider` nécessitent de posséder les permissions `java.security.SecurityPermission` suivantes :

- `insertProvider.*` ;
- `removeProvider.*` ;

Comme indiqué dans la documentation de l'API Java [4], l'utilisation de ces méthodes demande une attention particulière. Pour réaliser les appels à ces méthodes via la méthode `doPrivileged`, le développeur devra suivre les recommandations de la fiche 29 (page 66).

Toutefois, dans un environnement non-maîtrisé, le code de l'application peut ne pas disposer de privilèges suffisants pour modifier la configuration. Dans ce cas de figure, la méthode `static Provider[] Security.getProviders()` peut être utilisée pour identifier les *providers* présents et pour conditionner l'arrêt prématuré de l'application si la configuration de l'environnement est considérée comme non sûre.

Impacts potentiels sur le processus de développement :

Faible.

Références :

[1] [Con09c], 6.4 Cryptographie

[2] `http://www.ibm.com/developerworks/java/jdk/security/60/FIPShowto.html`

[3] `http://java.sun.com/javase/6/docs/api/java/security/Security.html#insertProviderAt(java.security.Provider,%20int)`

[4] `https://www.securecoding.cert.org/confluence/display/java/SEC31-J.+Guard+doPrivileged+blocks+against+untrusted+invocations`

Voir également :

- Fiche 33, page 75
- Fiche 34, page 77
- Fiche 29, page 66

Identifiant : 36	**Nom :** Choix du générateur d'aléa	**Risque :** E

Problématique :

Les tests cryptographiques réalisés ont montré que le générateur SHA1-PRNG du *provider* Sun présente des défauts statistiques. Dans la mesure du possible, il n'est pas recommandé d'utiliser ce générateur.

De plus, dans la configuration par défaut du JRE de Sun sous Windows, l'implémentation SHA1-PRNG de SUN constitue le générateur d'aléa utilisé par défaut. Dans ce cas de figure, le développeur ne doit pas utiliser le générateur par défaut.

Recommandations :

Le développeur ne doit pas utiliser le générateur d'aléa SHA1-PRNG du *provider* Sun. Sur un système Linux, il est recommandé d'utiliser le générateur d'aléa par défaut `NativePRNG` du *provider* Sun. Sur un système Windows, il est recommandé d'utiliser un générateur de confiance autre que celui de Sun, le générateur de Windows `Windows-PRNG` du *provider* `SunMSCAPI`.

Impacts potentiels sur le processus de développement :

Faible.

Références :

[1] [Con09c], 6.4 Cryptographie

Voir également :

- Fiche 33, page 75
- Fiche 34, page 77
- Fiche 35, page 80

Identifiant : 37	**Nom :** Paramétrage du générateur d'aléa	**Risque :** E

Problématique :

Il existe des méthodes prenant en paramètre de manière facultative une référence vers un générateur d'aléa. Par exemple, la méthode `init` de la classe `KeyGenerator` correspond à ce type de méthode :

- `void init(int keysize);`
- `void init(int keysize, SecureRandom random).`

Si la référence vers le générateur d'aléa est omise, la méthode `init` utilisera le générateur d'aléa par défaut (potentiellement il s'agira de SHA1-PRNG de Sun). Par conséquent, en raison du problème identifié dans la fiche 36 (page 82), l'omission du paramètre concernant le générateur d'aléa peut provoquer l'utilisation d'un service non-sûr.

Recommandations :

Le développeur doit toujours renseigner le paramètre du générateur d'aléa afin d'ôter toute ambigüité.

Impacts potentiels sur le processus de développement :

Faible. Le développeur doit identifier les méthodes de ce type, et modifier le code en conséquence.

Références :

[1] [Con09c], 6.4 Cryptographie

Voir également :

- Fiche 36, page 82
- Fiche 38, page 84
- Fiche 39, page 85

Identifiant : 38	**Nom :** Paramétrage du générateur de clé	**Risque :** E

Problématique :

Pour certains mécanismes cryptographiques, il est possible d'appeler le constructeur de clés sans préciser les paramètres de génération de clés. Le code 7 présenté en annexe page 105 constitue un exemple de génération « simplifiée ». Cette utilisation implique une génération à partir d'éléments pré-calculés pour les paramètres p, q et g des clés DSA de 512, 768 et 1024 bits. En effet, ce générateur utilise un système de cache permettant de stocker des triplets (p, q, g) pour qu'ils soient réutilisés. Ce dispositif permet d'accélérer les calculs au détriment de la diversification des clés. Il est également implémenté pour le générateur de clés Diffie-Hellman du provider Sun.

Recommandations :

Les appels aux constructeurs des générateurs de clés doivent toujours être effectués en passant explicitement les paramètres de génération lorsque cela est possible.

Le développeur préférera donc, pour la génération de clés DSA, le code suivant :

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

Ce code montre comment forcer la génération des éléments p, q et g.

Impacts potentiels sur le processus de développement :

Faible.

Références :

[1] [Con09c], 6.4 Cryptographie

[2] `http://java.sun.com/javase/6/docs/api/java/security/interfaces/DSAKeyPairGenerator.html`

Voir également :

- Fiche 36, page 82
- Fiche 39, page 85

Identifiant : 39	**Nom :** Paramétrage des algorithmes de chiffrement	**Risque :** M

Problématique :

L'instanciation d'un service de chiffrement pour un algorithme donné peut être réalisée de deux manières, illustrées par les exemples suivants :

- `Cipher c = Cipher.getInstance("AES", "PROVIDER1");`
- `Cipher c = Cipher.getInstance("AES/CBC/PKCS5PADDING", "PROVIDER1").`

La seconde méthode permet de préciser le mode de chiffrement et le mode de bourrage utilisés. La première méthode ne précise pas ces éléments. Ainsi, les modes de chiffrement et de bourrage sont définis par défaut. Les paramètres par défaut dépendent du *provider* utilisé. Par conséquent, dans le premier cas, le développeur ne maîtrise pas ces paramètres qui peuvent conduire à des éléments cryptographiques faibles.

Recommandations :

Lors de l'instanciation d'un service de chiffrement, le développeur doit utiliser la deuxième méthode afin de préciser les modes de chiffrement et de bourrage souhaités.

Impacts potentiels sur le processus de développement :

Faible.

Références :

[1] [Oak01] Section 13, *Cipher-Based Encryption*

Voir également :

- Fiche 36, page 82
- Fiche 38, page 84

5.4 Interfaçage avec l'environnement natif

Identifiant : 40	**Nom :** Interface native (JNI)	**Risque :** E

Problématique :

L'interface JNI permet d'appeler des fonctions natives, implémentées en C ou en C++, depuis le code Java. Elle permet également à ces fonctions de manipuler les objets Java. Les fonctions interfacées doivent êtres déclarées en tant que méthode `native` au sein d'une classe Java. Cette fonctionnalité permet par exemple au code Java d'accéder aux ressources natives du système. Elle permet également à une application Java d'implémenter nativement certaines opérations (par exemple, pour des raisons d'optimisation) ou de s'interfacer avec une bibliothèque existante développée en C/C++. L'interface avec d'autres langages est possible mais n'est pas supportée nativement par JNI. Il est donc nécessaire de développer une interface en C/C++ commune aux deux environnements (par exemple Java et Caml).

L'utilisation de cette fonctionnalité n'est pas sans risque : le code natif n'est pas portable, il est sujet aux vulnérabilités intrinsèques aux langages utilisés (absence de gestion de la mémoire, pas de vérification de la taille des tampons, etc.) et il ne fait pas l'objet de vérifications de contrôle d'accès.

Recommandations :

L'utilisation de méthodes natives doit être restreinte aux cas strictement nécessaires (interface avec une bibliothèque non portée en Java, accès à une ressource native non fournie par la bibliothèque standard). En raison des risques accrus, il convient de définir une méthode enveloppante permettant d'appliquer des vérifications sur les données passées en paramètres et sur les accès réalisés.

Ce type d'interface doit être utilisé avec parcimonie. Toutefois, il existe des cas où elle est incontournable. Il est alors nécessaire d'appliquer les recommandations suivantes :

- vérifier les valeurs des indices et des longueurs de tableaux passés en paramètre, de préférence dans le code Java ;
- pour les objets censés n'être lus que par le code natif, passer en paramètre une copie de l'objet (utilisation de la méthode `clone`) ;
- faire précéder l'appel au code natif par un point de contrôle d'accès en définissant éventuellement une permission adéquate (en particulier lors de l'accès à une ressource native) ;
- de manière générale, restreindre l'accès aux méthodes natives au code le nécessitant explicitement. Éviter en particulier de déclarer la méthode native `public`.

L'utilisation de JNI au sein d'un composant réutilisé (COTS) est potentiellement problématique. En effet, il n'est alors pas toujours possible d'appliquer les recommandations de ce document. De plus la détection et la restriction de son utilisation à l'aide des mécanismes standards de Java ne sont pas triviales.

La prise en compte de ces recommandations est illustrée par l'exemple suivant :

```
public final class NativeMethodWrapper {
  // La méthode native qui doit être contrôlée
  private native void nativeOperation(byte[] data, int offset,
      int len);
  // La méthode enveloppante qui vérifie les données en entrée
      et appel le gestionnaire de sécurité
  public void doOperation(byte[] data, int offset, int len) {
    // Vérification des permissions
    SecurityManager sm = System.getSecurityManager();
    if(sm != null) {
      sm.checkPermission(new NativeOperationPermission());
    }
    if (data == null) {
      throw new NullPointerException();
    }
    // copie des données susceptibles d'être modifiées
    data = data.clone();
    // vérification des paramètres
    if ((offset < 0) || (len < 0) || (offset > (data.length -
       len))) {
      throw new IllegalArgumentException();
    }
    nativeOperation(data, offset, len);
  }

  static {
    // Chargement de la bibliothèque native
    System.loadLibrary("NativeMethodLib");
  }
}
```

Impacts potentiels sur le processus de développement :

Faible.

Références :

[1] `https://www.securecoding.cert.org/confluence/display/java/SEC30-J.+Define+wrappers+around+native+methods`

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

[3] `http://java.sun.com/javase/6/docs/technotes/guides/jni/index.html`

[4] `http://cwe.mitre.org/` *Common Weakness Enumeration : CWE-111 : Direct Use of Unsafe JNI*

Voir également :

- Fiche 29, page 66
- Fiche 28, page 63

5.5 Chargement de classes

Le chargeur de classes est un élément critique pour la sécurité des applications Java :

- à partir d'un nom de classe, il identifie le fichier de classe correspondant et fournit son contenu à la JVM sous forme d'un tableau d'octets (définition de classe). Ce mécanisme peut faire intervenir différents chargeurs de classes, puisqu'un chargeur de classes peut déléguer le chargement à une autre instance. Cette délégation repose sur une hiérarchie définie à l'exécution, lors de la mise en place des chargeurs ;
- il permet de mettre en place des espaces de noms distincts (le type d'une classe étant déterminé par le nom de la classe et l'instance du chargeur de classes qui l'a défini). Des classes identiques (issu du même fichier de classes) possède ainsi des types incompatibles si elles ont été définies par des chargeurs de classes distincts. Ce mécanisme permet au code de l'application de mettre en œuvre une forme restreinte de confinement ;
- il participe au contrôle d'accès orienté code de Java (JPSA). Pour chaque classe qu'il définit, il doit réaliser les opération suivantes :
  - il délègue la vérification de signature,
  - il associe les domaines de sécurité correspondant à la classe,
  - il associe éventuellement certaines permissions statiques à la classe. Ces permissions s'ajoutent éventuellement aux permissions dynamiques fournies par le gestionnaire de politique.

Identifiant : 41	**Nom :** Définition d'un chargeur de classes	**Risque :** M

Problématique :

Le mécanisme de chargement de classes est un mécanisme complexe et dont le contrôle par un attaquant peut mener au piégeage de l'application, à des dénis de services ou au contournement de certains mécanismes de sécurité (signature de classes, contrôle d'accès). Par défaut, l'environnement d'exécution utilise les chargeurs de classes fournis par la bibliothèque standard et la JVM (chargeur de classes initial, utilisé pour charger les classes de la bibliothèque).

Le développeur peut instancier, au sein de l'application Java, des chargeurs de classes. Comme il s'agit d'un mécanisme critique, l'utilisation des chargeurs de classes doit être restreinte par le mécanisme de contrôle d'accès de Java (les chargeurs de classes définis dans la bibliothèque standard implémentent des points de contrôle *ad-hoc*).

Le développeur peut également implémenter ses propres chargeurs de classes en définissant des classes qui héritent de `java.lang.ClassLoader`. Toutefois, s'agissant d'un mécanisme critique, il convient de suivre certaines recommandations afin d'éviter qu'un attaquant utilise ce chargeur de classes dans le but de piéger l'application ou contourner le contrôle d'accès.

Recommandations :

L'utilisation d'un chargeur de classes nécessite par défaut que le code qui instancie un chargeur de classes possède la permission `RuntimePermission("createClassLoader")`. Définir un chargeur de classe est une opération très critique pour la sécurité. Il convient donc d'accorder cette permission uniquement à du code de confiance.

En raison de la complexité évoquée dans la section précédente, le développement d'un chargeur de classes doit donc être envisagé uniquement dans les cas qui le nécessitent explicitement : adaptation du processus de chargement à un mode de distribution particulier des classes de l'application, mise en œuvre d'opérations spécifiques (par exemple, des contrôles additionnels), cloisonnement de différents blocs fonctionnels de l'application Java s'exécutant sur l'instance de la JVM. Il doit être assuré par un développeur Java qui maîtrise ce mécanisme.

Le développeur doit garder à l'esprit qu'il est difficile de mettre en place un cloisonnement strict à l'aide du seul mécanisme de chargement de classes. Il est en particulier possible pour des classes appartenant à des domaines distincts d'échanger de l'information via les classes chargées par un ancêtre commun à leur chargeur de classes respectifs (par exemple, le chargeur de classes système de la bibliothèque standard ou le chargeur de classes initial). Un confinement total supposerait de pouvoir charger l'ensemble des classes de la bibliothèque via un chargeur de classes Java défini par l'utilisateur, ce qui n'est pas réalisable.

Il est conseillé de respecter autant que faire se peut le comportement par défaut (déléguer avant toute tentative de chargement) afin d'éviter tout problème de conflits de typage non désirés.

Il est impératif que le développeur s'assure que le chargeur de classes qu'il développe comprend les éléments suivants :

- **des points de contrôle permettant de limiter l'utilisation du chargeur de classes ;**
- **une délégation de la vérification de la signature des classes, si le chargeur est susceptible de lire des archives JAR ;**
- **un ensemble de code permettant d'associer les domaines de protection adéquat à chaque classe définie par le chargeur de classes.**

Afin de respecter les règles citées précédemment, le développeur peut faire hériter le chargeur de classes qu'il développe de la classe `java.net.URLClassLoader` ou de la classe `java.security.SecureClassLoader`. Ces classes définissent des méthodes qui facilitent l'implémentation des règles évoquées ci-dessus.

L'association de permissions statiques par le chargeur de classes n'est pas recommandé, il est préférable de déléguer l'association des permissions au gestionnaire de la politique (permissions dynamiques). Cette fonctionnalité est présente depuis la version 1.4 de Java. Si, pour des raisons de compatibilité, il s'avère nécessaire que les permissions soient associées par le chargeur de classes (permissions statiques), celui-ci doit s'appuyer autant que faire se peut sur le support externe d'expression de la politique de contrôle (généralement, un fichier texte). Il est déconseillé d'associer statiquement des permissions spécifiées au sein même du code du chargeur de classes, car les permissions accordées de la sorte n'apparaissent pas dans le support externe d'expression de la politique, ce qui nuit à l'analyse de la politique.

Impacts potentiels sur le processus de développement :

Faible.

Références :

[1] [Gon03] *Section 4, Secure Class Loading*

[2] [Oak01] *Section 16, Java Class Loaders*

[3] `http://java.sun.com/javase/6/docs/api/java/lang/ClassLoader.html`

[4] `https://www.securecoding.cert.org/confluence/display/java/SEC10-J.+Call+the+superclass\%27s+getPermissions+method+when+writing+a+custom+class+loader`

[5] `https://www.securecoding.cert.org/confluence/display/java/ENV33-J.+Do+not+grant+RuntimePermission+with+target+createClassLoader`

Voir également :

- Fiche 30, page 69
- Fiche 42, page 93

Identifiant : 42	**Nom :** Appels implicites au chargeur de classes	**Risque :** M

Problématique :

Certaines méthodes de l'API standard de Java utilisent le *class loader* de la méthode appelante directe (la méthode qui appelle l'une de ces méthodes) afin de réaliser certaines tâches critiques. Par exemple, la méthode `loadLibrary` de la classe `java.lang.System` utilise le *class loader* de la méthode appelante directe afin de charger des bibliothèques natives. Il s'agit d'une opération critique puisque le chargement d'une bibliothèque permet l'utilisation de méthodes natives. Il est possible que le *class loader* de cette méthode dispose de privilèges accrus lui permettant de charger les bibliothèques natives à la différence des autres *class loaders*. Une attention particulière doit donc être portée sur l'invocation de ces méthodes afin qu'elles ne puissent être utilisées par du code non privilégié pour charger des bibliothèques auxquelles il n'a pas accès directement (son *class loader* ne disposant pas des privilèges nécessaires).

Les méthodes suivantes ont un comportement similaire à `System.loadLibrary` et peuvent potentiellement être utilisées pour une escalade de privilèges :

- java.lang.Class.forName
- java.lang.Package.getPackage(s)
- java.lang.Runtime.load
- java.lang.Runtime.loadLibrary
- java.lang.System.load
- java.lang.System.loadLibrary
- java.sql.DriverManager.getConnection
- java.sql.DriverManager.getDriver(s)
- java.sql.DriverManager.deregisterDriver
- java.util.ResourceBundle.getBundle

Recommandations :

L'utilisation de ces méthodes nécessite de filtrer les interactions de la méthode appelante avec le reste du code.

Les recommandations suivantes doivent être appliquées :

- le code non privilégié ne doit pas pouvoir spécifier directement les paramètres de ces méthodes (par exemple, le nom et le chemin des bibliothèques natives à charger) ;
- les références sur les objets retournés par ces méthodes ne doivent être propagées qu'au code disposant des mêmes privilèges.

Impacts potentiels sur le processus de développement :

Faible.

Références :

[1] `https://www.securecoding.cert.org/confluence/display/java/SEC33-J.+Do+not+expose+standard+APIs+that+use+the+immediate+caller%27s+class+loader+instance+to+untrusted+code`

[2] `http://java.sun.com/security/seccodeguide.html#gcg`

Voir également :

- Fiche 41, page 90
- Fiche 30, page 69

6 ANNEXES

6.1 Utilisation des *threads* et de la programmation concurrente

Le code 1 est un exemple de serveur implémentant le protocole écho et créant un *thread* par connexion cliente. C'était la seule manière de procéder avant l'apparition de `nio`.

Listing 1 – Mauvaise utilisation des *threads* là où `nio` serait plus adapté : exemple de code **non conforme**

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class EchoServerThreaded {
  private static int PORT = 1234;

  public static void main(String[] args) {
    // Creation d'un socket serveur
    ServerSocket serverSocket = null;
    try {
      serverSocket = new ServerSocket(PORT);
      System.out.println("Server created on " + PORT);
    } catch (IOException e) {
      System.err.println("Could not listen to port "+ PORT);
      e.printStackTrace();
      System.exit(-1);
    }

    try {
      while (true) {
        final Socket clientSocket = serverSocket.accept();
        System.out.println("Client " + clientSocket.getInetAddress()
           + ":" + clientSocket.getPort() + " connected");
        Thread clientThread = new Thread(new Runnable() {
          public void run() {
            System.out.println("Starting client thread");
            InputStream is;
            try {
              is = clientSocket.getInputStream();

              OutputStream os = clientSocket.getOutputStream();
              while (true) {
                int r = is.read();
```

```
                os.write(r);
              }
            } catch (IOException e) {
              System.out.println("Stopped client thread");
            }
          }
        });
        clientThread.start();

      }
    }
    catch (IOException e) {
      System.err.println("Couldn't accept connection");
      e.printStackTrace();
    }
  }
}
```

Le code 2 est un exemple de serveur implémentant le protocole écho et utilisant `nio` en remplacement des *threads*.

Listing 2 – Utilisation de `nio` en remplacement d'un système de *threads* : exemple de code conforme

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Iterator;

public class EchoServerNio {
  private static final int PORT = 1234;
  private static final int BUFSIZE = 256;

  public static void main(String[] args){
    Selector selector;
    try {
      // Creation du selecteur
      selector = Selector.open();
      // Creation du ServerSocket
      ServerSocketChannel socketChannel = ServerSocketChannel.open();
      socketChannel.socket().bind(new InetSocketAddress(PORT));
      socketChannel.configureBlocking(false);
      // Enregistrement du ServerSocket sur le selecteur.
      socketChannel.register(selector, SelectionKey.OP_ACCEPT);
    }
```

```
    catch(IOException e) {
      System.err.println("Server initialisation failure:");
      e.printStackTrace();
      return;
    }

    while(true){
      try {
        selector.select();
        Iterator<SelectionKey> keyIter = selector.selectedKeys().
           iterator();
        while (keyIter.hasNext()) {
          SelectionKey key = (SelectionKey) keyIter.next();
          if (key.isAcceptable()) {
            // Une nouvelle demande de connection a été reçue, on
               récupère le socket, puis
            // on l'ajoute dans le sélecteur
            SocketChannel clientChannel = ((ServerSocketChannel) key.
               channel()).accept();
            System.out.println("Client " + clientChannel.socket().
               getInetAddress()
                + ":" + clientChannel.socket().getPort() + "
                   connected");
            clientChannel.configureBlocking(false);
            // Enregistrement de la nouvellle connection dans le
               sélecteur
            clientChannel.register(selector, SelectionKey.OP_READ,
               ByteBuffer.allocate(BUFSIZE));

          }
          else if (key.isReadable()) {
            SocketChannel clientChannel = (SocketChannel)key.channel
               ();
            ByteBuffer buffer = (ByteBuffer)key.attachment();
            try {
              if (clientChannel.read(buffer) > 0) {
                // Si on a lu des octets, on passe le selecteur en
                   OP_WRITE afin qu'on soit
                // notifié de la possibilité d'écrire des données
                   dans la connection client.
                key.interestOps(SelectionKey.OP_READ | SelectionKey.
                   OP_WRITE);
              }
              else {
                System.out.println("Client disconnected");
                clientChannel.close();
              }
            }
```

```
            catch(IOException e) {
              //Si pour n'importe quelle raison on n'arrive pas à
                 lire, on ferme le canal
              System.out.println("Client disconnected");
              clientChannel.close();
            }
          }
          else if (key.isWritable()) {
            SocketChannel clientChannel = (SocketChannel)key.channel
               ();
            ByteBuffer buffer = (ByteBuffer)key.attachment();
            buffer.flip();
            clientChannel.write(buffer);
            if (!buffer.hasRemaining()) {
              // Si on n'a plus rien à écrire, on ne souhaite plus
                 être mis au courant
              // du fait qu'il reste de la place pour écrire des
                 données dans le channel
              key.interestOps(SelectionKey.OP_READ);
            }
            buffer.compact();

          }
          keyIter.remove();
        }
      }
      catch(IOException e) {
        System.err.println("Runtime failure:");
        e.printStackTrace();
      }
    }
  }
}
```

Le code 3 est un exemple de programme présentant une Interface Homme Machine et effectuant des traitements potentiellement longs en tâche de fond. Cet exemple met en avant l'utilisation de la programmation concurrente en Java sans manipulation directe des *threads* et utilisation explicite du mot-clé `synchronized`. Il repose sur l'intégration de l'abstraction `java.util.concurrent.Future` dans le framework graphique Swing.

Ce programme illustre le cas d'utilisation d'un programme avec une IHM riche (basée sur l'API Swing), qui nécessite de sous-traiter des traitements lourds à une tâche de fond, qui s'exécute dans un *thread*. Nous avons choisi à cette fin l'exemple du calcul de factorielle.

Un *thread*, appelé *thread* graphique, est chargé de gérer les interactions utilisateurs. Les traitements sont gérés par un objet héritant de `SwingWorker`, qui est instancié à chaque fois que l'utilisateur active le bouton `Calculer`. Le calcul est effectué dans la méthode `doInBackground`, qui s'exécute dans le contexte d'un *thread* créé par l'environnement Java pour cette occasion. Une fois le calcul effectué, la méthode `doInBackground` renvoie le résultat. L'environnement Java appelle alors la méthode `done` dans le contexte du *thread* graphique. Le programmeur peut alors récupérer le résultat de son calcul par la méthode `get()`.

Listing 3 – Utilisation des *threads* avec le framework graphique Swing : exemple de code conforme

```
/*
 * Fichier SwingThreadingExample.java
 */
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.math.BigInteger;

import javax.swing.*;

public class SwingThreadingExample implements ActionListener {
  JTextField valeur = new JTextField("1");
  JTextField resultat = new JTextField("");

  SwingThreadingExample() {
    JFrame frame = new JFrame("Vertical");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    Box verticalBox = Box.createVerticalBox();
    verticalBox.add(new JLabel("Valeur␣:"));
    verticalBox.add(valeur);
    JButton start = new JButton("Calculer");
    start.setActionCommand("start");
    start.addActionListener(this);
    verticalBox.add(start);
    verticalBox.add(new JLabel("Resultat␣:"));
    verticalBox.add(resultat);
```

```
    frame.getContentPane().add(verticalBox, BorderLayout.CENTER);
    frame.setSize(150, 150);
    frame.setVisible(true);

    // Affiche la fenêtre
    frame.pack();
    frame.setVisible(true);
  }

  /**
   * Cette méthode est appelée dans le contexte du thread graphique
   */
  public void actionPerformed(ActionEvent e) {
    if ("start".equals(e.getActionCommand())) {
      FactorialWorker worker = new FactorialWorker(this,
          Integer.decode(valeur.getText()).longValue());
      worker.execute();
    }
  }

  /**
   * Cette méthode est appelée dans le contexte du thread graphique
   */
  void setResult(BigInteger value) {
    resultat.setText(value.toString());
  }

  /**
   * Cette méthode est appelée dans le contexte du thread graphique
   */
  private static void createAndShowGUI() {
    new SwingThreadingExample();
  }

  /**
   * Cette méthode est appelée dans le contexte du thread
   * principal (thread de démarrage).
   */
  public static void main(String[] args) {
    //Event dispatching thread
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        createAndShowGUI();
      }
    });
  }
}
```

```
/*
 * Fichier FactorialWorker.java
 */
import java.math.BigInteger;
import java.util.concurrent.ExecutionException;

import javax.swing.SwingWorker;

/*
 * Cette classe implémente un Worker chargé de calculer !n.
 */
public class FactorialWorker extends SwingWorker<BigInteger, Void> {
  long input;
  SwingThreadingExample window;

  FactorialWorker(SwingThreadingExample _window, long _input) {
    input = _input;
    window = _window;
  }

  /*
   * Cette méthode effectue le calcul, et est exécutée dans le
   * contexte d'un Worker Thread.
   */
  @Override
  protected BigInteger doInBackground() throws Exception {
    BigInteger res = BigInteger.valueOf(input);
    if(input != 0) {
      long i = input;
      while(i > 1) {
        res = res.multiply(BigInteger.valueOf(input -1));
        --i;
      }
    }
    else {
      res = BigInteger.ONE;
    }
    return res;
  }

  /*
   * Cette méthode est appelée dans le contexte du thread qui gère
   * l'IHM, et récupère le résultat du calcul par un appel à la
   * méthode get().
   */
   @Override
   public void done() {
     try {
```

```
        window.setResult(get());
      }
      catch (InterruptedException e) {
        e.printStackTrace();
      }
      catch (ExecutionException e) {
        e.printStackTrace();
      }
    }
}
```

6.2 Contrôle d'accès

Le code de l'exemple 4 n'est pas conforme dans l'utilisation de la méthode `doPrivileged` pour deux raisons. Premièrement, les opérations de la méthode `openPasswordFile` sont toujours exécutées de manière privilégiée, quelle que soit la méthode appelante, car l'appel à `doPrivileged` est réalisé à l'intérieur de la méthode `openPasswordFile`. Cette méthode renvoyant une référence sur la ressource lue (le fichier de mots de passe), il existe un risque d'élévation de privilège, n'importe quelle méthode de l'application pouvant appeler la méthode `openPasswordFile` et obtenir une référence sur la ressource protégée. Deuxièmement, le code de l'application contrôle totalement la ressource, puisque le nom du fichier de mot de passe est passé en paramètre à la méthode `openPasswordFile`.

Listing 4 – Utilisation de `doPrivileged` : exemple de code **non conforme**

```
public class Password {
  public static void changePassword(String password_file)
      throws FileNotFoundException {
    FileInputStream fin;
    fin = openPasswordFile(password_file);
  }
  public static FileInputStream openPasswordFile(String pwd_file)
      throws FileNotFoundException {
    // Les variables locales accédées dans la classe interne anonyme
       doivent être déclarées final
    // L'utilisation d'un tableau à un élément permet à la classe
       interne de retourner une valeur
    final FileInputStream f[]={null};
    final String file =  pwd_file;
    // Utilisation du mode privilégié pour accéder au fichier de mots
    // de passe confidentiels
    AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() {
        try {
          f[0] = new FileInputStream("c:\\" + file);
```

```
        }catch(FileNotFoundException cnf){
          System.err.println(cnf.getMessage());
        }
        return null;
      }
    });
    return f[0];
    // Ce code retourne une référence sur les objets obtenus par la
    // méthode privilégiée, ce qui n'est pas recommandé
  }
}
```

Le code de l'exemple 5 applique les recommandations sur l'utilisation de la méthode `doPrivileged`. L'appel de la méthode `doPrivileged` est maintenant réalisé à l'intérieur de la méthode `changePassword` qui ne renvoie pas de référence sur la ressource obtenue (le fichier de mot de passe). La méthode `openPasswordFile` renvoie certes une référence sur le fichier de mot de passe, mais elle ne peut accéder à ce fichier de mot de passe que lorsqu'elle est exécutée en mode privilégié par une méthode possédant la permission de lire le fichier de mot de passe. Si la politique de contrôle d'accès est correctement spécifiée, seules les méthodes de la classe `Password` possèdent cette permission. Le code de l'application n'a donc pas accès directement à une référence sur le fichier de mot de passe. Pour les mêmes raisons, le code de l'application ne peut spécifier directement l'emplacement du mot de passe. Les exceptions sont traitées localement, au sein de la méthode `changePassword`, afin d'éviter toute fuite d'information concernant l'emplacement du fichier de mot de passe.

Listing 5 – Utilisation de `doPrivileged` : exemple de code conforme

```
public class Password {
  private static void changePassword() {
    final String password_file = "password";
    final FileInputStream f[] = {null};
    // Utilisation du mode privilégié pour accéder au fichier de mot
    // de passes confidentiels
    AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() {
        try {
          f[0] = openPasswordFile(password_file);  //appel à la
              méthode privilégiée
        }
        catch(FileNotFoundException cnf) {
          System.err.println("Error: Operation could not be performed
              "); }
          return null;
        }
    });
  // Les différentes opération telles que la vérification des mots de
  // passe peuvent être réalisées ici
  }
```

```
  private static FileInputStream openPasswordFile(String
      password_file) throws FileNotFoundException {
    FileInputStream f = new FileInputStream("c:\\" + password_file);
    // Les opération de lecture et d'écriture sur le fichier de mot
    // de passe peuvent être réalisées ici
    return f;
  }
}
```

Lorsque l'emplacement de la ressource n'est pas une donnée confidentielle et de manière générale lorsque que les exceptions susceptibles d'être levées par le code privilégié ne contiennent aucune donnée confidentielle, il est possible de transmettre au code de l'application les exceptions levées par le code privilégié. Il convient dans ce cas d'utiliser la forme de la méthode `doPrivileged` utilisant une instance de la classe `PrivilegedExceptionAction`. Dans l'exemple 6, l'applet ne peut accéder directement au système de fichier et fait appel à un code privilégié afin de charger les polices de caractères. Le nom des fichiers de polices ne constituant pas une donnée confidentielle, les exceptions du code privilégié sont transmises à l'application afin que celle-ci traite au mieux les problèmes d'accès aux fichiers.

Listing 6 – Utilisation de `doPrivileged` : exemple de code conforme

```
public static void readFont() throws FileNotFoundException {
  final String font_file = "fontfile";
  //Utilisation du mode privilégié pour accéder au fichier de polices
  try {
    final InputStream in =
      AccessController.doPrivileged(new PrivilegedExceptionAction<
          InputStream>() {
        public InputStream run() throws FileNotFoundException {
          return openFontFile(font_file); //appel à la méthode
              privilégiée
        }
    });
    // Les différentes opération sur la police peuvent être
    // réalisées ici
  }
  catch (PrivilegedActionException exc) {
    Exception cause = exc.getException();
    if (cause instanceof FileNotFoundException) {
      throw (FileNotFoundException)cause;
    }
    else {
      throw new Error("Unexpected␣exception␣type",cause);
    }
  }
}
```

6.3 Mécanismes cryptographiques

Listing 7 – Exemple de mauvaise génération de clés DSA : exemple de code **non conforme**

```
KeyPairGenerator gen;
KeyPair keys;
gen.initialize (1024 , new SecureRandom());
keys = gen.generateKeyPair();
```

[Blo01] Bloch,, Joshua. *Effective Java Programming Language Guide*. Sun Microsystems, Inc., Mountain View, CA, USA, 2001.

[Cam00] 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.

[Cla08] Claude Delannoy. *Programmer en Java (3ème édition)*. Eyrolles, 2008.

[Con09a] Consortium JAVASEC. Comparatif des compilateurs. Technical Report Livrable 2.1 dans le CCTP JAVASEC, Silicom Région Ouest - Amossys - INRIA Rennes Bretagne Atlantique - SGDN, 2009.

[Con09b] Consortium JAVASEC. Comparatif des JVM. Technical Report Livrable 2.2 dans le CCTP JAVASEC, Silicom Région Ouest - Amossys - INRIA Rennes Bretagne Atlantique - SGDN, 2009.

[Con09c] Consortium JAVASEC. Rapport sur le langage Java. Technical Report Livrable 1.1 dans le CCTP JAVASEC, Silicom Région Ouest - Amossys - INRIA Rennes Bretagne Atlantique - SGDN, 2009.

[Con09d] Consortium JAVASEC. Rapport sur les modèles d'exécution Java. Technical Report Livrable 1.2 dans le CCTP JAVASEC, Silicom Région Ouest - Amossys - INRIA Rennes Bretagne Atlantique - SGDN, 2009.

[Con09e] Consortium JAVASEC. Stratégie d'évolution et d'implémentation d'une JVM pour l'exécution d'applications de sécurité. Technical Report Livrable 2.3 dans le CCTP JAVASEC, Silicom Région Ouest - Amossys - INRIA Rennes Bretagne Atlantique - SGDN, 2009.

[Eck02] Eckel,, Bruce. *Thinking in Java*. Prentice Hall Professional Technical Reference, 2002.

[Fla05] Flanagan,, David. *Java In A Nutshell, 5th Edition*. O'Reilly Media, Inc., 2005.

[Gon03] Gong,, Li and Ellison,, Gary. *Inside Java(TM) 2 Platform Security : Architecture, API Design, and Implementation*. Pearson Education, 2003.

[Hor08] Horstmann, Cay S and Cornell, Gary. *Core Java. Revised and Updated for Java SE 6 ; 8th ed.* Prentice-Hall, Upper Saddle River, NJ, 2008.

[Jam05] 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.

[Jos08] Joshua Bloch. *Effective Java, 2nd edition*. Addison Wesley, 2008.

[Kal04] Kalinovsky,, Alex. *Covert Java : Techniques for Decompiling, Patching, and Reverse Engineering*. Pearson Higher Education, 2004.

[Oak01] Oaks,, Scott. *Java Security Second Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[Tim99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.